```
1     typedef   struct  Exec    Exec;
2     struct    Exec
3     {
4       long    magic;          /* magic number */
5       long    text;           /* size of text segment */
6       long    data;           /* size of initialized data */
7       long    bss;            /* size of uninitialized data */
8       long    syms;           /* size of symbol table */
9       long    entry;          /* entry point */
10      long    spsz;           /* size of pc/sp offset table */
11      long    pcsz;           /* size of pc/line number table */
12    };
13
14    #define   _MAGIC(b)       ((((4*b)+0)*b)+7)
15    #define   A_MAGIC         _MAGIC(8)       /* 68020 */
16    #define   I_MAGIC         _MAGIC(11)      /* intel 386 */
17    #define   J_MAGIC         _MAGIC(12)      /* intel 960 */
18    #define   K_MAGIC         _MAGIC(13)      /* sparc */
19    #define   V_MAGIC         _MAGIC(16)      /* mips 3000 */
20    #define   X_MAGIC         _MAGIC(17)      /* att dsp 3210 */
21    #define   M_MAGIC         _MAGIC(18)      /* mips 4000 */
22    #define   D_MAGIC         _MAGIC(19)      /* amd 29000 */
23    #define   E_MAGIC         _MAGIC(20)      /* arm 7-something */
24    #define   Q_MAGIC         _MAGIC(21)      /* powerpc */
25    #define   N_MAGIC         _MAGIC(22)      /* mips 4000 LE */
26    #define   L_MAGIC         _MAGIC(23)      /* dec alpha */
27
28    typedef   struct  Sym     Sym;
29    struct    Sym
30    {
31      long    value;
32      char    type;
33      char    *name;
34    };
```

```
typedef struct Conf       Conf;
typedef struct FPsave     FPsave;
typedef struct ISAConf    ISAConf;
typedef struct Label      Label;
typedef struct Lock       Lock;
typedef struct MMU        MMU;
typedef struct Mach       Mach;
typedef struct Notsave    Notsave;
typedef struct PCArch     PCArch;
typedef struct Pcidev     Pcidev;
typedef struct PCMmap     PCMmap;
typedef struct Page       Page;
typedef struct PMMU       PMMU;
typedef struct Proc       Proc;
typedef struct Segdesc    Segdesc;
typedef struct Ureg       Ureg;
typedef struct Vctl       Vctl;

/*
 *  parameters for sysproc.c
 */
#define AOUT_MAGIC        (I_MAGIC)

struct Lock
{
  ulong   key;
  ulong   sr;
  ulong   pc;
  Proc    *p;
  ushort  isilock;
};

struct Label
{
  ulong   sp;
  ulong   pc;
};


/*
 * FPsave.status
 */
enum
{
  FPinit,
  FPactive,
  FPinactive,
};

struct    FPsave
{
  ushort  control;
  ushort  r1;
  ushort  status;
  ushort  r2;
  ushort  tag;
  ushort  r3;
  ulong   pc;
  ushort  selector;
  ushort  r4;
  ulong   operand;
  ushort  oselector;
  ushort  r5;
  uchar   regs[80];       /* floating point registers */
};

struct Conf
{
  ulong   nmach;          /* processors */
  ulong   nproc;          /* processes */
  ulong   monitor;        /* has monitor? */
  ulong   npage0;         /* total physical pages of memory */
  ulong   npage1;         /* total physical pages of memory */
  ulong   npage;          /* total physical pages of memory */
  ulong   upages;         /* user page pool */
  ulong   nimage;         /* number of page cache image headers */
  ulong   nswap;          /* number of swap pages */
  int     nswppo;         /* max # of pageouts per segment pass */
  ulong   base0;          /* base of bank 0 */
  ulong   base1;          /* base of bank 1 */
  ulong   copymode;       /* 0 is copy on write, 1 is copy on reference */
  ulong   ialloc;         /* max interrupt time allocation in bytes */
  ulong   pipeqsize;      /* size in bytes of pipe queues */
};

/*
 *  MMU stuff in proc
 */
#define NCOLOR 1
struct PMMU
{
  Page*   mmupdb;                 /* page directory base */
  Page*   mmufree;                /* unused page table pages */
  Page*   mmuused;                /* used page table pages */
};

/*
 *  things saved in the Proc structure during a notify
 */
struct Notsave
{
  ulong   svflags;
  ulong   svcs;
  ulong   svss;
};

#include "../port/portdat.h"

typedef struct {
  ulong   link;                   /* link (old TSS selector) */
  ulong   esp0;                   /* privilege level 0 stack pointer */
  ulong   ss0;                    /* privilege level 0 stack selector */
  ulong   esp1;                   /* privilege level 1 stack pointer */
  ulong   ss1;                    /* privilege level 1 stack selector */
  ulong   esp2;                   /* privilege level 2 stack pointer */
  ulong   ss2;                    /* privilege level 2 stack selector */
  ulong   cr3;                    /* page directory base register */
  ulong   eip;                    /* instruction pointer */
  ulong   eflags;                 /* flags register */
  ulong   eax;                    /* general registers */
  ulong   ecx;
  ulong   edx;
  ulong   ebx;
  ulong   esp;
  ulong   ebp;
  ulong   esi;
  ulong   edi;
  ulong   es;                     /* segment selectors */
  ulong   cs;
  ulong   ss;
  ulong   ds;
  ulong   fs;
  ulong   gs;
  ulong   ldt;                    /* selector for task's LDT */
  ulong   iomap;                  /* I/O map base address + T-bit */
} Tss;
```

```
137
138   struct Segdesc
139   {
140     ulong   d0;
141     ulong   d1;
142   };
143
144   struct Mach
145   {
146     int     machno;                 /* physical id of processor */
147     ulong   splpc;                  /* pc of last caller to splhi */
148
149     ulong*  pdb;                    /* page directory base for this processor (va) */
150     Tss*    tss;                    /* tss for this processor */
151     Segdesc gdt[6];                 /* gdt for this processor */
152
153     Proc*   proc;                   /* current process on this processor */
154     Proc*   externup;               /* extern register Proc *up */
155
156     Page*   pdbpool;
157     int     pdbcnt;
158
159     ulong   ticks;                  /* of the clock since boot time */
160     Label   sched;                  /* scheduler wakeup */
161     Lock    alarmlock;              /* access to alarm list */
162     void*   alarm;                  /* alarms bound to this clock */
163
164     ulong   fairness;               /* for runproc */
165
166     int     tlbfault;
167     int     tlbpurge;
168     int     pfault;
169     int     cs;
170     int     syscall;
171     int     load;
172     int     intr;
173     vlong   fastclock;              /* last sampled value */
174     vlong   intrts;                 /* time stamp of last interrupt */
175     int     flushmmu;               /* make current proc flush it's mmu state */
176
177     ulong   spuriousintr;
178     int     lastintr;
179
180     int     loopconst;
181
182     int     cpumhz;
183     int     cpuhz;
184     int     cpuidax;
185     int     cpuiddx;
186     char    cpuidid[16];
187     char*   cpuidtype;
188
189     vlong   mtrrcap;
190     vlong   mtrrdef;
191     vlong   mtrrfix[11];
192     vlong   mtrrvar[32];            /* 256 max. */
193
194     int     stack[1];
195   };
196
197   /*
198    * Fake kmap
199    */
200   typedef void              KMap;
201   #define   VA(k)           ((ulong)(k))
202   #define   kmap(p)         (KMap*)((p)->pa|KZERO)
203   #define   kunmap(k)
204
205   struct
206   {
207     Lock;
208     int     machs;                  /* bitmap of active CPUs */
209     int     exiting;                /* shutdown */
210     int     ispanic;                /* shutdown in response to a panic */
211   }active;
212
213   /*
214    *  routines for things outside the PC model, like power management
215    */
216   struct PCArch
217   {
218     char*   id;
219     int     (*ident)(void);         /* this should be in the model */
220     void    (*reset)(void);         /* this should be in the model */
221     int     (*serialpower)(int);    /* 1 == on, 0 == off */
222     int     (*modempower)(int);     /* 1 == on, 0 == off */
223
224     void    (*intrinit)(void);
225     int     (*intrenable)(Vctl*);
226
227     void    (*clockenable)(void);
228     uvlong  (*fastclock)(uvlong*);
229   };
230
231   /*
232    *  a parsed plan9.ini line
233    */
234   #define ISAOPTLEN 16
235   #define NISAOPT           8
236
237   struct ISAConf {
238     char    type[NAMELEN];
239     ulong   port;
240     ulong   irq;
241     ulong   dma;
242     ulong   mem;
243     ulong   size;
244     ulong   freq;
245
246     int     nopt;
247     char    opt[NISAOPT][ISAOPTLEN];
248   };
249
250   extern PCArch     *arch;                  /* PC architecture */
251
252   /*
253    * Each processor sees its own Mach structure at address MACHADDR.
254    * However, the Mach structures must also be available via the per-processor
255    * MMU information array machp, mainly for disambiguation and access to
256    * the clock which is only maintained by the bootstrap processor (0).
257    */
258   Mach* machp[MAXMACH];
259
260   #define   MACHP(n)        (machp[n])
261
262   extern Mach       *m;
263   #define up        (((Mach*)MACHADDR)->externup)
```

```
typedef struct List {
  void    *next;
} List;

typedef struct Alarm {
  List;
  int     busy;
  long    dt;
  void    (*f)(void*);
  void    *arg;
} Alarm;

typedef struct Block Block;
struct Block {
  Block*  next;
  uchar*  rp;                   /* first unconsumed byte */
  uchar*  wp;                   /* first empty byte */
  uchar*  lim;                  /* 1 past the end of the buffer */
  uchar*  base;                 /* start of the buffer */
  ulong   flag;
};
#define BLEN(s)   ((s)->wp - (s)->rp)

typedef struct IOQ IOQ;
typedef struct IOQ {
  uchar   buf[4096];
  uchar   *in;
  uchar   *out;
  int     state;
  int     (*getc)(IOQ*);
  int     (*putc)(IOQ*, int);
  void    *ptr;
};

enum {
  Eaddrlen        = 6,
  ETHERMINTU      = 60,         /* minimum transmit size */
  ETHERMAXTU      = 1514,       /* maximum transmit size */
  ETHERHDRSIZE    = 14,         /* size of an ethernet header */

  MaxEther        = 4,
};

typedef struct {
  uchar   d[Eaddrlen];
  uchar   s[Eaddrlen];
  uchar   type[2];
  uchar   data[1500];
  uchar   crc[4];
} Etherpkt;

extern uchar broadcast[Eaddrlen];

typedef struct Ureg Ureg;

typedef struct Segdesc {
  ulong   d0;
  ulong   d1;
} Segdesc;

typedef struct Mach {
  ulong   ticks;                /* of the clock since boot time */
  void    *alarm;               /* alarms bound to this clock */
} Mach;

extern Mach *m;

#define I_MAGIC           ((((4*11)+0)*11)+7)

typedef struct Exec Exec;
struct    Exec
{
  uchar   magic[4];             /* magic number */
  uchar   text[4];              /* size of text segment */
  uchar   data[4];              /* size of initialized data */
  uchar   bss[4];               /* size of uninitialized data */
  uchar   syms[4];              /* size of symbol table */
  uchar   entry[4];             /* entry point */
  uchar   spsz[4];              /* size of sp/pc offset table */
  uchar   pcsz[4];              /* size of pc/line number table */
};

/*
 *  a parsed .ini line
 */
#define ISAOPTLEN 16
#define NISAOPT           8

typedef struct  ISAConf {
  char    type[NAMELEN];
  ulong   port;
  ulong   irq;
  ulong   mem;
  ulong   size;
  uchar   ea[6];

  int     nopt;
  char    opt[NISAOPT][ISAOPTLEN];
} ISAConf;

typedef struct Pcidev Pcidev;
typedef struct PCMmap PCMmap;

#define BOOTLINE  ((char*)CONFADDR)

enum {
  MB =            (1024*1024),
};
#define ROUND(s, sz)      (((s)+((sz)-1))&~((sz)-1))


typedef struct Type Type;
typedef struct Medium Medium;
typedef struct Boot Boot;

enum {                                  /* type */
  Tnil            = 0x00,

  Tfloppy         = 0x01,
  Tsd             = 0x02,
  Tether          = 0x03,

  Tany            = -1,
};

enum {                                  /* name and flag */
  Fnone           = 0x00,

  Ndos            = 0x00,
  Fdos            = (1<<Ndos),
  Nboot           = 0x01,
  Fboot           = (1<<Nboot),
  Nbootp          = 0x02,
  Fbootp          = (1<<Nbootp),
  NName           = 3,
```

```
137     Fany            = Fbootp|Fboot|Fdos,
138
139     Fini            = 0x10,
140     Fprobe          = 0x80,
141   };
142
143   typedef struct Type {
144     int     type;
145     int     flag;
146     int     (*init)(void);
147     void    (*initdev)(int, char*);
148     void* (*getdospart)(int, char*);        /* actually returns Dos* */
149     void    (*addconf)(int);
150     int     (*boot)(int, char*, Boot*);
151
152     int     mask;
153     Medium* media;
154   } Type;
155
156   extern void (*etherdetach)(void);
157
158   typedef struct Lock {     /* for ilock, iunlock */
159     int locked;
160     int spl;
161   } Lock;
162
163   enum {    /* returned by bootpass */
164     MORE, ENOUGH, FAIL
165   };
166   enum {
167     INITKERNEL,
168     READEXEC,
169     READTEXT,
170     READDATA,
171     READGZIP,
172     TRYBOOT,
173     INIT9LOAD,
174     READ9LOAD,
175     FAILED
176   };
177
178   struct Boot {
179     int state;
180
181     Exec exec;
182     char *bp;       /* base ptr */
183     char *wp;       /* write ptr */
184     char *ep;       /* end ptr */
185   };
186
187   extern int        debug;
```

```
#include "u.h"
#include "../port/lib.h"
#include "mem.h"
#include "dat.h"
#include "fns.h"
#include "../port/error.h"

enum {
  Black,
  Blue,
  Green,
  Cyan,
  Red,
  Magenta,
  Brown,
  Grey,

  Bright = 0x08,
  Blinking = 0x80,

  Yellow = Bright|Brown,
  White = Bright|Grey,
};

enum {
  Width           = 80*2,
  Height          = 25,

  Attr            = (Black<<4)|Grey,      /* high nibble background
                                           * low foreground
                                           */
};

#define CGASCREENBASE     ((uchar*)KADDR(0xB8000))

static int cgapos;
static int screeninitdone;
static Lock cgascreenlock;

static uchar
cgaregr(int index)
{
  outb(0x3D4, index);
  return inb(0x3D4+1) & 0xFF;
}

static void
cgaregw(int index, int data)
{
  outb(0x3D4, index);
  outb(0x3D4+1, data);
}

static void
movecursor(void)
{
  cgaregw(0x0E, (cgapos/2>>8) & 0xFF);
  cgaregw(0x0F, cgapos/2 & 0xFF);
  CGASCREENBASE[cgapos+1] = Attr;
}

static void
cgascreenputc(int c)
{
  int i;
  uchar *p;

  if(c == '\n'){
          cgapos = cgapos/Width;
          cgapos = (cgapos+1)*Width;
  }
  else if(c == '\t'){
          i = 8 - ((cgapos/2)&7);
          while(i-->0)
                  cgascreenputc(' ');
  }
  else if(c == '\b'){
          if(cgapos >= 2)
                  cgapos -= 2;
          cgascreenputc(' ');
          cgapos -= 2;
  }
  else{
          CGASCREENBASE[cgapos++] = c;
          CGASCREENBASE[cgapos++] = Attr;
  }
  if(cgapos >= Width*Height){
          memmove(CGASCREENBASE, &CGASCREENBASE[Width], Width*(Height-1));
          p = &CGASCREENBASE[Width*(Height-1)];
          for(i=0; i<Width/2; i++){
                  *p++ = ' ';
                  *p++ = Attr;
          }
          cgapos = Width*(Height-1);
  }
  movecursor();
}

void
screeninit(void)
{
  cgapos = cgaregr(0x0E)<<8;
  cgapos |= cgaregr(0x0F);
  cgapos *= 2;
  screeninitdone = 1;
}

static void
cgascreenputs(char* s, int n)
{
  if(!islo()){
          /*
           * Don't deadlock trying to
           * print in an interrupt.
           */
          if(!canlock(&cgascreenlock))
                  return;
  }
  else
          lock(&cgascreenlock);

  while(n-- > 0)
          cgascreenputc(*s++);

  unlock(&cgascreenlock);
}

void (*screenputs)(char*, int) = cgascreenputs;
```

```
#include "u.h"
#include "../port/lib.h"
#include "mem.h"
#include "dat.h"
#include "fns.h"
#include "io.h"
#include "ureg.h"

void (*kproftimer)(ulong);

typedef struct Clock0link Clock0link;
typedef struct Clock0link {
  void            (*clock)(void);
  Clock0link*     link;
} Clock0link;

static Clock0link *clock0link;
static Lock clock0lock;

void
addclock0link(void (*clock)(void))
{
  Clock0link *lp;

  if((lp = malloc(sizeof(Clock0link))) == 0){
          print("addclock0link: too many links\n");
          return;
  }
  ilock(&clock0lock);
  lp->clock = clock;
  lp->link = clock0link;
  clock0link = lp;
  iunlock(&clock0lock);
}

void
clockintr(Ureg* ureg, void*)
{
  Clock0link *lp;

  // this needs to be here for synchronizing mp clocks
  // see mp.c's squidboy()
  fastticks(nil);
  m->ticks++;
  if(m->proc)
          m->proc->pc = ureg->pc;

  accounttime();
  if(kproftimer != nil)
          kproftimer(ureg->pc);
  if((active.machs & (1<<m->machno)) == 0)
          return;
  if(active.exiting && (active.machs & (1<<m->machno)))
          exit(0);

  checkalarms();
  if(m->machno == 0){
          ilock(&clock0lock);
          for(lp = clock0link; lp; lp = lp->link)
                  lp->clock();
          iunlock(&clock0lock);
  }

  if(m->flushmmu){
          if(up)
                  flushmmu();
          m->flushmmu = 0;
  }

  if(up == 0 || up->state != Running)
          return;

  if(anyready())
          sched();

  if((ureg->cs & 0xFFFF) == UESEL) {
          (*(ulong*)(USTKTOP-BY2WD)) += TK2MS(1);
          segclock(ureg->pc);
  }
}

void
delay(int millisecs)
{
  millisecs *= m->loopconst;
  if(millisecs <= 0)
          millisecs = 1;
  aamloop(millisecs);
}

void
microdelay(int microsecs)
{
  microsecs *= m->loopconst;
  microsecs /= 1000;
  if(microsecs <= 0)
          microsecs = 1;
  aamloop(microsecs);
}

ulong
TK2MS(ulong ticks)
{
  uvlong t, hz;

  t = ticks;
  hz = HZ;
  t *= 1000L;
  t = t/hz;
  ticks = t;
  return ticks;
}
```

```
typedef struct Conf       Conf;
typedef struct FPsave     FPsave;
typedef struct ISAConf    ISAConf;
typedef struct Label      Label;
typedef struct Lock       Lock;
typedef struct MMU        MMU;
typedef struct Mach       Mach;
typedef struct Notsave    Notsave;
typedef struct PCArch     PCArch;
typedef struct Pcidev     Pcidev;
typedef struct PCMmap     PCMmap;
typedef struct Page       Page;
typedef struct PMMU       PMMU;
typedef struct Proc       Proc;
typedef struct Segdesc    Segdesc;
typedef struct Ureg       Ureg;
typedef struct Vctl       Vctl;

/*
 *  parameters for sysproc.c
 */
#define AOUT_MAGIC        (I_MAGIC)

struct Lock
{
  ulong   key;
  ulong   sr;
  ulong   pc;
  Proc    *p;
  ushort  isilock;
};

struct Label
{
  ulong   sp;
  ulong   pc;
};


/*
 * FPsave.status
 */
enum
{
  FPinit,
  FPactive,
  FPinactive,
};

struct    FPsave
{
  ushort  control;
  ushort  r1;
  ushort  status;
  ushort  r2;
  ushort  tag;
  ushort  r3;
  ulong   pc;
  ushort  selector;
  ushort  r4;
  ulong   operand;
  ushort  oselector;
  ushort  r5;
  uchar   regs[80];       /* floating point registers */
};

struct Conf
{
  ulong   nmach;          /* processors */
  ulong   nproc;          /* processes */
  ulong   monitor;        /* has monitor? */
  ulong   npage0;         /* total physical pages of memory */
  ulong   npage1;         /* total physical pages of memory */
  ulong   npage;          /* total physical pages of memory */
  ulong   upages;         /* user page pool */
  ulong   nimage;         /* number of page cache image headers */
  ulong   nswap;          /* number of swap pages */
  int     nswppo;         /* max # of pageouts per segment pass */
  ulong   base0;          /* base of bank 0 */
  ulong   base1;          /* base of bank 1 */
  ulong   copymode;       /* 0 is copy on write, 1 is copy on reference */
  ulong   ialloc;         /* max interrupt time allocation in bytes */
  ulong   pipeqsize;      /* size in bytes of pipe queues */
};

/*
 *  MMU stuff in proc
 */
#define NCOLOR 1
struct PMMU
{
  Page*   mmupdb;                 /* page directory base */
  Page*   mmufree;                /* unused page table pages */
  Page*   mmuused;                /* used page table pages */
};

/*
 *  things saved in the Proc structure during a notify
 */
struct Notsave
{
  ulong   svflags;
  ulong   svcs;
  ulong   svss;
};

#include "../port/portdat.h"

typedef struct {
  ulong   link;                   /* link (old TSS selector) */
  ulong   esp0;                   /* privilege level 0 stack pointer */
  ulong   ss0;                    /* privilege level 0 stack selector */
  ulong   esp1;                   /* privilege level 1 stack pointer */
  ulong   ss1;                    /* privilege level 1 stack selector */
  ulong   esp2;                   /* privilege level 2 stack pointer */
  ulong   ss2;                    /* privilege level 2 stack selector */
  ulong   cr3;                    /* page directory base register */
  ulong   eip;                    /* instruction pointer */
  ulong   eflags;                 /* flags register */
  ulong   eax;                    /* general registers */
  ulong   ecx;
  ulong   edx;
  ulong   ebx;
  ulong   esp;
  ulong   ebp;
  ulong   esi;
  ulong   edi;
  ulong   es;                     /* segment selectors */
  ulong   cs;
  ulong   ss;
  ulong   ds;
  ulong   fs;
  ulong   gs;
  ulong   ldt;                    /* selector for task's LDT */
  ulong   iomap;                  /* I/O map base address + T-bit */
} Tss;
```

```
137
138   struct Segdesc
139   {
140     ulong   d0;
141     ulong   d1;
142   };
143
144   struct Mach
145   {
146     int     machno;                 /* physical id of processor */
147     ulong   splpc;                  /* pc of last caller to splhi */
148
149     ulong*  pdb;                    /* page directory base for this processor (va) */
150     Tss*    tss;                    /* tss for this processor */
151     Segdesc gdt[6];                 /* gdt for this processor */
152
153     Proc*   proc;                   /* current process on this processor */
154     Proc*   externup;               /* extern register Proc *up */
155
156     Page*   pdbpool;
157     int     pdbcnt;
158
159     ulong   ticks;                  /* of the clock since boot time */
160     Label   sched;                  /* scheduler wakeup */
161     Lock    alarmlock;              /* access to alarm list */
162     void*   alarm;                  /* alarms bound to this clock */
163
164     ulong   fairness;               /* for runproc */
165
166     int     tlbfault;
167     int     tlbpurge;
168     int     pfault;
169     int     cs;
170     int     syscall;
171     int     load;
172     int     intr;
173     vlong   fastclock;              /* last sampled value */
174     vlong   intrts;                 /* time stamp of last interrupt */
175     int     flushmmu;               /* make current proc flush it's mmu state */
176
177     ulong   spuriousintr;
178     int     lastintr;
179
180     int     loopconst;
181
182     int     cpumhz;
183     int     cpuhz;
184     int     cpuidax;
185     int     cpuiddx;
186     char    cpuidid[16];
187     char*   cpuidtype;
188
189     vlong   mtrrcap;
190     vlong   mtrrdef;
191     vlong   mtrrfix[11];
192     vlong   mtrrvar[32];            /* 256 max. */
193
194     int     stack[1];
195   };
196
197   /*
198    * Fake kmap
199    */
200   typedef void              KMap;
201   #define   VA(k)           ((ulong)(k))
202   #define   kmap(p)         (KMap*)((p)->pa|KZERO)
203   #define   kunmap(k)
204
205   struct
206   {
207     Lock;
208     int     machs;                  /* bitmap of active CPUs */
209     int     exiting;                /* shutdown */
210     int     ispanic;                /* shutdown in response to a panic */
211   }active;
212
213   /*
214    *  routines for things outside the PC model, like power management
215    */
216   struct PCArch
217   {
218     char*   id;
219     int     (*ident)(void);         /* this should be in the model */
220     void    (*reset)(void);         /* this should be in the model */
221     int     (*serialpower)(int);    /* 1 == on, 0 == off */
222     int     (*modempower)(int);     /* 1 == on, 0 == off */
223
224     void    (*intrinit)(void);
225     int     (*intrenable)(Vctl*);
226
227     void    (*clockenable)(void);
228     uvlong  (*fastclock)(uvlong*);
229   };
230
231   /*
232    *  a parsed plan9.ini line
233    */
234   #define ISAOPTLEN 16
235   #define NISAOPT           8
236
237   struct ISAConf {
238     char    type[NAMELEN];
239     ulong   port;
240     ulong   irq;
241     ulong   dma;
242     ulong   mem;
243     ulong   size;
244     ulong   freq;
245
246     int     nopt;
247     char    opt[NISAOPT][ISAOPTLEN];
248   };
249
250   extern PCArch     *arch;                  /* PC architecture */
251
252   /*
253    * Each processor sees its own Mach structure at address MACHADDR.
254    * However, the Mach structures must also be available via the per-processor
255    * MMU information array machp, mainly for disambiguation and access to
256    * the clock which is only maintained by the bootstrap processor (0).
257    */
258   Mach* machp[MAXMACH];
259
260   #define   MACHP(n)        (machp[n])
261
262   extern Mach       *m;
263   #define up        (((Mach*)MACHADDR)->externup)
```

```
#include "u.h"
#include "../port/lib.h"
#include "mem.h"
#include "dat.h"
#include "fns.h"
#include "io.h"
#include "ureg.h"
#include "../port/error.h"


typedef struct IOMap IOMap;
struct IOMap
{
  IOMap   *next;
  char    tag[13];
  ulong   start;
  ulong   end;
};

static struct
{
  Lock;
  IOMap   *m;
  IOMap   *free;
  IOMap   maps[32];               // some initial free maps

  QLock   ql;                     // lock for reading map
} iomap;

enum {
  Qdir,
  Qcputype,
  Qioalloc,
  Qiob,
  Qiow,
  Qiol,
  Qirqalloc,
};

static Dirtab ioallocdir[] = {
  "cputype",      { Qcputype, 0 },        0,      0444,
  "ioalloc",      { Qioalloc, 0 },        0,      0444,
  "iob",          { Qiob, 0 },            0,      0660,
  "iow",          { Qiow, 0 },            0,      0660,
  "iol",          { Qiol, 0 },            0,      0660,
  "irqalloc",     { Qirqalloc, 0},        0,      0444,
};

void
ioinit(void)
{
  int i;

  for(i = 0; i < nelem(iomap.maps)-1; i++)
          iomap.maps[i].next = &iomap.maps[i+1];
  iomap.maps[i].next = nil;
  iomap.free = iomap.maps;

  // a dummy entry at 2^16
  ioalloc(0x10000, 1, 0, "dummy");
}

static long cputyperead(char*, int, ulong);

//
//        alloc some io port space and remember who it was
//        alloced to.  if port < 0, find a free region.
//
int
ioalloc(int port, int size, int align, char *tag)
{
  IOMap *m, **l;
  int i;

  lock(&iomap);
  if(port < 0){
          // find a free port above 0x400 and below 0x1000
          port = 0x400;
          for(l = &iomap.m; *l; l = &(*l)->next){
                  m = *l;
                  i = m->start - port;
                  if(i > size)
                          break;
                  if(align > 0)
                          port = ((port+align-1)/align)*align;
                  else
                          port = m->end;
          }
          if(*l == nil){
                  unlock(&iomap);
                  return -1;
          }
  } else {
          // see if the space clashes with previously allocated ports
          for(l = &iomap.m; *l; l = &(*l)->next){
                  m = *l;
                  if(m->end <= port)
                          continue;
                  if(m->start >= port+size)
                          break;
                  unlock(&iomap);
                  return -1;
          }
  }
  m = iomap.free;
  if(m == nil){
          print("ioalloc: out of maps");
          unlock(&iomap);
          return port;
  }
  iomap.free = m->next;
  m->next = *l;
  m->start = port;
  m->end = port + size;
  strncpy(m->tag, tag, sizeof(m->tag));
  m->tag[sizeof(m->tag)-1] = 0;
  *l = m;

  ioallocdir[0].qid.vers++;

  unlock(&iomap);
  return m->start;
}

void
iofree(int port)
{
  IOMap *m, **l;

  lock(&iomap);
  for(l = &iomap.m; *l; l = &(*l)->next){
          if((*l)->start == port){
                  m = *l;
                  *l = m->next;
                  m->next = iomap.free;
                  iomap.free = m;
```

```
137                     break;
138             }
139             if((*l)->start > port)
140                     break;
141     }
142     ioallocdir[0].qid.vers++;
143     unlock(&iomap);
144   }
145
146   int
147   iounused(int start, int end)
148   {
149     IOMap *m;
150
151     for(m = iomap.m; m; m = m->next){
152             if(start >= m->start && start < m->end
153             || start <= m->start && end > m->start)
154                     return 0;
155     }
156     return 1;
157   }
158
159   static void
160   checkport(int start, int end)
161   {
162     /* standard vga regs are OK */
163     if(start >= 0x2b0 && end <= 0x2df+1)
164             return;
165     if(start >= 0x3c0 && end <= 0x3da+1)
166             return;
167
168     if(iounused(start, end))
169             return;
170     error(Eperm);
171   }
172
173   static Chan*
174   archattach(char* spec)
175   {
176     return devattach('P', spec);
177   }
178
179   int
180   archwalk(Chan* c, char* name)
181   {
182     return devwalk(c, name, ioallocdir, nelem(ioallocdir), devgen);
183   }
184
185   static void
186   archstat(Chan* c, char* dp)
187   {
188     devstat(c, dp, ioallocdir, nelem(ioallocdir), devgen);
189   }
190
191   static Chan*
192   archopen(Chan* c, int omode)
193   {
194     return devopen(c, omode, ioallocdir, nelem(ioallocdir), devgen);
195   }
196
197   static void
198   archclose(Chan*)
199   {
200   }
201
202   enum
203   {
204     Linelen= 31,
205   };
206
207   static long
208   archread(Chan *c, void *a, long n, vlong offset)
209   {
210     char *p;
211     IOMap *m;
212     char buf[Linelen+1];
213     int port;
214     ushort *sp;
215     ulong *lp;
216
217     switch(c->qid.path & ~CHDIR){
218
219     case Qdir:
220             return devdirread(c, a, n, ioallocdir, nelem(ioallocdir), devgen);
221
222     case Qiob:
223             port = offset;
224             checkport(offset, offset+n);
225             for(p = a; port < offset+n; port++)
226                     *p++ = inb(port);
227             return n;
228
229     case Qiow:
230             if((n & 0x01) || (offset & 0x01))
231                     error(Ebadarg);
232             checkport(offset, offset+n+1);
233             n /= 2;
234             sp = a;
235             for(port = offset; port < offset+n; port += 2)
236                     *sp++ = ins(port);
237             return n*2;
238
239     case Qiol:
240             if((n & 0x03) || (offset & 0x03))
241                     error(Ebadarg);
242             checkport(offset, offset+n+3);
243             n /= 4;
244             lp = a;
245             for(port = offset; port < offset+n; port += 4)
246                     *lp++ = inl(port);
247             return n*4;
248
249     case Qioalloc:
250             break;
251
252     case Qirqalloc:
253             return irqallocread(a, n, offset);
254
255     case Qcputype:
256             return cputyperead(a, n, offset);
257
258     default:
259             error(Eperm);
260             break;
261     }
262
263     offset = offset/Linelen;
264     n = n/Linelen;
265     p = a;
266     lock(&iomap);
267     for(m = iomap.m; n > 0 && m != nil; m = m->next){
268             if(offset-- > 0)
269                     continue;
270             if(strcmp(m->tag, "dummy") == 0)
271                     break;
272             sprint(buf, "%8lux %8lux %-12.12s\n", m->start, m->end-1, m->tag);
```

```
273             memmove(p, buf, Linelen);
274             p += Linelen;
275             n--;
276     }
277     unlock(&iomap);
278
279     return p - (char*)a;
280   }
281
282   static long
283   archwrite(Chan *c, void *a, long n, vlong offset)
284   {
285     int port;
286     ushort *sp;
287     ulong *lp;
288     char *p;
289
290     switch(c->qid.path & ~CHDIR){
291
292     case Qiob:
293             p = a;
294             checkport(offset, offset+n);
295             for(port = offset; port < offset+n; port++)
296                     outb(port, *p++);
297             return n;
298
299     case Qiow:
300             if((n & 01) || (offset & 01))
301                     error(Ebadarg);
302             checkport(offset, offset+n+1);
303             n /= 2;
304             sp = a;
305             for(port = offset; port < offset+n; port += 2)
306                     outs(port, *sp++);
307             return n*2;
308
309     case Qiol:
310             if((n & 0x03) || (offset & 0x03))
311                     error(Ebadarg);
312             checkport(offset, offset+n+3);
313             n /= 4;
314             lp = a;
315             for(port = offset; port < offset+n; port += 4)
316                     outl(port, *lp++);
317             return n*4;
318
319     default:
320             error(Eperm);
321             break;
322     }
323     return 0;
324   }
325
326   Dev archdevtab = {
327     'P',
328     "arch",
329
330     devreset,
331     devinit,
332     archattach,
333     devclone,
334     archwalk,
335     archstat,
336     archopen,
337     devcreate,
338     archclose,
339     archread,
340     devbread,
341     archwrite,
342     devbwrite,
343     devremove,
344     devwstat,
345   };
346
347   /*
348    *  the following is a generic version of the
349    *  architecture specific stuff
350    */
351
352   static int
353   unimplemented(int)
354   {
355     return 0;
356   }
357
358   static void
359   nop(void)
360   {
361   }
362
363   void (*coherence)(void) = nop;
364   void cycletimerinit(void);
365   uvlong cycletimer(uvlong*);
366
367   PCArch* arch;
368   extern PCArch* knownarch[];
369
370   PCArch archgeneric = {
371     "generic",                          /* id */
372     0,                                  /* ident */
373     i8042reset,                         /* reset */
374     unimplemented,                      /* serialpower */
375     unimplemented,                      /* modempower */
376
377     i8259init,                          /* intrinit */
378     i8259enable,                        /* intrenable */
379
380     i8253enable,                        /* clockenable */
381
382     i8253read,                          /* read the standard timer */
383   };
384
385   typedef struct {
386     int     family;
387     int     model;
388     int     aalcycles;
389     char*   name;
390   } X86type;
391
392   static X86type x86intel[] =
393   {
394     { 4,    0,      22,     "486DX", },     /* known chips */
395     { 4,    1,      22,     "486DX50", },
396     { 4,    2,      22,     "486SX", },
397     { 4,    3,      22,     "486DX2", },
398     { 4,    4,      22,     "486SL", },
399     { 4,    5,      22,     "486SX2", },
400     { 4,    7,      22,     "DX2WB", },     /* P24D */
401     { 4,    8,      22,     "DX4", },       /* P24C */
402     { 4,    9,      22,     "DX4WB", },     /* P24CT */
403     { 5,    0,      23,     "P5", },
404     { 5,    1,      23,     "P5", },
405     { 5,    2,      23,     "P54C", },
406     { 5,    3,      23,     "P24T", },
407     { 5,    4,      23,     "P55C MMX", },
408     { 5,    7,      23,     "P54C VRT", },
```

```
409     { 6,    1,      16,     "PentiumPro", },/* trial and error */
410     { 6,    3,      16,     "PentiumII", },
411     { 6,    5,      16,     "PentiumII/Xeon", },
412     { 6,    6,      16,     "Celeron", },
413     { 6,    7,      16,     "PentiumIII/Xeon", },
414     { 6,    8,      16,     "PentiumIII/Xeon", },
415
416     { 3,    -1,     32,     "386", },       /* family defaults */
417     { 4,    -1,     22,     "486", },
418     { 5,    -1,     23,     "P5", },
419     { 6,    -1,     16,     "P6", },
420
421     { -1,   -1,     23,     "unknown", },   /* total default */
422   };
423
424   /*
425    * The AMD processors all implement the CPUID instruction.
426    * The later ones also return the processor name via functions
427    * 0x80000002, 0x80000003 and 0x80000004 in registers AX, BX, CX
428    * and DX:
429    *        K5      "AMD-K5(tm) Processor"
430    *        K6      "AMD-K6tm w/ multimedia extensions"
431    *        K6 3D   "AMD-K6(tm) 3D processor"
432    *        K6 3D+  ?
433    */
434   static X86type x86amd[] =
435   {
436     { 5,    0,      23,     "AMD-K5", },    /* guesswork */
437     { 5,    1,      23,     "AMD-K5", },    /* guesswork */
438     { 5,    2,      23,     "AMD-K5", },    /* guesswork */
439     { 5,    3,      23,     "AMD-K5", },    /* guesswork */
440     { 5,    6,      11,     "AMD-K6", },    /* trial and error */
441     { 5,    7,      11,     "AMD-K6", },    /* trial and error */
442     { 5,    8,      11,     "AMD-K6-2", },  /* trial and error */
443     { 5,    9,      11,     "AMD-K6-III", },/* trial and error */
444
445     { 6,    1,      11,     "AMD-Athlon", },/* trial and error */
446     { 6,    2,      11,     "AMD-Athlon", },/* trial and error */
447
448     { 4,    -1,     22,     "Am486", },     /* guesswork */
449     { 5,    -1,     23,     "AMD-K5/K6", }, /* guesswork */
450     { 6,    -1,     11,     "AMD-Athlon", },/* guesswork */
451
452     { -1,   -1,     23,     "unknown", },   /* total default */
453   };
454
455   static uvlong fasthz;
456   static X86type *cputype;
457
458   void
459   cpuidprint(void)
460   {
461     int i;
462     char buf[128];
463
464     i = sprint(buf, "cpu%d: %dMHz ", m->machno, m->cpumhz);
465     if(m->cpuidid[0])
466             i += sprint(buf+i, "%s ", m->cpuidid);
467     sprint(buf+i, "%s (cpuid: AX 0x%4.4uX DX 0x%4.4uX)\n",
468             m->cpuidtype, m->cpuidax, m->cpuiddx);
469     print(buf);
470   }
471
472   int
473   cpuidentify(void)
474   {
475     int family, model;
476     X86type *t;
477     ulong cr4;
478     vlong mct;
479
480     cpuid(m->cpuidid, &m->cpuidax, &m->cpuiddx);
481     if(strncmp(m->cpuidid, "AuthenticAMD", 12) == 0)
482             t = x86amd;
483     else
484             t = x86intel;
485     family = X86FAMILY(m->cpuidax);
486     model = X86MODEL(m->cpuidax);
487     while(t->name){
488             if((t->family == family && t->model == model)
489             || (t->family == family && t->model == -1)
490             || (t->family == -1))
491                     break;
492             t++;
493     }
494     m->cpuidtype = t->name;
495     i8253init(t->aalcycles, t->family >= 5);
496
497     /*
498      * If machine check exception or page size extensions are supported
499      * enable them in CR4 and clear any other set extensions.
500      * If machine check was enabled clear out any lingering status.
501      */
502     if(m->cpuiddx & 0x88){
503             cr4 = 0;
504             if(m->cpuiddx & 0x08)
505                     cr4 |= 0x10;            /* page size extensions */
506             if(m->cpuiddx & 0x80)
507                     cr4 |= 0x40;            /* machine check enable */
508             putcr4(cr4);
509             if(m->cpuiddx & 0x80)
510                     rdmsr(0x01, &mct);
511     }
512
513     cputype = t;
514     return t->family;
515   }
516
517   static long
518   cputyperead(char *a, int n, ulong offset)
519   {
520     char str[32];
521     ulong mhz;
522
523     mhz = (m->cpuhz+999999)/1000000;
524
525     snprint(str, sizeof(str), "%s %lud\n", cputype->name, mhz);
526     return readstr(offset, a, n, str);
527   }
528
529   void
530   archinit(void)
531   {
532     PCArch **p;
533
534     arch = 0;
535     for(p = knownarch; *p; p++){
536             if((*p)->ident && (*p)->ident() == 0){
537                     arch = *p;
538                     break;
539             }
540     }
541     if(arch == 0)
542             arch = &archgeneric;
543     else{
544             if(arch->id == 0)
```

```
545                     arch->id = archgeneric.id;
546             if(arch->reset == 0)
547                     arch->reset = archgeneric.reset;
548             if(arch->serialpower == 0)
549                     arch->serialpower = archgeneric.serialpower;
550             if(arch->modempower == 0)
551                     arch->modempower = archgeneric.modempower;
552
553             if(arch->intrinit == 0)
554                     arch->intrinit = archgeneric.intrinit;
555             if(arch->intrenable == 0)
556                     arch->intrenable = archgeneric.intrenable;
557     }
558
559     /* pick the better timer */
560     if(X86FAMILY(m->cpuidax) >= 5){
561             cycletimerinit();
562             arch->fastclock = cycletimer;
563     }
564
565     /*
566      * Decide whether to use copy-on-reference (386 and mp).
567      */
568     if(X86FAMILY(m->cpuidax) == 3 || conf.nmach > 1)
569             conf.copymode = 1;
570
571     if(X86FAMILY(m->cpuidax) >= 5)
572             coherence = wbflush;
573   }
574
575   void
576   cycletimerinit(void)
577   {
578     wrmsr(0x10, 0);
579     fasthz = m->cpuhz;
580   }
581
582   /*
583    *  return the most precise clock we have
584    */
585   uvlong
586   cycletimer(uvlong *hz)
587   {
588     uvlong tsc;
589
590     rdmsr(0x10, (vlong*)&tsc);
591     m->fastclock = tsc;
592     if(hz != nil)
593             *hz = fasthz;
594     return tsc;
595   }
596
597   vlong
598   fastticks(uvlong *hz)
599   {
600     return (*arch->fastclock)(hz);
601   }
```

```
#include  "u.h"
#include  "../port/lib.h"
#include  "mem.h"
#include  "dat.h"
#include  "fns.h"
#include  "../port/error.h"

void      (*consdebug)(void) = nil;

Queue*    kbdq;                   /* unprocessed console input */
Queue*    lineq;                  /* processed console input */
Queue*    printq;                 /* console output */

static struct
{
  QLock;

  int     raw;            /* true if we shouldn't process input */
  int     ctl;            /* number of opens to the control file */
  int     x;              /* index into line */
  char    line[1024];     /* current input line */

  Rune    c;
  int     count;
  int     repeat;
  int     ctlpoff;
} kbd;

char      sysname[NAMELEN];
vlong     fasthz;

static ulong      randomread(void*, ulong);
static void       randominit(void);
static void       seedrand(void);
static int        readtime(ulong, char*, int);
static int        readbintime(char*, int);
static int        writetime(char*, int);
static int        writebintime(char*, int);

void
printinit(void)
{
  lineq = qopen(2*1024, 0, 0, 0);
  if(lineq == nil)
          panic("printinit");
  qnoblock(lineq, 1);
}

int
consactive(void)
{
  if(printq)
          return qlen(printq) > 0;
  return 0;
}

void
prflush(void)
{
  while(consactive())
          ;
}

/*
 *   Print a string on the console.  Convert \n to \r\n for serial
 *   line consoles.  Locking of the queues is left up to the screen
 *   or uart code.  Multi-line messages to serial consoles may get
 *   interspersed with other messages.
 */
static void
putstrn0(char *str, int n, int usewrite)
{
  int m;
  char *t;
  char buf[PRINTSIZE+2];

  /*
   *  if there's an attached bit mapped display,
   *  put the message there.  screenputs is defined
   *  as a null macro for systems that have no such
   *  display.
   */
  screenputs(str, n);

  /*
   *  if there's a serial line being used as a console,
   *  put the message there.
   */
  if(printq == 0)
          return;

  while(n > 0) {
          t = memchr(str, '\n', n);
          if(t) {
                  m = t - str;
                  memmove(buf, str, m);
                  buf[m] = '\r';
                  buf[m+1] = '\n';
                  if(usewrite)
                          qwrite(printq, buf, m+2);
                  else
                          qiwrite(printq, buf, m+2);
                  str = t + 1;
                  n -= m + 1;
          } else {
                  if(usewrite)
                          qwrite(printq, str, n);
                  else
                          qiwrite(printq, str, n);
                  break;
          }
  }
}

void
putstrn(char *str, int n)
{
  putstrn0(str, n, 0);
}

int
snprint(char *s, int n, char *fmt, ...)
{
  va_list arg;

  va_start(arg, fmt);
  n = doprint(s, s+n, fmt, arg) - s;
  va_end(arg);
  return n;
}

int
sprint(char *s, char *fmt, ...)
{
  int n;
  va_list arg;
```

```
137
138     va_start(arg, fmt);
139     n = doprint(s, s+PRINTSIZE, fmt, arg) - s;
140     va_end(arg);
141     return n;
142   }
143
144   int noprint;
145
146   int
147   print(char *fmt, ...)
148   {
149     int n;
150     va_list arg;
151     char buf[PRINTSIZE];
152
153     if(noprint)
154             return -1;
155
156     va_start(arg, fmt);
157     n = doprint(buf, buf+sizeof(buf), fmt, arg) - buf;
158     va_end(arg);
159     putstrn(buf, n);
160
161     return n;
162   }
163
164   int
165   iprint(char *fmt, ...)
166   {
167     int n, s;
168     va_list arg;
169     char buf[PRINTSIZE];
170
171     s = splhi();
172     va_start(arg, fmt);
173     n = doprint(buf, buf+sizeof(buf), fmt, arg) - buf;
174     va_end(arg);
175     serialputs(buf, n);
176   //        screenputs(buf, n);
177     splx(s);
178
179     return n;
180   }
181
182   void
183   panic(char *fmt, ...)
184   {
185     int n;
186     va_list arg;
187     char buf[PRINTSIZE];
188
189     splhi();
190     strcpy(buf, "panic: ");
191     va_start(arg, fmt);
192     n = doprint(buf+strlen(buf), buf+sizeof(buf), fmt, arg) - buf;
193     va_end(arg);
194     buf[n] = '\n';
195     serialputs(buf, n+1);
196     if(consdebug)
197             consdebug();
198     putstrn(buf, n+1);
199     spllo();
200     prflush();
201     dumpstack();
202
203     exit(1);
204   }
205
206   void
207   _assert(char *fmt)
208   {
209     panic("assert failed: %s", fmt);
210   }
211
212   int
213   pprint(char *fmt, ...)
214   {
215     int n;
216     Chan *c;
217     va_list arg;
218     char buf[2*PRINTSIZE];
219
220     if(up == nil || up->fgrp == nil)
221             return 0;
222
223     c = up->fgrp->fd[2];
224     if(c==0 || (c->mode!=OWRITE && c->mode!=ORDWR))
225             return 0;
226     n = sprint(buf, "%s %lud: ", up->text, up->pid);
227     va_start(arg, fmt);
228     n = doprint(buf+n, buf+sizeof(buf), fmt, arg) - buf;
229     va_end(arg);
230
231     if(waserror())
232             return 0;
233     devtab[c->type]->write(c, buf, n, c->offset);
234     poperror();
235
236     lock(c);
237     c->offset += n;
238     unlock(c);
239
240     return n;
241   }
242
243   void
244   echo(Rune r, char *buf, int n)
245   {
246     static int ctrlt, pid;
247     extern ulong etext;
248     int x;
249
250     /*
251      * ^p hack
252      */
253     if(r==0x10 && cpuserver && !kbd.ctlpoff){
254             lock(&active);
255             active.exiting = 1;
256             unlock(&active);
257     }
258
259     /*
260      * ^t hack BUG
261      */
262     if(ctrlt == 2){
263             ctrlt = 0;
264             switch(r){
265             case 0x14:
266                     break;  /* pass it on */
267             case 's':
268                     dumpstack();
269                     break;
270             case 'x':
271                     xsummary();
272                     ixsummary();
```

```
273                     mallocsummary();
274                     pagersummary();
275                     break;
276             case 'd':
277                     if(consdebug == nil)
278                             consdebug = rdb;
279                     else
280                             consdebug = nil;
281                     print("consdebug now 0x%p\n", consdebug);
282                     return;
283             case 'D':
284                     if(consdebug == nil)
285                             consdebug = rdb;
286                     consdebug();
287                     return;
288             case 'p':
289                     x = spllo();
290                     procdump();
291                     splx(x);
292                     return;
293             case 'q':
294                     scheddump();
295                     break;
296             case 'k':
297                     if(!cpuserver)
298                             killbig();
299                     break;
300             case 'r':
301                     exit(0);
302                     break;
303             }
304     }
305     else if(r == 0x14){
306             ctrlt++;
307             return;
308     }
309     ctrlt = 0;
310     if(kbd.raw)
311             return;
312
313     /*
314      *  finally, the actual echoing
315      */
316     if(r == '\n'){
317             if(printq)
318                     qiwrite(printq, "\r", 1);
319     } else if(r == 0x15){
320             buf = "^U\n";
321             n = 3;
322     }
323     screenputs(buf, n);
324     if(printq)
325             qiwrite(printq, buf, n);
326   }
327
328   /*
329    *  Called by a uart interrupt for console input.
330    *
331    *  turn '\r' into '\n' before putting it into the queue.
332    */
333   int
334   kbdcr2nl(Queue *q, int ch)
335   {
336     if(ch == '\r')
337             ch = '\n';
338     return kbdputc(q, ch);
339   }
340
```

```
341   /*
342    *  Put character, possibly a rune, into read queue at interrupt time.
343    *  Called at interrupt time to process a character.
344    */
345   int
346   kbdputc(Queue*, int ch)
347   {
348     int n;
349     char buf[3];
350     Rune r;
351
352     r = ch;
353     n = runetochar(buf, &r);
354     if(n == 0)
355             return 0;
356     echo(r, buf, n);
357     kbd.c = r;
358     qproduce(kbdq, buf, n);
359     return 0;
360   }
361
362   void
363   kbdrepeat(int rep)
364   {
365     kbd.repeat = rep;
366     kbd.count = 0;
367   }
368
369   void
370   kbdclock(void)
371   {
372     if(kbd.repeat == 0)
373             return;
374     if(kbd.repeat==1 && ++kbd.count>HZ){
375             kbd.repeat = 2;
376             kbd.count = 0;
377             return;
378     }
379     if(++kbd.count&1)
380             kbdputc(kbdq, kbd.c);
381   }
382
383   enum{
384     Qdir,
385     Qauth,
386     Qauthcheck,
387     Qauthent,
388     Qbintime,
389     Qcons,
390     Qconsctl,
391     Qcputime,
392     Qdrivers,
393     Qkey,
394     Qhostdomain,
395     Qhostowner,
396     Qnull,
397     Qpgrpid,
398     Qpid,
399     Qppid,
400     Qrandom,
401     Qreboot,
402     Qswap,
403     Qsysname,
404     Qsysstat,
405     Qtime,
406     Quser,
407     Qzero,
408   };
```

```
409
410  enum
411  {
412    VLNUMSIZE=      22,
413  };
414
415  static Dirtab consdir[]={
416    "authenticate", {Qauth},        0,              0666,
417    "authcheck",    {Qauthcheck},   0,              0666,
418    "authenticator", {Qauthent},    0,              0666,
419    "bintime",      {Qbintime},     24,             0664,
420    "cons",         {Qcons},        0,              0660,
421    "consctl",      {Qconsctl},     0,              0220,
422    "cputime",      {Qcputime},     6*NUMSIZE,      0444,
423    "drivers",      {Qdrivers},     0,              0644,
424    "hostdomain",   {Qhostdomain},  DOMLEN,         0664,
425    "hostowner",    {Qhostowner},   NAMELEN,        0664,
426    "key",          {Qkey},         DESKEYLEN,      0622,
427    "null",         {Qnull},        0,              0666,
428    "pgrpid",       {Qpgrpid},      NUMSIZE,        0444,
429    "pid",          {Qpid},         NUMSIZE,        0444,
430    "ppid",         {Qppid},        NUMSIZE,        0444,
431    "random",       {Qrandom},      0,              0664,
432    "reboot",       {Qreboot},      0,              0664,
433    "swap",         {Qswap},        0,              0664,
434    "sysname",      {Qsysname},     0,              0664,
435    "sysstat",      {Qsysstat},     0,              0666,
436    "time",         {Qtime},        NUMSIZE+3*VLNUMSIZE,    0664,
437    "user",         {Quser},        NAMELEN,        0666,
438    "zero",         {Qzero},        0,              0444,
439  };
440
441  int
442  readnum(ulong off, char *buf, ulong n, ulong val, int size)
443  {
444    char tmp[64];
445
446    snprint(tmp, sizeof(tmp), "%*.0lud", size-1, val);
447    tmp[size-1] = ' ';
448    if(off >= size)
449            return 0;
450    if(off+n > size)
451            n = size-off;
452    memmove(buf, tmp+off, n);
453    return n;
454  }
455
456  int
457  readstr(ulong off, char *buf, ulong n, char *str)
458  {
459    int size;
460
461    size = strlen(str);
462    if(off >= size)
463            return 0;
464    if(off+n > size)
465            n = size-off;
466    memmove(buf, str+off, n);
467    return n;
468  }
469
470  static void
471  consinit(void)
472  {
473    todinit();
474    randominit();
475  }
476
477  static Chan*
478  consattach(char *spec)
479  {
480    return devattach('c', spec);
481  }
482
483  static int
484  conswalk(Chan *c, char *name)
485  {
486    return devwalk(c, name, consdir, nelem(consdir), devgen);
487  }
488
489  static void
490  consstat(Chan *c, char *dp)
491  {
492    devstat(c, dp, consdir, nelem(consdir), devgen);
493  }
494
495  static Chan*
496  consopen(Chan *c, int omode)
497  {
498    c->aux = 0;
499    switch(c->qid.path){
500    case Qconsctl:
501            if(!iseve())
502                    error(Eperm);
503            qlock(&kbd);
504            kbd.ctl++;
505            qunlock(&kbd);
506            break;
507    }
508    return devopen(c, omode, consdir, nelem(consdir), devgen);
509  }
510
511  static void
512  consclose(Chan *c)
513  {
514    /* last close of control file turns off raw */
515    switch(c->qid.path){
516    case Qconsctl:
517            if(c->flag&COPEN){
518                    qlock(&kbd);
519                    if(--kbd.ctl == 0)
520                            kbd.raw = 0;
521                    qunlock(&kbd);
522            }
523            break;
524    case Qauth:
525    case Qauthcheck:
526    case Qauthent:
527            authclose(c);
528    }
529  }
530
531  static long
532  consread(Chan *c, void *buf, long n, vlong off)
533  {
534    ulong l;
535    Mach *mp;
536    char *b, *bp;
537    char tmp[128];          /* must be >= 6*NUMSIZE */
538    char *cbuf = buf;
539    int ch, i, k, id, eol;
540    vlong offset = off;
541
542    if(n <= 0)
543            return n;
544    switch(c->qid.path & ~CHDIR){
```

```
545     case Qdir:
546             return devdirread(c, buf, n, consdir, nelem(consdir), devgen);
547
548     case Qcons:
549             qlock(&kbd);
550             if(waserror()) {
551                     qunlock(&kbd);
552                     nexterror();
553             }
554             if(kbd.raw) {
555                     if(qcanread(lineq))
556                             n = qread(lineq, buf, n);
557                     else {
558                             /* read as much as possible */
559                             do {
560                                     i = qread(kbdq, cbuf, n);
561                                     cbuf += i;
562                                     n -= i;
563                             } while (n>0 && qcanread(kbdq));
564                             n = cbuf - (char*)buf;
565                     }
566             } else {
567                     while(!qcanread(lineq)) {
568                             qread(kbdq, &kbd.line[kbd.x], 1);
569                             ch = kbd.line[kbd.x];
570                             if(kbd.raw){
571                                     qiwrite(lineq, kbd.line, kbd.x+1);
572                                     kbd.x = 0;
573                                     continue;
574                             }
575                             eol = 0;
576                             switch(ch){
577                             case '\b':
578                                     if(kbd.x)
579                                             kbd.x--;
580                                     break;
581                             case 0x15:
582                                     kbd.x = 0;
583                                     break;
584                             case '\n':
585                             case 0x04:
586                                     eol = 1;
587                             default:
588                                     kbd.line[kbd.x++] = ch;
589                                     break;
590                             }
591                             if(kbd.x == sizeof(kbd.line) || eol){
592                                     if(ch == 0x04)
593                                             kbd.x--;
594                                     qwrite(lineq, kbd.line, kbd.x);
595                                     kbd.x = 0;
596                             }
597                     }
598                     n = qread(lineq, buf, n);
599             }
600             qunlock(&kbd);
601             poperror();
602             return n;
603
604     case Qcputime:
605             k = offset;
606             if(k >= 6*NUMSIZE)
607                     return 0;
608             if(k+n > 6*NUMSIZE)
609                     n = 6*NUMSIZE - k;
610             /* easiest to format in a separate buffer and copy out */
611             for(i=0; i<6 && NUMSIZE*i<k+n; i++){
612                     l = up->time[i];
613                     if(i == TReal)
614                             l = MACHP(0)->ticks - l;
615                     l = TK2MS(l);
616                     readnum(0, tmp+NUMSIZE*i, NUMSIZE, l, NUMSIZE);
617             }
618             memmove(buf, tmp+k, n);
619             return n;
620
621     case Qpgrpid:
622             return readnum((ulong)offset, buf, n, up->pgrp->pgrpid, NUMSIZE);
623
624     case Qpid:
625             return readnum((ulong)offset, buf, n, up->pid, NUMSIZE);
626
627     case Qppid:
628             return readnum((ulong)offset, buf, n, up->parentpid, NUMSIZE);
629
630     case Qtime:
631             return readtime((ulong)offset, buf, n);
632
633     case Qbintime:
634             return readbintime(buf, n);
635
636     case Qkey:
637             return keyread(buf, n, offset);
638
639     case Qauth:
640             return authread(c, cbuf, n);
641
642     case Qauthcheck:
643             return authcheckread(c, cbuf, n);
644
645     case Qauthent:
646             return authentread(c, cbuf, n);
647
648     case Qhostowner:
649             return readstr((ulong)offset, buf, n, eve);
650
651     case Qhostdomain:
652             return readstr((ulong)offset, buf, n, hostdomain);
653
654     case Quser:
655             return readstr((ulong)offset, buf, n, up->user);
656
657     case Qnull:
658             return 0;
659
660     case Qsysstat:
661             b = smalloc(conf.nmach*(NUMSIZE*8+1) + 1);      /* +1 for NUL */
662             bp = b;
663             for(id = 0; id < 32; id++) {
664                     if(active.machs & (1<<id)) {
665                             mp = MACHP(id);
666                             readnum(0, bp, NUMSIZE, id, NUMSIZE);
667                             bp += NUMSIZE;
668                             readnum(0, bp, NUMSIZE, mp->cs, NUMSIZE);
669                             bp += NUMSIZE;
670                             readnum(0, bp, NUMSIZE, mp->intr, NUMSIZE);
671                             bp += NUMSIZE;
672                             readnum(0, bp, NUMSIZE, mp->syscall, NUMSIZE);
673                             bp += NUMSIZE;
674                             readnum(0, bp, NUMSIZE, mp->pfault, NUMSIZE);
675                             bp += NUMSIZE;
676                             readnum(0, bp, NUMSIZE, mp->tlbfault, NUMSIZE);
677                             bp += NUMSIZE;
678                             readnum(0, bp, NUMSIZE, mp->tlbpurge, NUMSIZE);
679                             bp += NUMSIZE;
680                             readnum(0, bp, NUMSIZE, mp->load, NUMSIZE);
```

```
681                             bp += NUMSIZE;
682                             *bp++ = '\n';
683                     }
684             }
685             n = readstr((ulong)offset, buf, n, b);
686             free(b);
687             return n;
688
689     case Qswap:
690             sprint(tmp, "%lud/%lud memory %lud/%lud swap\n",
691                     palloc.user-palloc.freecount,
692                     palloc.user, conf.nswap-swapalloc.free, conf.nswap);
693
694             return readstr((ulong)offset, buf, n, tmp);
695
696     case Qsysname:
697             return readstr((ulong)offset, buf, n, sysname);
698
699     case Qrandom:
700             return randomread(buf, n);
701
702     case Qdrivers:
703             b = malloc(READSTR);
704             if(b == nil)
705                     error(Enomem);
706             n = 0;
707             for(i = 0; devtab[i] != nil; i++)
708                     n += snprint(b+n, READSTR-n, "#%C %s\n", devtab[i]->dc,  devtab[i]->name);
709             n = readstr((ulong)offset, buf, n, b);
710             free(b);
711             return n;
712
713     case Qzero:
714             memset(buf, 0, n);
715             return n;
716
717     default:
718             print("consread %lux\n", c->qid.path);
719             error(Egreg);
720     }
721     return -1;              /* never reached */
722   }
723
724   static long
725   conswrite(Chan *c, void *va, long n, vlong off)
726   {
727     char buf[256];
728     long l, bp;
729     char *a = va;
730     Mach *mp;
731     int id, fd;
732     Chan *swc;
733     ulong offset = off;
734
735     switch(c->qid.path){
736     case Qcons:
737             /*
738              * Can't page fault in putstrn, so copy the data locally.
739              */
740             l = n;
741             while(l > 0){
742                     bp = l;
743                     if(bp > sizeof buf)
744                             bp = sizeof buf;
745                     memmove(buf, a, bp);
746                     putstrn0(a, bp, 1);
747                     a += bp;
748                     l -= bp;
749             }
750             break;
751
752     case Qconsctl:
753             if(n >= sizeof(buf))
754                     n = sizeof(buf)-1;
755             strncpy(buf, a, n);
756             buf[n] = 0;
757             for(a = buf; a;){
758                     if(strncmp(a, "rawon", 5) == 0){
759                             qlock(&kbd);
760                             if(kbd.x){
761                                     qwrite(kbdq, kbd.line, kbd.x);
762                                     kbd.x = 0;
763                             }
764                             kbd.raw = 1;
765                             qunlock(&kbd);
766                     } else if(strncmp(a, "rawoff", 6) == 0){
767                             kbd.raw = 0;
768                             kbd.x = 0;
769                     } else if(strncmp(a, "ctlpon", 6) == 0){
770                             kbd.ctlpoff = 0;
771                     } else if(strncmp(a, "ctlpoff", 7) == 0){
772                             kbd.ctlpoff = 1;
773                     }
774                     if(a = strchr(a, ' '))
775                             a++;
776             }
777             break;
778
779     case Qtime:
780             if(!iseve())
781                     error(Eperm);
782             return writetime(a, n);
783
784     case Qbintime:
785             if(!iseve())
786                     error(Eperm);
787             return writebintime(a, n);
788
789     case Qkey:
790             return keywrite(a, n);
791
792     case Qhostowner:
793             return hostownerwrite(a, n);
794
795     case Qhostdomain:
796             return hostdomainwrite(a, n);
797
798     case Quser:
799             return userwrite(a, n);
800
801     case Qauth:
802             return authwrite(c, a, n);
803
804     case Qauthcheck:
805             return authcheck(c, a, n);
806
807     case Qauthent:
808             return authentwrite(c, a, n);
809
810     case Qnull:
811             break;
812
813     case Qreboot:
814             if(!iseve())
815                     error(Eperm);
816             if(strncmp(a, "reboot", 6) == 0){
```

```
817                     print("conswrite: reboot\n");
818                     exit(0);
819             }
820             if(strncmp(a, "malloc", 6) == 0){       /* rsc bug */
821                     a = malloc(2);
822                     strcpy(a, "hi");
823                     free(a);
824                     a = malloc(2);
825                     strcpy(a, "helo");
826                     free(a);
827                     panic("not reached conswrite");
828             }
829             if(strncmp(a, "panic", 5) == 0)
830                     panic("/dev/reboot");
831             break;
832
833     case Qsysstat:
834             for(id = 0; id < 32; id++) {
835                     if(active.machs & (1<<id)) {
836                             mp = MACHP(id);
837                             mp->cs = 0;
838                             mp->intr = 0;
839                             mp->syscall = 0;
840                             mp->pfault = 0;
841                             mp->tlbfault = 0;
842                             mp->tlbpurge = 0;
843                     }
844             }
845             break;
846
847     case Qswap:
848             if(n >= sizeof buf)
849                     error(Egreg);
850             memmove(buf, va, n);    /* so we can NUL-terminate */
851             buf[n] = 0;
852             /* start a pager if not already started */
853             if(strncmp(buf, "start", 5) == 0){
854                     kickpager();
855                     break;
856             }
857             if(cpuserver && !iseve())
858                     error(Eperm);
859             if(buf[0]<'0' || '9'<buf[0])
860                     error(Ebadarg);
861             fd = strtoul(buf, 0, 0);
862             swc = fdtochan(fd, -1, 1, 1);
863             setswapchan(swc);
864             break;
865
866     case Qsysname:
867             if(offset != 0)
868                     error(Ebadarg);
869             if(n <= 0 || n >= NAMELEN)
870                     error(Ebadarg);
871             strncpy(sysname, a, n);
872             sysname[n] = 0;
873             if(sysname[n-1] == '\n')
874                     sysname[n-1] = 0;
875             break;
876
877     default:
878             print("conswrite: %lud\n", c->qid.path);
879             error(Egreg);
880     }
881     return n;
882   }
883
884   void
885   setterm(char *f)
886   {
887     char buf[2*NAMELEN];
888
889     sprint(buf, f, conffile);
890     ksetenv("terminal", buf);
891   }
892
893   Dev consdevtab = {
894     'c',
895     "cons",
896
897     devreset,
898     consinit,
899     consattach,
900     devclone,
901     conswalk,
902     consstat,
903     consopen,
904     devcreate,
905     consclose,
906     consread,
907     devbread,
908     conswrite,
909     devbwrite,
910     devremove,
911     devwstat,
912   };
913
914   struct Rb
915   {
916     QLock;
917     Rendez  producer;
918     Rendez  consumer;
919     ulong   randomcount;
920     uchar   buf[1024];
921     uchar   *ep;
922     uchar   *rp;
923     uchar   *wp;
924     uchar   next;
925     uchar   wakeme;
926     ushort  bits;
927     ulong   randn;
928   } rb;
929
930   static void
931   seedrand(void)
932   {
933     randomread((void*)&rb.randn, sizeof(rb.randn));
934   }
935
936   int
937   nrand(int n)
938   {
939     if(rb.randn == 0)
940             seedrand();
941     rb.randn = rb.randn*1103515245 + 12345 + MACHP(0)->ticks;
942     return (rb.randn>>16) % n;
943   }
944
945   int
946   rand(void)
947   {
948     nrand(1);
949     return rb.randn;
950   }
951
952
```

```
static int
rbnotfull(void*)
{
  int i;

  i = rb.rp - rb.wp;
  return i != 1 && i != (1 - sizeof(rb.buf));
}

static int
rbnotempty(void*)
{
  return rb.wp != rb.rp;
}

void
genrandom(void*)
{
  up->basepri = PriNormal;
  up->priority = up->basepri;

  for(;;){
          for(;;)
                  if(++rb.randomcount > 100000)
                          break;
          if(anyhigher())
                  sched();
          if(!rbnotfull(0))
                  sleep(&rb.producer, rbnotfull, 0);
  }
}

/*
 *  produce random bits in a circular buffer
 */
static void
randomclock(void)
{
  if(rb.randomcount == 0 || !rbnotfull(0))
          return;

  rb.bits = (rb.bits<<2) ^ rb.randomcount;
  rb.randomcount = 0;

  rb.next++;
  if(rb.next != 8/2)
          return;
  rb.next = 0;

  *rb.wp ^= rb.bits;
  if(rb.wp+1 == rb.ep)
          rb.wp = rb.buf;
  else
          rb.wp = rb.wp+1;

  if(rb.wakeme)
          wakeup(&rb.consumer);
}

static void
randominit(void)
{
  addclock0link(randomclock);
  rb.ep = rb.buf + sizeof(rb.buf);
  rb.rp = rb.wp = rb.buf;
  kproc("genrandom", genrandom, 0);
}

/*
 *  consume random bytes from a circular buffer
 */
static ulong
randomread(void *xp, ulong n)
{
  uchar *e, *p;
  ulong x;

  p = xp;

  if(waserror()){
          qunlock(&rb);
          nexterror();
  }

  qlock(&rb);
  for(e = p + n; p < e; ){
          if(rb.wp == rb.rp){
                  rb.wakeme = 1;
                  wakeup(&rb.producer);
                  sleep(&rb.consumer, rbnotempty, 0);
                  rb.wakeme = 0;
                  continue;
          }

          /*
           *  beating clocks will be precictable if
           *  they are synchronized.  Use a cheap pseudo
           *  random number generator to obscure any cycles.
           */
          x = rb.randn*1103515245 ^ *rb.rp;
          *p++ = rb.randn = x;

          if(rb.rp+1 == rb.ep)
                  rb.rp = rb.buf;
          else
                  rb.rp = rb.rp+1;
  }
  qunlock(&rb);
  poperror();

  wakeup(&rb.producer);

  return n;
}

static uvlong uvorder = 0x0001020304050607ULL;

static uchar*
le2vlong(vlong *to, uchar *f)
{
  uchar *t, *o;
  int i;

  t = (uchar*)to;
  o = (uchar*)&uvorder;
  for(i = 0; i < sizeof(vlong); i++)
          t[o[i]] = f[i];
  return f+sizeof(vlong);
}

static uchar*
vlong2le(uchar *t, vlong from)
{
  uchar *f, *o;
  int i;
```

```
1089    f = (uchar*)&from;
1090    o = (uchar*)&uvorder;
1091    for(i = 0; i < sizeof(vlong); i++)
1092            t[i] = f[o[i]];
1093    return t+sizeof(vlong);
1094  }
1095
1096  static long order = 0x00010203;
1097
1098  static uchar*
1099  le2long(long *to, uchar *f)
1100  {
1101    uchar *t, *o;
1102    int i;
1103
1104    t = (uchar*)to;
1105    o = (uchar*)&order;
1106    for(i = 0; i < sizeof(long); i++)
1107            t[o[i]] = f[i];
1108    return f+sizeof(long);
1109  }
1110
1111  static uchar*
1112  long2le(uchar *t, long from)
1113  {
1114    uchar *f, *o;
1115    int i;
1116
1117    f = (uchar*)&from;
1118    o = (uchar*)&order;
1119    for(i = 0; i < sizeof(long); i++)
1120            t[i] = f[o[i]];
1121    return t+sizeof(long);
1122  }
1123
1124  char *Ebadtimectl = "bad time control";
1125
1126  /*
1127   *  like the old #c/time but with added info.  Return
1128   *
1129   *        secs    nanosecs        fastticks       fasthz
1130   */
1131  static int
1132  readtime(ulong off, char *buf, int n)
1133  {
1134    vlong   nsec, ticks;
1135    long sec;
1136    char str[7*NUMSIZE+4];  // extra 4 bytes are null plus doprint
1137                            // reserving space for a frigging UTF
1138                            // char
1139
1140    nsec = todget(&ticks);
1141    if(fasthz == 0LL)
1142            fastticks((uvlong*)&fasthz);
1143    sec = nsec/1000000000ULL;
1144    snprint(str, sizeof(str), "%*.0lud %*.0llud %*.0llud %*.0llud ",
1145            NUMSIZE-1, sec,
1146            VLNUMSIZE-1, nsec,
1147            VLNUMSIZE-1, ticks,
1148            VLNUMSIZE-1, fasthz);
1149    return readstr(off, buf, n, str);
1150  }
1151
1152  /*
1153   *  set the time in seconds
1154   */
1155  static int
1156  writetime(char *buf, int n)
1157  {
1158    char b[13];
1159    long i;
1160    vlong now;
1161
1162    if(n >= sizeof(b))
1163            error(Ebadtimectl);
1164    strncpy(b, buf, n);
1165    b[n] = 0;
1166    i = strtol(b, 0, 0);
1167    if(i <= 0)
1168            error(Ebadtimectl);
1169    now = i*1000000000LL;
1170    todset(now, 0, 0);
1171    return n;
1172  }
1173
1174  /*
1175   *  read binary time info.  all numbers are little endian.
1176   *  ticks and nsec are syncronized.
1177   */
1178  static int
1179  readbintime(char *buf, int n)
1180  {
1181    int i;
1182    vlong nsec, ticks;
1183    uchar *b = (uchar*)buf;
1184
1185    i = 0;
1186    if(fasthz == 0LL)
1187            fastticks((uvlong*)&fasthz);
1188    nsec = todget(&ticks);
1189    if(n >= 3*sizeof(uvlong)){
1190            vlong2le(b+2*sizeof(uvlong), fasthz);
1191            i += sizeof(uvlong);
1192    }
1193    if(n >= 2*sizeof(uvlong)){
1194            vlong2le(b+sizeof(uvlong), ticks);
1195            i += sizeof(uvlong);
1196    }
1197    if(n >= 8){
1198            vlong2le(b, nsec);
1199            i += sizeof(vlong);
1200    }
1201    return i;
1202  }
1203
1204  /*
1205   *  set any of the following
1206   *        - time in nsec
1207   *        - nsec trim applied over some seconds
1208   *        - clock frequency
1209   */
1210  static int
1211  writebintime(char *buf, int n)
1212  {
1213    uchar *p;
1214    vlong delta;
1215    long period;
1216
1217    n--;
1218    p = (uchar*)buf + 1;
1219    switch(*buf){
1220    case 'n':
1221            if(n < sizeof(vlong))
1222                    error(Ebadtimectl);
1223            le2vlong(&delta, p);
1224            todset(delta, 0, 0);
```

```
1225            break;
1226    case 'd':
1227            if(n < sizeof(vlong)+sizeof(long))
1228                    error(Ebadtimectl);
1229            p = le2vlong(&delta, p);
1230            le2long(&period, p);
1231            todset(-1, delta, period);
1232            break;
1233    case 'f':
1234            if(n < sizeof(uvlong))
1235                    error(Ebadtimectl);
1236            le2vlong(&fasthz, p);
1237            todsetfreq(fasthz);
1238            break;
1239    }
1240    return n;
1241  }
```

```
#include "u.h"
#include "../port/lib.h"
#include "mem.h"
#include "dat.h"
#include "fns.h"
#include "io.h"
#include "ureg.h"
#include "../port/error.h"
#include "../port/netif.h"

#include "etherif.h"

static Ether *etherxx[MaxEther];

Chan*
etherattach(char* spec)
{
  ulong ctlrno;
  char *p;
  Chan *chan;

  ctlrno = 0;
  if(spec && *spec){
          ctlrno = strtoul(spec, &p, 0);
          if((ctlrno == 0 && p == spec) || *p || (ctlrno >= MaxEther))
                  error(Ebadarg);
  }
  if(etherxx[ctlrno] == 0)
          error(Enodev);

  chan = devattach('l', spec);
  chan->dev = ctlrno;
  if(etherxx[ctlrno]->attach)
          etherxx[ctlrno]->attach(etherxx[ctlrno]);
  return chan;
}

static int
etherwalk(Chan* chan, char* name)
{
  return netifwalk(etherxx[chan->dev], chan, name);
}

static void
etherstat(Chan* chan, char* dp)
{
  netifstat(etherxx[chan->dev], chan, dp);
}

static Chan*
etheropen(Chan* chan, int omode)
{
  return netifopen(etherxx[chan->dev], chan, omode);
}

static void
ethercreate(Chan*, char*, int, ulong)
{
}

static void
etherclose(Chan* chan)
{
  netifclose(etherxx[chan->dev], chan);
}

static long
etherread(Chan* chan, void* buf, long n, vlong off)
{
  Ether *ether;
  ulong offset = off;

  ether = etherxx[chan->dev];
  if((chan->qid.path & CHDIR) == 0 && ether->ifstat){
          /*
           * With some controllers it is necessary to reach
           * into the chip to extract statistics.
           */
          if(NETTYPE(chan->qid.path) == Nifstatqid)
                  return ether->ifstat(ether, buf, n, offset);
          else if(NETTYPE(chan->qid.path) == Nstatqid)
                  ether->ifstat(ether, buf, 0, offset);
  }

  return netifread(ether, chan, buf, n, offset);
}

static Block*
etherbread(Chan* chan, long n, ulong offset)
{
  return netifbread(etherxx[chan->dev], chan, n, offset);
}

static void
etherremove(Chan*)
{
}

static void
etherwstat(Chan* chan, char* dp)
{
  netifwstat(etherxx[chan->dev], chan, dp);
}

static void
etherrtrace(Netfile* f, Etherpkt* pkt, int len)
{
  int i, n;
  Block *bp;

  if(qwindow(f->in) <= 0)
          return;
  if(len > 64)
          n = 64;
  else
          n = len;
  bp = iallocb(n);
  if(bp == 0)
          return;
  memmove(bp->wp, pkt->d, n);
  i = TK2MS(MACHP(0)->ticks);
  bp->wp[58] = len>>8;
  bp->wp[59] = len;
  bp->wp[60] = i>>24;
  bp->wp[61] = i>>16;
  bp->wp[62] = i>>8;
  bp->wp[63] = i;
  bp->wp += 64;
  qpass(f->in, bp);
}

Block*
etheriq(Ether* ether, Block* bp, int fromwire)
{
  Etherpkt *pkt;
  ushort type;
```

```
137     int len, multi, tome, fromme;
138     Netfile **ep, *f, **fp, *fx;
139     Block *xbp;
140
141     ether->inpackets++;
142
143     pkt = (Etherpkt*)bp->rp;
144     len = BLEN(bp);
145     type = (pkt->type[0]<<8)|pkt->type[1];
146     fx = 0;
147     ep = &ether->f[Ntypes];
148
149     multi = pkt->d[0] & 1;
150     /* check for valid multcast addresses */
151     if(multi && memcmp(pkt->d, ether->bcast, sizeof(pkt->d)) && ether->prom == 0){
152             if(!activemulti(ether, pkt->d, sizeof(pkt->d))){
153                     if(fromwire){
154                             freeb(bp);
155                             bp = 0;
156                     }
157                     return bp;
158             }
159     }
160
161     /* is it for me? */
162     tome = memcmp(pkt->d, ether->ea, sizeof(pkt->d)) == 0;
163     fromme = memcmp(pkt->s, ether->ea, sizeof(pkt->s)) == 0;
164
165     /*
166      * Multiplex the packet to all the connections which want it.
167      * If the packet is not to be used subsequently (fromwire != 0),
168      * attempt to simply pass it into one of the connections, thereby
169      * saving a copy of the data (usual case hopefully).
170      */
171     for(fp = ether->f; fp < ep; fp++){
172             if(f = *fp)
173             if(f->type == type || f->type < 0)
174             if(tome || multi || f->prom){
175                     /* Don't want to hear bridged packets */
176                     if(f->bridge && !fromwire && !fromme)
177                             continue;
178                     if(!f->headersonly){
179                             if(fromwire && fx == 0)
180                                     fx = f;
181                             else if(xbp = iallocb(len)){
182                                     memmove(xbp->wp, pkt, len);
183                                     xbp->wp += len;
184                                     qpass(f->in, xbp);
185                             }
186                             else
187                                     ether->soverflows++;
188                     }
189                     else
190                             etherrtrace(f, pkt, len);
191             }
192     }
193
194     if(fx){
195             if(qpass(fx->in, bp) < 0)
196                     ether->soverflows++;
197             return 0;
198     }
199     if(fromwire){
200             freeb(bp);
201             return 0;
202     }
203
204     return bp;
205   }
206
207   static int
208   etheroq(Ether* ether, Block* bp)
209   {
210     int len, loopback, s;
211     Etherpkt *pkt;
212
213     ether->outpackets++;
214
215     /*
216      * Check if the packet has to be placed back onto the input queue,
217      * i.e. if it's a loopback or broadcast packet or the interface is
218      * in promiscuous mode.
219      * If it's a loopback packet indicate to etheriq that the data isn't
220      * needed and return, etheriq will pass-on or free the block.
221      * To enable bridging to work, only packets that were originated
222      * by this interface are fed back.
223      */
224     pkt = (Etherpkt*)bp->rp;
225     len = BLEN(bp);
226     loopback = memcmp(pkt->d, ether->ea, sizeof(pkt->d)) == 0;
227     if(loopback || memcmp(pkt->d, ether->bcast, sizeof(pkt->d)) == 0 || ether->prom){
228             s = splhi();
229             etheriq(ether, bp, 0);
230             splx(s);
231     }
232
233     if(!loopback){
234             qbwrite(ether->oq, bp);
235             ether->transmit(ether);
236     } else
237             freeb(bp);
238
239     return len;
240   }
241
242   static long
243   etherwrite(Chan* chan, void* buf, long n, vlong)
244   {
245     Ether *ether;
246     Block *bp;
247
248     ether = etherxx[chan->dev];
249     if(NETTYPE(chan->qid.path) != Ndataqid)
250             return netifwrite(ether, chan, buf, n);
251
252     if(n > ETHERMAXTU)
253             error(Etoobig);
254     if(n < ETHERMINTU)
255             error(Etoosmall);
256
257     bp = allocb(n);
258     if(waserror()){
259             freeb(bp);
260             nexterror();
261     }
262     memmove(bp->rp, buf, n);
263     memmove(bp->rp+Eaddrlen, ether->ea, Eaddrlen);
264     poperror();
265     bp->wp += n;
266
267     return etheroq(ether, bp);
268   }
269
270   static long
271   etherbwrite(Chan* chan, Block* bp, ulong)
272   {
```

```
273     Ether *ether;
274     long n;
275
276     n = BLEN(bp);
277     ether = etherxx[chan->dev];
278     if(NETTYPE(chan->qid.path) != Ndataqid){
279             n = netifwrite(ether, chan, bp->rp, n);
280             freeb(bp);
281             return n;
282     }
283
284     if(n > ETHERMAXTU){
285             freeb(bp);
286             error(Ebadarg);
287     }
288     if(n < ETHERMINTU){
289             freeb(bp);
290             error(Etoosmall);
291     }
292
293     return etheroq(ether, bp);
294   }
295
296   static struct {
297     char*   type;
298     int     (*reset)(Ether*);
299   } cards[MaxEther+1];
300
301   void
302   addethercard(char* t, int (*r)(Ether*))
303   {
304     static int ncard;
305
306     if(ncard == MaxEther)
307             panic("too many ether cards");
308     cards[ncard].type = t;
309     cards[ncard].reset = r;
310     ncard++;
311   }
312
313   int
314   parseether(uchar *to, char *from)
315   {
316     char nip[4];
317     char *p;
318     int i;
319
320     p = from;
321     for(i = 0; i < 6; i++){
322             if(*p == 0)
323                     return -1;
324             nip[0] = *p++;
325             if(*p == 0)
326                     return -1;
327             nip[1] = *p++;
328             nip[2] = 0;
329             to[i] = strtoul(nip, 0, 16);
330             if(*p == ':')
331                     p++;
332     }
333     return 0;
334   }
335
336   static void
337   etherreset(void)
338   {
339     Ether *ether;
340     int i, n, ctlrno;
341     char name[NAMELEN], buf[128];
342
343     for(ether = 0, ctlrno = 0; ctlrno < MaxEther; ctlrno++){
344             if(ether == 0)
345                     ether = malloc(sizeof(Ether));
346             memset(ether, 0, sizeof(Ether));
347             ether->ctlrno = ctlrno;
348             ether->tbdf = BUSUNKNOWN;
349             ether->mbps = 10;
350             if(isaconfig("ether", ctlrno, ether) == 0)
351                     continue;
352             for(n = 0; cards[n].type; n++){
353                     if(cistrcmp(cards[n].type, ether->type))
354                             continue;
355                     for(i = 0; i < ether->nopt; i++){
356                             if(strncmp(ether->opt[i], "ea=", 3))
357                                     continue;
358                             if(parseether(ether->ea, &ether->opt[i][3]) == -1)
359                                     memset(ether->ea, 0, Eaddrlen);
360                     }
361                     if(cards[n].reset(ether))
362                             break;
363
364                     /*
365                      * IRQ2 doesn't really exist, it's used to gang the interrupt
366                      * controllers together. A device set to IRQ2 will appear on
367                      * the second interrupt controller as IRQ9.
368                      */
369                     if(ether->irq == 2)
370                             ether->irq = 9;
371                     snprint(name, sizeof(name), "ether%d", ctlrno);
372                     intrenable(ether->irq, ether->interrupt, ether, ether->tbdf, name);
373
374                     i = sprint(buf, "#l%d: %s: %dMbps port 0x%luX irq %lud",
375                             ctlrno, ether->type, ether->mbps, ether->port, ether->irq);
376                     if(ether->mem)
377                             i += sprint(buf+i, " addr 0x%luX", PADDR(ether->mem));
378                     if(ether->size)
379                             i += sprint(buf+i, " size 0x%luX", ether->size);
380                     i += sprint(buf+i, ": %2.2uX%2.2uX%2.2uX%2.2uX%2.2uX%2.2uX",
381                             ether->ea[0], ether->ea[1], ether->ea[2],
382                             ether->ea[3], ether->ea[4], ether->ea[5]);
383                     sprint(buf+i, "\n");
384                     print(buf);
385
386                     if(ether->mbps == 100){
387                             netifinit(ether, name, Ntypes, 256*1024);
388                             if(ether->oq == 0)
389                                     ether->oq = qopen(256*1024, 1, 0, 0);
390                     }
391                     else{
392                             netifinit(ether, name, Ntypes, 65*1024);
393                             if(ether->oq == 0)
394                                     ether->oq = qopen(65*1024, 1, 0, 0);
395                     }
396                     if(ether->oq == 0)
397                             panic("etherreset %s", name);
398                     ether->alen = Eaddrlen;
399                     memmove(ether->addr, ether->ea, Eaddrlen);
400                     memset(ether->bcast, 0xFF, Eaddrlen);
401
402                     etherxx[ctlrno] = ether;
403                     ether = 0;
404                     break;
405             }
406     }
407     if(ether)
408             free(ether);
```

```
409   }
410
411   #define POLY 0xedb88320
412
413   /* really slow 32 bit crc for ethers */
414   ulong
415   ethercrc(uchar *p, int len)
416   {
417     int i, j;
418     ulong crc, b;
419
420     crc = 0xffffffff;
421     for(i = 0; i < len; i++){
422             b = *p++;
423             for(j = 0; j < 8; j++){
424                     crc = (crc>>1) ^ (((crc^b) & 1) ? POLY : 0);
425                     b >>= 1;
426             }
427     }
428     return crc;
429   }
430
431   Dev etherdevtab = {
432     'l',
433     "ether",
434
435     etherreset,
436     devinit,
437     etherattach,
438     devclone,
439     etherwalk,
440     etherstat,
441     etheropen,
442     ethercreate,
443     etherclose,
444     etherread,
445     etherbread,
446     etherwrite,
447     etherbwrite,
448     etherremove,
449     etherwstat,
450   };
```

```
#include "u.h"
#include "../port/lib.h"
#include "mem.h"
#include "dat.h"
#include "fns.h"
#include "io.h"
#include "ureg.h"
#include "../port/error.h"
#include "../port/netif.h"

#include "etherif.h"

static Ether *etherxx[MaxEther];

Chan*
etherattach(char* spec)
{
  ulong ctlrno;
  char *p;
  Chan *chan;

  ctlrno = 0;
  if(spec && *spec){
          ctlrno = strtoul(spec, &p, 0);
          if((ctlrno == 0 && p == spec) || *p || (ctlrno >= MaxEther))
                  error(Ebadarg);
  }
  if(etherxx[ctlrno] == 0)
          error(Enodev);

  chan = devattach('l', spec);
  chan->dev = ctlrno;
  if(etherxx[ctlrno]->attach)
          etherxx[ctlrno]->attach(etherxx[ctlrno]);
  return chan;
}

static int
etherwalk(Chan* chan, char* name)
{
  return netifwalk(etherxx[chan->dev], chan, name);
}

static void
etherstat(Chan* chan, char* dp)
{
  netifstat(etherxx[chan->dev], chan, dp);
}

static Chan*
etheropen(Chan* chan, int omode)
{
  return netifopen(etherxx[chan->dev], chan, omode);
}

static void
ethercreate(Chan*, char*, int, ulong)
{
}

static void
etherclose(Chan* chan)
{
  netifclose(etherxx[chan->dev], chan);
}

static long
etherread(Chan* chan, void* buf, long n, vlong off)
{
  Ether *ether;
  ulong offset = off;

  ether = etherxx[chan->dev];
  if((chan->qid.path & CHDIR) == 0 && ether->ifstat){
          /*
           * With some controllers it is necessary to reach
           * into the chip to extract statistics.
           */
          if(NETTYPE(chan->qid.path) == Nifstatqid)
                  return ether->ifstat(ether, buf, n, offset);
          else if(NETTYPE(chan->qid.path) == Nstatqid)
                  ether->ifstat(ether, buf, 0, offset);
  }

  return netifread(ether, chan, buf, n, offset);
}

static Block*
etherbread(Chan* chan, long n, ulong offset)
{
  return netifbread(etherxx[chan->dev], chan, n, offset);
}

static void
etherremove(Chan*)
{
}

static void
etherwstat(Chan* chan, char* dp)
{
  netifwstat(etherxx[chan->dev], chan, dp);
}

static void
etherrtrace(Netfile* f, Etherpkt* pkt, int len)
{
  int i, n;
  Block *bp;

  if(qwindow(f->in) <= 0)
          return;
  if(len > 64)
          n = 64;
  else
          n = len;
  bp = iallocb(n);
  if(bp == 0)
          return;
  memmove(bp->wp, pkt->d, n);
  i = TK2MS(MACHP(0)->ticks);
  bp->wp[58] = len>>8;
  bp->wp[59] = len;
  bp->wp[60] = i>>24;
  bp->wp[61] = i>>16;
  bp->wp[62] = i>>8;
  bp->wp[63] = i;
  bp->wp += 64;
  qpass(f->in, bp);
}

Block*
etheriq(Ether* ether, Block* bp, int fromwire)
{
  Etherpkt *pkt;
  ushort type;
```

```
137     int len, multi, tome, fromme;
138     Netfile **ep, *f, **fp, *fx;
139     Block *xbp;
140
141     ether->inpackets++;
142
143     pkt = (Etherpkt*)bp->rp;
144     len = BLEN(bp);
145     type = (pkt->type[0]<<8)|pkt->type[1];
146     fx = 0;
147     ep = &ether->f[Ntypes];
148
149     multi = pkt->d[0] & 1;
150     /* check for valid multcast addresses */
151     if(multi && memcmp(pkt->d, ether->bcast, sizeof(pkt->d)) && ether->prom == 0){
152             if(!activemulti(ether, pkt->d, sizeof(pkt->d))){
153                     if(fromwire){
154                             freeb(bp);
155                             bp = 0;
156                     }
157                     return bp;
158             }
159     }
160
161     /* is it for me? */
162     tome = memcmp(pkt->d, ether->ea, sizeof(pkt->d)) == 0;
163     fromme = memcmp(pkt->s, ether->ea, sizeof(pkt->s)) == 0;
164
165     /*
166      * Multiplex the packet to all the connections which want it.
167      * If the packet is not to be used subsequently (fromwire != 0),
168      * attempt to simply pass it into one of the connections, thereby
169      * saving a copy of the data (usual case hopefully).
170      */
171     for(fp = ether->f; fp < ep; fp++){
172             if(f = *fp)
173             if(f->type == type || f->type < 0)
174             if(tome || multi || f->prom){
175                     /* Don't want to hear bridged packets */
176                     if(f->bridge && !fromwire && !fromme)
177                             continue;
178                     if(!f->headersonly){
179                             if(fromwire && fx == 0)
180                                     fx = f;
181                             else if(xbp = iallocb(len)){
182                                     memmove(xbp->wp, pkt, len);
183                                     xbp->wp += len;
184                                     qpass(f->in, xbp);
185                             }
186                             else
187                                     ether->soverflows++;
188                     }
189                     else
190                             etherrtrace(f, pkt, len);
191             }
192     }
193
194     if(fx){
195             if(qpass(fx->in, bp) < 0)
196                     ether->soverflows++;
197             return 0;
198     }
199     if(fromwire){
200             freeb(bp);
201             return 0;
202     }
203
204     return bp;
205   }
206
207   static int
208   etheroq(Ether* ether, Block* bp)
209   {
210     int len, loopback, s;
211     Etherpkt *pkt;
212
213     ether->outpackets++;
214
215     /*
216      * Check if the packet has to be placed back onto the input queue,
217      * i.e. if it's a loopback or broadcast packet or the interface is
218      * in promiscuous mode.
219      * If it's a loopback packet indicate to etheriq that the data isn't
220      * needed and return, etheriq will pass-on or free the block.
221      * To enable bridging to work, only packets that were originated
222      * by this interface are fed back.
223      */
224     pkt = (Etherpkt*)bp->rp;
225     len = BLEN(bp);
226     loopback = memcmp(pkt->d, ether->ea, sizeof(pkt->d)) == 0;
227     if(loopback || memcmp(pkt->d, ether->bcast, sizeof(pkt->d)) == 0 || ether->prom){
228             s = splhi();
229             etheriq(ether, bp, 0);
230             splx(s);
231     }
232
233     if(!loopback){
234             qbwrite(ether->oq, bp);
235             ether->transmit(ether);
236     } else
237             freeb(bp);
238
239     return len;
240   }
241
242   static long
243   etherwrite(Chan* chan, void* buf, long n, vlong)
244   {
245     Ether *ether;
246     Block *bp;
247
248     ether = etherxx[chan->dev];
249     if(NETTYPE(chan->qid.path) != Ndataqid)
250             return netifwrite(ether, chan, buf, n);
251
252     if(n > ETHERMAXTU)
253             error(Etoobig);
254     if(n < ETHERMINTU)
255             error(Etoosmall);
256
257     bp = allocb(n);
258     if(waserror()){
259             freeb(bp);
260             nexterror();
261     }
262     memmove(bp->rp, buf, n);
263     memmove(bp->rp+Eaddrlen, ether->ea, Eaddrlen);
264     poperror();
265     bp->wp += n;
266
267     return etheroq(ether, bp);
268   }
269
270   static long
271   etherbwrite(Chan* chan, Block* bp, ulong)
272   {
```

```
273     Ether *ether;
274     long n;
275
276     n = BLEN(bp);
277     ether = etherxx[chan->dev];
278     if(NETTYPE(chan->qid.path) != Ndataqid){
279             n = netifwrite(ether, chan, bp->rp, n);
280             freeb(bp);
281             return n;
282     }
283
284     if(n > ETHERMAXTU){
285             freeb(bp);
286             error(Ebadarg);
287     }
288     if(n < ETHERMINTU){
289             freeb(bp);
290             error(Etoosmall);
291     }
292
293     return etheroq(ether, bp);
294   }
295
296   static struct {
297     char*   type;
298     int     (*reset)(Ether*);
299   } cards[MaxEther+1];
300
301   void
302   addethercard(char* t, int (*r)(Ether*))
303   {
304     static int ncard;
305
306     if(ncard == MaxEther)
307             panic("too many ether cards");
308     cards[ncard].type = t;
309     cards[ncard].reset = r;
310     ncard++;
311   }
312
313   int
314   parseether(uchar *to, char *from)
315   {
316     char nip[4];
317     char *p;
318     int i;
319
320     p = from;
321     for(i = 0; i < 6; i++){
322             if(*p == 0)
323                     return -1;
324             nip[0] = *p++;
325             if(*p == 0)
326                     return -1;
327             nip[1] = *p++;
328             nip[2] = 0;
329             to[i] = strtoul(nip, 0, 16);
330             if(*p == ':')
331                     p++;
332     }
333     return 0;
334   }
335
336   static void
337   etherreset(void)
338   {
339     Ether *ether;
340     int i, n, ctlrno;
341     char name[NAMELEN], buf[128];
342
343     for(ether = 0, ctlrno = 0; ctlrno < MaxEther; ctlrno++){
344             if(ether == 0)
345                     ether = malloc(sizeof(Ether));
346             memset(ether, 0, sizeof(Ether));
347             ether->ctlrno = ctlrno;
348             ether->tbdf = BUSUNKNOWN;
349             ether->mbps = 10;
350             if(isaconfig("ether", ctlrno, ether) == 0)
351                     continue;
352             for(n = 0; cards[n].type; n++){
353                     if(cistrcmp(cards[n].type, ether->type))
354                             continue;
355                     for(i = 0; i < ether->nopt; i++){
356                             if(strncmp(ether->opt[i], "ea=", 3))
357                                     continue;
358                             if(parseether(ether->ea, &ether->opt[i][3]) == -1)
359                                     memset(ether->ea, 0, Eaddrlen);
360                     }
361                     if(cards[n].reset(ether))
362                             break;
363
364                     /*
365                      * IRQ2 doesn't really exist, it's used to gang the interrupt
366                      * controllers together. A device set to IRQ2 will appear on
367                      * the second interrupt controller as IRQ9.
368                      */
369                     if(ether->irq == 2)
370                             ether->irq = 9;
371                     snprint(name, sizeof(name), "ether%d", ctlrno);
372                     intrenable(ether->irq, ether->interrupt, ether, ether->tbdf, name);
373
374                     i = sprint(buf, "#l%d: %s: %dMbps port 0x%luX irq %lud",
375                             ctlrno, ether->type, ether->mbps, ether->port, ether->irq);
376                     if(ether->mem)
377                             i += sprint(buf+i, " addr 0x%luX", PADDR(ether->mem));
378                     if(ether->size)
379                             i += sprint(buf+i, " size 0x%luX", ether->size);
380                     i += sprint(buf+i, ": %2.2uX%2.2uX%2.2uX%2.2uX%2.2uX%2.2uX",
381                             ether->ea[0], ether->ea[1], ether->ea[2],
382                             ether->ea[3], ether->ea[4], ether->ea[5]);
383                     sprint(buf+i, "\n");
384                     print(buf);
385
386                     if(ether->mbps == 100){
387                             netifinit(ether, name, Ntypes, 256*1024);
388                             if(ether->oq == 0)
389                                     ether->oq = qopen(256*1024, 1, 0, 0);
390                     }
391                     else{
392                             netifinit(ether, name, Ntypes, 65*1024);
393                             if(ether->oq == 0)
394                                     ether->oq = qopen(65*1024, 1, 0, 0);
395                     }
396                     if(ether->oq == 0)
397                             panic("etherreset %s", name);
398                     ether->alen = Eaddrlen;
399                     memmove(ether->addr, ether->ea, Eaddrlen);
400                     memset(ether->bcast, 0xFF, Eaddrlen);
401
402                     etherxx[ctlrno] = ether;
403                     ether = 0;
404                     break;
405             }
406     }
407     if(ether)
408             free(ether);
```

```
409   }
410
411   #define POLY 0xedb88320
412
413   /* really slow 32 bit crc for ethers */
414   ulong
415   ethercrc(uchar *p, int len)
416   {
417     int i, j;
418     ulong crc, b;
419
420     crc = 0xffffffff;
421     for(i = 0; i < len; i++){
422             b = *p++;
423             for(j = 0; j < 8; j++){
424                     crc = (crc>>1) ^ (((crc^b) & 1) ? POLY : 0);
425                     b >>= 1;
426             }
427     }
428     return crc;
429   }
430
431   Dev etherdevtab = {
432     'l',
433     "ether",
434
435     etherreset,
436     devinit,
437     etherattach,
438     devclone,
439     etherwalk,
440     etherstat,
441     etheropen,
442     ethercreate,
443     etherclose,
444     etherread,
445     etherbread,
446     etherwrite,
447     etherbwrite,
448     etherremove,
449     etherwstat,
450   };
```

```
#include  "u.h"
#include  "../port/lib.h"
#include  "mem.h"
#include  "dat.h"
#include  "fns.h"
#include  "io.h"
#include  "../port/error.h"

#include  "floppy.h"

/* Intel 82077A (8272A compatible) floppy controller */

/* This module expects the following functions to be defined
 * elsewhere:
 *
 * inb()
 * outb()
 * floppyexec()
 * floppyeject()
 * floppysetup0()
 * floppysetup1()
 * dmainit()
 * dmasetup()
 * dmaend()
 *
 * On DMA systems, floppyexec() should be an empty function;
 * on non-DMA systems, dmaend() should be an empty function;
 * dmasetup() may enforce maximum transfer sizes.
 */

enum {
  /* file types */
  Qdir=           0,
  Qdata=          (1<<2),
  Qctl=           (2<<2),
  Qmask=          (3<<2),

  DMAchan=        2,      /* floppy dma channel */
};

#define DPRINT if(floppydebug)print
int floppydebug = 0;

/*
 *  types of drive (from PC equipment byte)
 */
enum
{
  Tnone=          0,
  T360kb=         1,
  T1200kb=        2,
  T720kb=         3,
  T1440kb=        4,
};

FType floppytype[] =
{
 { "3½HD",         T1440kb, 512, 18, 2, 1, 80, 0x1B, 0x54, 0, },
 { "3½DD",         T1440kb, 512,  9, 2, 1, 80, 0x1B, 0x54, 2, },
 { "3½DD",         T720kb,  512,  9, 2, 1, 80, 0x1B, 0x54, 2, },
 { "5¼HD",         T1200kb, 512, 15, 2, 1, 80, 0x2A, 0x50, 0, },
 { "5¼DD",         T1200kb, 512,  9, 2, 2, 40, 0x2A, 0x50, 1, },
 { "ATT3B1",      T1200kb, 512,  8, 2, 2, 48, 0x2A, 0x50, 1, },
 { "5¼DD",         T360kb,  512,  9, 2, 1, 40, 0x2A, 0x50, 2, },
};

/*
 *  bytes per sector encoding for the controller.
 *  - index for b2c is is (bytes per sector/128).
 *  - index for c2b is code from b2c
 */
static int b2c[] =
{
[1]       0,
[2]       1,
[4]       2,
[8]       3,
};
static int c2b[] =
{
  128,
  256,
  512,
  1024,
};

FController       fl;

#define MOTORBIT(i)       (1<<((i)+4))

/*
 *  predeclared
 */
static int        cmddone(void*);
static void       floppyformat(FDrive*, char*);
static void       floppykproc(void*);
static void       floppypos(FDrive*,long);
static int        floppyrecal(FDrive*);
static int        floppyresult(void);
static void       floppyrevive(void);
static long       floppyseek(FDrive*, long);
static int        floppysense(void);
static void       floppywait(void);
static long       floppyxfer(FDrive*, int, void*, long, long);

Dirtab floppydir[]={
  "fd0disk",              {Qdata + 0},    0,      0660,
  "fd0ctl",               {Qctl + 0},     0,      0660,
  "fd1disk",              {Qdata + 1},    0,      0660,
  "fd1ctl",               {Qctl + 1},     0,      0660,
  "fd2disk",              {Qdata + 2},    0,      0660,
  "fd2ctl",               {Qctl + 2},     0,      0660,
  "fd3disk",              {Qdata + 3},    0,      0660,
  "fd3ctl",               {Qctl + 3},     0,      0660,
};
#define NFDIR     2       /* directory entries/drive */

static void
fldump(void)
{
  DPRINT("sra %ux srb %ux dor %ux msr %ux dir %ux\n", inb(Psra), inb(Psrb),
          inb(Pdor), inb(Pmsr), inb(Pdir));
}

/*
 *  set floppy drive to its default type
 */
static void
floppysetdef(FDrive *dp)
{
  FType *t;

  for(t = floppytype; t < &floppytype[nelem(floppytype)]; t++)
          if(dp->dt == t->dt){
                  dp->t = t;
                  floppydir[NFDIR*dp->dev].length = dp->t->cap;
```

```
137                     break;
138             }
139     }
140
141     static void
142     floppyreset(void)
143     {
144       FDrive *dp;
145       FType *t;
146       ulong maxtsize;
147
148       floppysetup0(&fl);
149       if(fl.ndrive == 0)
150               return;
151
152       /*
153        *  init dependent parameters
154        */
155       maxtsize = 0;
156       for(t = floppytype; t < &floppytype[nelem(floppytype)]; t++){
157               t->cap = t->bytes * t->heads * t->sectors * t->tracks;
158               t->bcode = b2c[t->bytes/128];
159               t->tsize = t->bytes * t->sectors;
160               if(maxtsize < t->tsize)
161                       maxtsize = t->tsize;
162       }
163
164       dmainit(DMAchan, maxtsize);
165
166       /*
167        *  allocate the drive storage
168        */
169       fl.d = xalloc(fl.ndrive*sizeof(FDrive));
170       fl.selected = fl.d;
171
172       /*
173        *  stop the motors
174        */
175       fl.motor = 0;
176       delay(10);
177       outb(Pdor, fl.motor | Fintena | Fena);
178       delay(10);
179
180       /*
181        *  init drives
182        */
183       for(dp = fl.d; dp < &fl.d[fl.ndrive]; dp++){
184               dp->dev = dp - fl.d;
185               dp->dt = T1440kb;
186               floppysetdef(dp);
187               dp->cyl = -1;                   /* because we don't know */
188               dp->cache = (uchar*)xspanalloc(maxtsize, BY2PG, 64*1024);
189               dp->ccyl = -1;
190               dp->vers = 0;
191       }
192
193       /*
194        *  first operation will recalibrate
195        */
196       fl.confused = 1;
197
198       floppysetup1(&fl);
199     }
200
201     static Chan*
202     floppyattach(char *spec)
203     {
204       static int kstarted;
205
206       if(fl.ndrive == 0)
207               error(Enodev);
208
209       if(kstarted == 0){
210               /*
211                *  watchdog to turn off the motors
212                */
213               kstarted = 1;
214               kproc("floppy", floppykproc, 0);
215       }
216       return devattach('f', spec);
217     }
218
219     static int
220     floppywalk(Chan *c, char *name)
221     {
222       return devwalk(c, name, floppydir, fl.ndrive*NFDIR, devgen);
223     }
224
225     static void
226     floppystat(Chan *c, char *dp)
227     {
228       devstat(c, dp, floppydir, fl.ndrive*NFDIR, devgen);
229     }
230
231     static Chan*
232     floppyopen(Chan *c, int omode)
233     {
234       return devopen(c, omode, floppydir, fl.ndrive*NFDIR, devgen);
235     }
236
237     static void
238     floppyclose(Chan *)
239     {
240     }
241
242     static void
243     islegal(ulong offset, long n, FDrive *dp)
244     {
245       if(offset % dp->t->bytes)
246               error(Ebadarg);
247       if(n % dp->t->bytes)
248               error(Ebadarg);
249     }
250
251     /*
252      *  check if the floppy has been replaced under foot.  cause
253      *  an error if it has.
254      *
255      *  a seek and a read clears the condition.  this was determined
256      *  experimentally, there has to be a better way.
257      *
258      *  if the read fails, cycle through the possible floppy
259      *  density till one works or we've cycled through all
260      *  possibilities for this drive.
261      */
262     static void
263     changed(Chan *c, FDrive *dp)
264     {
265       ulong old;
266       FType *start;
267
268       /*
269        *  if floppy has changed or first time through
270        */
271       if((inb(Pdir)&Fchange) || dp->vers == 0){
272               DPRINT("changed\n");
```

```
273             fldump();
274             dp->vers++;
275             floppysetdef(dp);
276             start = dp->t;
277             dp->confused = 1;       /* make floppyon recal */
278             floppyon(dp);
279             floppyseek(dp, dp->t->heads*dp->t->tsize);
280             while(waserror()){
281                     while(++dp->t){
282                             if(dp->t == &floppytype[nelem(floppytype)])
283                                     dp->t = floppytype;
284                             if(dp->dt == dp->t->dt)
285                                     break;
286                     }
287                     floppydir[NFDIR*dp->dev].length = dp->t->cap;
288                     floppyon(dp);
289                     DPRINT("changed: trying %s\n", dp->t->name);
290                     fldump();
291                     if(dp->t == start)
292                             nexterror();
293             }
294             floppyxfer(dp, Fread, dp->cache, 0, dp->t->tsize);
295             poperror();
296     }
297
298     old = c->qid.vers;
299     c->qid.vers = dp->vers;
300     if(old && old != dp->vers)
301             error(Eio);
302   }
303
304   static int
305   readtrack(FDrive *dp, int cyl, int head)
306   {
307     int i, nn, sofar;
308     ulong pos;
309
310     nn = dp->t->tsize;
311     if(dp->ccyl==cyl && dp->chead==head)
312             return nn;
313     pos = (cyl*dp->t->heads+head) * nn;
314     for(sofar = 0; sofar < nn; sofar += i){
315             dp->ccyl = -1;
316             i = floppyxfer(dp, Fread, dp->cache + sofar, pos + sofar, nn - sofar);
317             if(i <= 0)
318                     return -1;
319     }
320     dp->ccyl = cyl;
321     dp->chead = head;
322     return nn;
323   }
324
325   static long
326   floppyread(Chan *c, void *a, long n, vlong off)
327   {
328     FDrive *dp;
329     long rv;
330     int sec, head, cyl;
331     long len;
332     uchar *aa;
333     ulong offset = off;
334
335     if(c->qid.path == CHDIR)
336             return devdirread(c, a, n, floppydir, fl.ndrive*NFDIR, devgen);
337
338     rv = 0;
339     dp = &fl.d[c->qid.path & ~Qmask];
340     switch ((int)(c->qid.path & Qmask)) {
341     case Qdata:
342             islegal(offset, n, dp);
343             aa = a;
344
345             qlock(&fl);
346             if(waserror()){
347                     qunlock(&fl);
348                     nexterror();
349             }
350             floppyon(dp);
351             changed(c, dp);
352             for(rv = 0; rv < n; rv += len){
353                     /*
354                      *  all xfers come out of the track cache
355                      */
356                     dp->len = n - rv;
357                     floppypos(dp, offset+rv);
358                     cyl = dp->tcyl;
359                     head = dp->thead;
360                     len = dp->len;
361                     sec = dp->tsec;
362                     if(readtrack(dp, cyl, head) < 0)
363                             break;
364                     memmove(aa+rv, dp->cache + (sec-1)*dp->t->bytes, len);
365             }
366             qunlock(&fl);
367             poperror();
368
369             break;
370     case Qctl:
371             return readstr(offset, a, n, dp->t->name);
372     default:
373             panic("floppyread: bad qid");
374     }
375
376     return rv;
377   }
378
379   #define SNCMP(a, b) strncmp(a, b, sizeof(b)-1)
380   static long
381   floppywrite(Chan *c, void *a, long n, vlong off)
382   {
383     FDrive *dp;
384     long rv, i;
385     char *aa = a;
386     char ctlmsg[64];
387     ulong offset = off;
388
389     rv = 0;
390     dp = &fl.d[c->qid.path & ~Qmask];
391     switch ((int)(c->qid.path & Qmask)) {
392     case Qdata:
393             islegal(offset, n, dp);
394             qlock(&fl);
395             if(waserror()){
396                     qunlock(&fl);
397                     nexterror();
398             }
399             floppyon(dp);
400             changed(c, dp);
401             for(rv = 0; rv < n; rv += i){
402                     floppypos(dp, offset+rv);
403                     if(dp->tcyl == dp->ccyl)
404                             dp->ccyl = -1;
405                     i = floppyxfer(dp, Fwrite, aa+rv, offset+rv, n-rv);
406                     if(i < 0)
407                             break;
408                     if(i == 0)
```

```
409                             error(Eio);
410             }
411             qunlock(&fl);
412             poperror();
413             break;
414     case Qctl:
415             rv = n;
416             qlock(&fl);
417             if(waserror()){
418                     qunlock(&fl);
419                     nexterror();
420             }
421             if(n >= sizeof(ctlmsg))
422                     n = sizeof(ctlmsg) - 1;
423             memmove(ctlmsg, aa, n);
424             ctlmsg[n] = 0;
425             if(SNCMP(ctlmsg, "eject") == 0){
426                     floppyeject(dp);
427             } else if(SNCMP(ctlmsg, "reset") == 0){
428                     fl.confused = 1;
429                     floppyon(dp);
430             } else if(SNCMP(ctlmsg, "format") == 0){
431                     floppyformat(dp, ctlmsg);
432             } else if(SNCMP(ctlmsg, "debug") == 0){
433                     floppydebug = 1;
434             } else
435                     error(Ebadctl);
436             poperror();
437             qunlock(&fl);
438             break;
439     default:
440             panic("floppywrite: bad qid");
441     }
442
443     return rv;
444   }
445
446   static void
447   floppykproc(void *)
448   {
449     FDrive *dp;
450
451     while(waserror())
452             ;
453     for(;;){
454             for(dp = fl.d; dp < &fl.d[fl.ndrive]; dp++){
455                     if((fl.motor&MOTORBIT(dp->dev))
456                     && TK2SEC(m->ticks - dp->lasttouched) > 5
457                     && canqlock(&fl)){
458                             if(TK2SEC(m->ticks - dp->lasttouched) > 5)
459                                     floppyoff(dp);
460                             qunlock(&fl);
461                     }
462             }
463             tsleep(&fl.kr, return0, 0, 1000);
464     }
465   }
466
467   /*
468    *  start a floppy drive's motor.
469    */
470   static void
471   floppyon(FDrive *dp)
472   {
473     int alreadyon;
474     int tries;
475
476     if(fl.confused)
477             floppyrevive();
478
479     /* start motor and select drive */
480     alreadyon = fl.motor & MOTORBIT(dp->dev);
481     fl.motor |= MOTORBIT(dp->dev);
482     outb(Pdor, fl.motor | Fintena | Fena | dp->dev);
483     if(!alreadyon){
484             /* wait for drive to spin up */
485             tsleep(&dp->r, return0, 0, 750);
486
487             /* clear any pending interrupts */
488             floppysense();
489     }
490
491     /* set transfer rate */
492     if(fl.rate != dp->t->rate){
493             fl.rate = dp->t->rate;
494             outb(Pdsr, fl.rate);
495     }
496
497     /* get drive to a known cylinder */
498     if(dp->confused)
499             for(tries = 0; tries < 4; tries++)
500                     if(floppyrecal(dp) >= 0)
501                             break;
502     dp->lasttouched = m->ticks;
503     fl.selected = dp;
504   }
505
506   /*
507    *  stop the floppy if it hasn't been used in 5 seconds
508    */
509   static void
510   floppyoff(FDrive *dp)
511   {
512     fl.motor &= ~MOTORBIT(dp->dev);
513     outb(Pdor, fl.motor | Fintena | Fena | dp->dev);
514   }
515
516   /*
517    *  send a command to the floppy
518    */
519   static int
520   floppycmd(void)
521   {
522     int i;
523     int tries;
524
525     fl.nstat = 0;
526     for(i = 0; i < fl.ncmd; i++){
527             for(tries = 0; ; tries++){
528                     if((inb(Pmsr)&(Ffrom|Fready)) == Fready)
529                             break;
530                     if(tries > 1000){
531                             DPRINT("cmd %ux can't be sent (%d)\n", fl.cmd[0], i);
532                             fldump();
533
534                             /* empty fifo, might have been a bad command */
535                             floppyresult();
536                             return -1;
537                     }
538                     microdelay(8);  /* for machine independence */
539             }
540             outb(Pfdata, fl.cmd[i]);
541     }
542     return 0;
543   }
544
```

```
545  /*
546   *  get a command result from the floppy
547   *
548   *  when the controller goes ready waiting for a command
549   *  (instead of sending results), we're done
550   *
551   */
552  static int
553  floppyresult(void)
554  {
555    int i, s;
556    int tries;
557
558    /* get the result of the operation */
559    for(i = 0; i < sizeof(fl.stat); i++){
560            /* wait for status byte */
561            for(tries = 0; ; tries++){
562                    s = inb(Pmsr)&(Ffrom|Fready);
563                    if(s == Fready){
564                            fl.nstat = i;
565                            return fl.nstat;
566                    }
567                    if(s == (Ffrom|Fready))
568                            break;
569                    if(tries > 1000){
570                            DPRINT("floppyresult: %d stats\n", i);
571                            fldump();
572                            fl.confused = 1;
573                            return -1;
574                    }
575                    microdelay(8);  /* for machine independence */
576            }
577            fl.stat[i] = inb(Pfdata);
578    }
579    fl.nstat = sizeof(fl.stat);
580    return fl.nstat;
581  }
582
583  /*
584   *  calculate physical address of a logical byte offset into the disk
585   *
586   *  truncate dp->length if it crosses a track boundary
587   */
588  static void
589  floppypos(FDrive *dp, long off)
590  {
591    int lsec;
592    int ltrack;
593    int end;
594
595    lsec = off/dp->t->bytes;
596    ltrack = lsec/dp->t->sectors;
597    dp->tcyl = ltrack/dp->t->heads;
598    dp->tsec = (lsec % dp->t->sectors) + 1;
599    dp->thead = (lsec/dp->t->sectors) % dp->t->heads;
600
601    /*
602     *  can't read across track boundaries.
603     *  if so, decrement the bytes to be read.
604     */
605    end = (ltrack+1)*dp->t->sectors*dp->t->bytes;
606    if(off+dp->len > end)
607            dp->len = end - off;
608  }
609
610  /*
611   *  get the interrupt cause from the floppy.
612   */
613  static int
614  floppysense(void)
615  {
616    fl.ncmd = 0;
617    fl.cmd[fl.ncmd++] = Fsense;
618    if(floppycmd() < 0)
619            return -1;
620    if(floppyresult() < 2){
621            DPRINT("can't read sense response\n");
622            fldump();
623            fl.confused = 1;
624            return -1;
625    }
626    return 0;
627  }
628
629  static int
630  cmddone(void *)
631  {
632    return fl.ncmd == 0;
633  }
634
635  /*
636   *  Wait for a floppy interrupt.  If none occurs in 5 seconds, we
637   *  may have missed one.  This only happens on some portables which
638   *  do power management behind our backs.  Call the interrupt
639   *  routine to try to clear any conditions.
640   */
641  static void
642  floppywait(void)
643  {
644    tsleep(&fl.r, cmddone, 0, 5000);
645    if(!cmddone(0)){
646            floppyintr(0);
647            fl.confused = 1;
648    }
649  }
650
651  /*
652   *  we've lost the floppy position, go to cylinder 0.
653   */
654  static int
655  floppyrecal(FDrive *dp)
656  {
657    dp->ccyl = -1;
658    dp->cyl = -1;
659
660    fl.ncmd = 0;
661    fl.cmd[fl.ncmd++] = Frecal;
662    fl.cmd[fl.ncmd++] = dp->dev;
663    if(floppycmd() < 0)
664            return -1;
665    floppywait();
666    if(fl.nstat < 2){
667            DPRINT("recalibrate: confused %ux\n", inb(Pmsr));
668            fl.confused = 1;
669            return -1;
670    }
671    if((fl.stat[0] & (Codemask|Seekend)) != Seekend){
672            DPRINT("recalibrate: failed\n");
673            dp->confused = 1;
674            return -1;
675    }
676    dp->cyl = fl.stat[1];
677    if(dp->cyl != 0){
678            DPRINT("recalibrate: wrong cylinder %d\n", dp->cyl);
679            dp->cyl = -1;
680            dp->confused = 1;
```

```
681             return -1;
682     }
683
684     dp->confused = 0;
685     return 0;
686   }
687
688   /*
689    *  if the controller or a specific drive is in a confused state,
690    *  reset it and get back to a known state
691    */
692   static void
693   floppyrevive(void)
694   {
695     FDrive *dp;
696
697     /*
698      *  reset the controller if it's confused
699      */
700     if(fl.confused){
701             DPRINT("floppyrevive in\n");
702             fldump();
703
704             /* reset controller and turn all motors off */
705             splhi();
706             fl.ncmd = 1;
707             fl.cmd[0] = 0;
708             outb(Pdor, 0);
709             delay(10);
710             outb(Pdor, Fintena|Fena);
711             delay(10);
712             spllo();
713             fl.motor = 0;
714             fl.confused = 0;
715             floppywait();
716
717             /* mark all drives in an unknown state */
718             for(dp = fl.d; dp < &fl.d[fl.ndrive]; dp++)
719                     dp->confused = 1;
720
721             /* set rate to a known value */
722             outb(Pdsr, 0);
723             fl.rate = 0;
724
725             DPRINT("floppyrevive out\n");
726             fldump();
727     }
728   }
729
730   /*
731    *  seek to the target cylinder
732    *
733    *        interrupt, no results
734    */
735   static long
736   floppyseek(FDrive *dp, long off)
737   {
738     floppypos(dp, off);
739     if(dp->cyl == dp->tcyl)
740             return dp->tcyl;
741     dp->cyl = -1;
742
743     fl.ncmd = 0;
744     fl.cmd[fl.ncmd++] = Fseek;
745     fl.cmd[fl.ncmd++] = (dp->thead<<2) | dp->dev;
746     fl.cmd[fl.ncmd++] = dp->tcyl * dp->t->steps;
747     if(floppycmd() < 0)
748             return -1;
749     floppywait();
750     if(fl.nstat < 2){
751             DPRINT("seek: confused\n");
752             fl.confused = 1;
753             return -1;
754     }
755     if((fl.stat[0] & (Codemask|Seekend)) != Seekend){
756             DPRINT("seek: failed\n");
757             dp->confused = 1;
758             return -1;
759     }
760
761     dp->cyl = dp->tcyl;
762     return dp->tcyl;
763   }
764
765   /*
766    *  read or write to floppy.  try up to three times.
767    */
768   static long
769   floppyxfer(FDrive *dp, int cmd, void *a, long off, long n)
770   {
771     long offset;
772     int tries;
773
774     if(off >= dp->t->cap)
775             return 0;
776     if(off + n > dp->t->cap)
777             n = dp->t->cap - off;
778
779     /* retry on error (until it gets ridiculous) */
780     tries = 0;
781     while(waserror()){
782             if(tries++ > 20)
783                     nexterror();
784             DPRINT("floppyxfer: retrying\n");
785             /*floppyon(dp);*/
786     }
787
788     dp->len = n;
789     if(floppyseek(dp, off) < 0){
790             DPRINT("xfer: seek failed\n");
791             dp->confused = 1;
792             error(Eio);
793     }
794
795     /*
796      *  set up the dma (dp->len may be trimmed)
797      */
798     if(waserror()){
799             dmaend(DMAchan);
800             nexterror();
801     }
802     dp->len = dmasetup(DMAchan, a, dp->len, cmd==Fread);
803     if(dp->len < 0)
804             error(Eio);
805
806     /*
807      *  start operation
808      */
809     fl.ncmd = 0;
810     fl.cmd[fl.ncmd++] = cmd | (dp->t->heads > 1 ? Fmulti : 0);
811     fl.cmd[fl.ncmd++] = (dp->thead<<2) | dp->dev;
812     fl.cmd[fl.ncmd++] = dp->tcyl;
813     fl.cmd[fl.ncmd++] = dp->thead;
814     fl.cmd[fl.ncmd++] = dp->tsec;
815     fl.cmd[fl.ncmd++] = dp->t->bcode;
816     fl.cmd[fl.ncmd++] = dp->t->sectors;
```

```
817     fl.cmd[fl.ncmd++] = dp->t->gpl;
818     fl.cmd[fl.ncmd++] = 0xFF;
819     if(floppycmd() < 0)
820             error(Eio);
821
822     /* Poll ready bits and transfer data */
823     floppyexec((char*)a, dp->len, cmd==Fread);
824
825     /*
826      *  give bus to DMA, floppyintr() will read result
827      */
828     floppywait();
829     dmaend(DMAchan);
830     poperror();
831
832     /*
833      *  check for errors
834      */
835     if(fl.nstat < 7){
836             DPRINT("xfer: confused\n");
837             fl.confused = 1;
838             error(Eio);
839     }
840     if((fl.stat[0] & Codemask)!=0 || fl.stat[1] || fl.stat[2]){
841             DPRINT("xfer: failed %ux %ux %ux\n", fl.stat[0],
842                     fl.stat[1], fl.stat[2]);
843             DPRINT("offset %lud len %ld\n", off, dp->len);
844             if((fl.stat[0]&Codemask)==Cmdexec && fl.stat[1]==Overrun){
845                     DPRINT("DMA overrun: retry\n");
846             } else
847                     dp->confused = 1;
848             error(Eio);
849     }
850
851     /*
852      *  check for correct cylinder
853      */
854     offset = fl.stat[3] * dp->t->heads + fl.stat[4];
855     offset = offset*dp->t->sectors + fl.stat[5] - 1;
856     offset = offset * c2b[fl.stat[6]];
857     if(offset != off+dp->len){
858             DPRINT("xfer: ends on wrong cyl\n");
859             dp->confused = 1;
860             error(Eio);
861     }
862     poperror();
863
864     dp->lasttouched = m->ticks;
865     return dp->len;
866   }
867
868   /*
869    *  format a track
870    */
871   static void
872   floppyformat(FDrive *dp, char *params)
873   {
874     int cyl, h, sec;
875     ulong track;
876     uchar *buf, *bp;
877     FType *t;
878     char *f[3];
879
880     /*
881      *  set the type
882      */
883     if(getfields(params, f, 3, 1, " ") > 1){
884             for(t = floppytype; t < &floppytype[nelem(floppytype)]; t++){
885                     if(strcmp(f[1], t->name)==0 && t->dt==dp->dt){
886                             dp->t = t;
887                             floppydir[NFDIR*dp->dev].length = dp->t->cap;
888                             break;
889                     }
890             }
891             if(t >= &floppytype[nelem(floppytype)])
892                     error(Ebadarg);
893     } else {
894             floppysetdef(dp);
895             t = dp->t;
896     }
897
898     /*
899      *  buffer for per track info
900      */
901     buf = smalloc(t->sectors*4);
902     if(waserror()){
903             free(buf);
904             nexterror();
905     }
906
907     /* force a recalibrate to cylinder 0 */
908     dp->confused = 1;
909     if(!waserror()){
910             floppyon(dp);
911             poperror();
912     }
913
914     /*
915      *  format a track at time
916      */
917     for(track = 0; track < t->tracks*t->heads; track++){
918             cyl = track/t->heads;
919             h = track % t->heads;
920
921             /*
922              *  seek to track, ignore errors
923              */
924             floppyseek(dp, track*t->tsize);
925             dp->cyl = cyl;
926             dp->confused = 0;
927
928             /*
929              *  set up the dma (dp->len may be trimmed)
930              */
931             bp = buf;
932             for(sec = 1; sec <= t->sectors; sec++){
933                     *bp++ = cyl;
934                     *bp++ = h;
935                     *bp++ = sec;
936                     *bp++ = t->bcode;
937             }
938             if(waserror()){
939                     dmaend(DMAchan);
940                     nexterror();
941             }
942             if(dmasetup(DMAchan, buf, bp-buf, 0) < 0)
943                     error(Eio);
944
945             /*
946              *  start operation
947              */
948             fl.ncmd = 0;
949             fl.cmd[fl.ncmd++] = Fformat;
950             fl.cmd[fl.ncmd++] = (h<<2) | dp->dev;
951             fl.cmd[fl.ncmd++] = t->bcode;
952             fl.cmd[fl.ncmd++] = t->sectors;
```

```
953             fl.cmd[fl.ncmd++] = t->fgpl;
954             fl.cmd[fl.ncmd++] = 0x5a;
955             if(floppycmd() < 0)
956                     error(Eio);
957
958             /* Poll ready bits and transfer data */
959             floppyexec((char *)buf, bp-buf, 0);
960
961             /*
962              *  give bus to DMA, floppyintr() will read result
963              */
964             floppywait();
965             dmaend(DMAchan);
966             poperror();
967
968             /*
969              *  check for errors
970              */
971             if(fl.nstat < 7){
972                     DPRINT("format: confused\n");
973                     fl.confused = 1;
974                     error(Eio);
975             }
976             if((fl.stat[0]&Codemask)!=0 || fl.stat[1]|| fl.stat[2]){
977                     DPRINT("format: failed %ux %ux %ux\n",
978                             fl.stat[0], fl.stat[1], fl.stat[2]);
979                     dp->confused = 1;
980                     error(Eio);
981             }
982     }
983     free(buf);
984     dp->confused = 1;
985     poperror();
986   }
987
988   static void
989   floppyintr(Ureg *)
990   {
991     switch(fl.cmd[0]&~Fmulti){
992     case Fread:
993     case Fwrite:
994     case Fformat:
995     case Fdumpreg:
996             floppyresult();
997             break;
998     case Fseek:
999     case Frecal:
1000    default:
1001            floppysense();  /* to clear interrupt */
1002            break;
1003    }
1004    fl.ncmd = 0;
1005    wakeup(&fl.r);
1006  }
1007
1008  Dev floppydevtab = {
1009    'f',
1010    "floppy",
1011
1012    floppyreset,
1013    devinit,
1014    floppyattach,
1015    devclone,
1016    floppywalk,
1017    floppystat,
1018    floppyopen,
1019    devcreate,
1020    floppyclose,
1021    floppyread,
1022    devbread,
1023    floppywrite,
1024    devbwrite,
1025    devremove,
1026    devwstat,
1027  };
```

```
#include  "u.h"
#include  "../port/lib.h"
#include  "mem.h"
#include  "dat.h"
#include  "fns.h"
#include  "io.h"
#include  "../port/error.h"

/* Centronix parallel (printer) port */

/* base addresses */
static int lptbase[] = {
  0x378,  /* lpt1 */
  0x3bc,  /* lpt2 */
  0x278   /* lpt3 (sic) */
};
#define NDEV      nelem(lptbase)
static int lptallocd[NDEV];

/* offsets, and bits in the registers */
enum
{
  /* data latch register */
  Qdlr=           0x0,
  /* printer status register */
  Qpsr=           0x1,
  Fnotbusy=       0x80,
  Fack=           0x40,
  Fpe=            0x20,
  Fselect=        0x10,
  Fnoerror=       0x08,
  /* printer control register */
  Qpcr=           0x2,
  Fie=            0x10,
  Fselectin=      0x08,
  Finitbar=       0x04,
  Faf=            0x02,
  Fstrobe=        0x01,
  /* fake 'data register' */
  Qdata=          0x3,
};

static int        lptready(void*);
static void       outch(int, int);
static void       lptintr(Ureg*, void*);

static Rendez     lptrendez;

Dirtab lptdir[]={
  "dlr",          {Qdlr},         1,              0666,
  "psr",          {Qpsr},         5,              0444,
  "pcr",          {Qpcr},         0,              0222,
  "data",         {Qdata},        0,              0222,
};

static int
lptgen(Chan *c, Dirtab *tab, int ntab, int i, Dir *dp)
{
  Qid qid;
  char name[NAMELEN];

  if(i == DEVDOTDOT){
          sprint(name, "#L%lud", c->dev+1);
          devdir(c, (Qid){CHDIR, 0}, name, 0, eve, 0555, dp);
          return 1;
  }
  if(tab==0 || i>=ntab)
          return -1;
  tab += i;
  qid = tab->qid;
  if(qid.path < Qdata)
          qid.path += lptbase[c->dev];
  qid.vers = c->dev;
  sprint(name, "lpt%lud%s", c->dev+1, tab->name);
  devdir(c, qid, name, tab->length, eve, tab->perm, dp);
  return 1;
}

static Chan*
lptattach(char *spec)
{
  Chan *c;
  int i  = (spec && *spec) ? strtol(spec, 0, 0) : 1;
  char name[5];
  static int set;

  if(!set){
          outb(lptbase[i-1]+Qpcr, 0);     /* turn off interrupts */
          set = 1;
          intrenable(IrqLPT, lptintr, 0, BUSUNKNOWN, "lpt");
  }
  if(i < 1 || i > NDEV)
          error(Ebadarg);
  if(lptallocd[i-1] == 0){
          sprint(name, "lpt%d", i-1);
          if(ioalloc(lptbase[i-1], 3, 0, name) < 0)
                  error("lpt port space in use");
          lptallocd[i-1] = 1;
  }
  c = devattach('L', spec);
  c->dev = i-1;
  return c;
}

static int
lptwalk(Chan *c, char *name)
{
  return devwalk(c, name, lptdir, nelem(lptdir), lptgen);
}

static void
lptstat(Chan *c, char *dp)
{
  devstat(c, dp, lptdir, nelem(lptdir), lptgen);
}

static Chan*
lptopen(Chan *c, int omode)
{
  return devopen(c, omode, lptdir, nelem(lptdir), lptgen);
}

static void
lptclose(Chan *)
{
}

static long
lptread(Chan *c, void *a, long n, vlong)
{
  char str[16];
  int size;
  ulong o;

  if(c->qid.path == CHDIR)
          return devdirread(c, a, n, lptdir, nelem(lptdir), lptgen);
```

```
137     size = sprint(str, "0x%2.2ux\n", inb(c->qid.path));
138     o = c->offset;
139     if(o >= size)
140             return 0;
141     if(o+n > size)
142             n = size-c->offset;
143     memmove(a, str+o, n);
144     return n;
145   }
146
147   static long
148   lptwrite(Chan *c, void *a, long n, vlong)
149   {
150     char str[16], *p;
151     long base, k;
152
153     if(n <= 0)
154             return 0;
155     if(c->qid.path != Qdata){
156             if(n > sizeof str-1)
157                     n = sizeof str-1;
158             memmove(str, a, n);
159             str[n] = 0;
160             outb(c->qid.path, strtoul(str, 0, 0));
161             return n;
162     }
163     p = a;
164     k = n;
165     base = lptbase[c->dev];
166     if(waserror()){
167             outb(base+Qpcr, Finitbar);
168             nexterror();
169     }
170     while(--k >= 0)
171             outch(base, *p++);
172     poperror();
173     return n;
174   }
175
176   static void
177   outch(int base, int c)
178   {
179     int status, tries;
180
181     for(tries = 0;; tries++){
182             status = inb(base+Qpsr);
183             if(!(status & Fselect) || !(status & Fnoerror))
184                     error(Eio);
185             if(status & Fnotbusy)
186                     break;
187             if(tries > 1000){
188                     outb(base+Qpcr, Finitbar|Fie);
189                     tsleep(&lptrendez, lptready, (void *)base, MS2HZ);
190             }
191     }
192     outb(base+Qdlr, c);
193     outb(base+Qpcr, Finitbar|Fstrobe);
194     outb(base+Qpcr, Finitbar);
195   }
196
197   static int
198   lptready(void *base)
199   {
200     return inb((int)base+Qpsr)&Fnotbusy;
201   }
202
203   static void
204   lptintr(Ureg *, void *)
205   {
206     wakeup(&lptrendez);
207   }
208
209   Dev lptdevtab = {
210     'L',
211     "lpt",
212
213     devreset,
214     devinit,
215     lptattach,
216     devclone,
217     lptwalk,
218     lptstat,
219     lptopen,
220     devcreate,
221     lptclose,
222     lptread,
223     devbread,
224     lptwrite,
225     devbwrite,
226     devremove,
227     devwstat,
228   };
```

```
/*
 * VGA controller
 */
#include "u.h"
#include "../port/lib.h"
#include "mem.h"
#include "dat.h"
#include "fns.h"
#include "../port/error.h"

#define   Image   IMAGE
#include <draw.h>
#include <memdraw.h>
#include <cursor.h>
#include "screen.h"

enum {
  Qdir,
  Qvgactl,
};

static Dirtab vgadir[] = {
  "vgactl",       { Qvgactl, 0 },         0,      0660,
};

static void
vgareset(void)
{
  /* reserve the 'standard' vga registers */
  if(ioalloc(0x2b0, 0x2df-0x2b0+1, 0, "vga") < 0)
          panic("vga ports already allocated");
  if(ioalloc(0x3c0, 0x3da-0x3c0+1, 0, "vga") < 0)
          panic("vga ports already allocated");
  conf.monitor = 1;
}

static Chan*
vgaattach(char* spec)
{
  if(*spec && strcmp(spec, "0"))
          error(Eio);
  return devattach('v', spec);
}

int
vgawalk(Chan* c, char* name)
{
  return devwalk(c, name, vgadir, nelem(vgadir), devgen);
}

static void
vgastat(Chan* c, char* dp)
{
  devstat(c, dp, vgadir, nelem(vgadir), devgen);
}

static Chan*
vgaopen(Chan* c, int omode)
{
  return devopen(c, omode, vgadir, nelem(vgadir), devgen);
}

static void
vgaclose(Chan*)
{
}

static void
checkport(int start, int end)
{
  /* standard vga regs are OK */
  if(start >= 0x2b0 && end <= 0x2df+1)
          return;
  if(start >= 0x3c0 && end <= 0x3da+1)
          return;

  if(iounused(start, end))
          return;
  error(Eperm);
}

static long
vgaread(Chan* c, void* a, long n, vlong off)
{
  int len;
  char *p, *s;
  VGAscr *scr;
  ulong offset = off;
  char chbuf[30];

  switch(c->qid.path & ~CHDIR){

  case Qdir:
          return devdirread(c, a, n, vgadir, nelem(vgadir), devgen);

  case Qvgactl:
          scr = &vgascreen[0];

          p = malloc(READSTR);
          if(waserror()){
                  free(p);
                  nexterror();
          }

          len = 0;

          if(scr->dev)
                  s = scr->dev->name;
          else
                  s = "cga";
          len += snprint(p+len, READSTR-len, "type %s\n", s);

          if(scr->gscreen) {
                  len += snprint(p+len, READSTR-len, "size %dx%dx%d %s\n",
                          scr->gscreen->r.max.x, scr->gscreen->r.max.y,
                          scr->gscreen->depth, chantostr(chbuf, scr->gscreen->chan));

                  if(Dx(scr->gscreen->r) != Dx(physgscreenr)
                  || Dy(scr->gscreen->r) != Dy(physgscreenr))
                          len += snprint(p+len, READSTR-len, "actualsize %dx%d\n",
                                  physgscreenr.max.x, physgscreenr.max.y);
          }

          len += snprint(p+len, READSTR-len, "blanktime %lud\n", blanktime);
          len += snprint(p+len, READSTR-len, "hwaccel %s\n", hwaccel ? "on" : "off");
          len += snprint(p+len, READSTR-len, "hwblank %s\n", hwblank ? "on" : "off");
          snprint(p+len, READSTR-len, "addr 0x%lux\n", scr->aperture);
          n = readstr(offset, a, n, p);
          poperror();
          free(p);

          return n;

  default:
          error(Egreg);
          break;
```

```
137     }
138
139     return 0;
140   }
141
142   static char Ebusy[] = "vga already configured";
143
144   static void
145   vgactl(char* a)
146   {
147     int align, i, n, size, x, y, z;
148     char *chanstr, *field[6], *p;
149     ulong chan;
150     VGAscr *scr;
151     extern VGAdev *vgadev[];
152     extern VGAcur *vgacur[];
153     Rectangle r;
154
155     n = getfields(a, field, nelem(field), 1, " \t\n");
156     if(n < 1)
157             error(Ebadarg);
158
159     scr = &vgascreen[0];
160     if(strcmp(field[0], "hwgc") == 0){
161             if(n < 2)
162                     error(Ebadarg);
163
164             if(strcmp(field[1], "off") == 0){
165                     lock(&cursor);
166                     if(scr->cur){
167                             if(scr->cur->disable)
168                                     scr->cur->disable(scr);
169                             scr->cur = nil;
170                     }
171                     unlock(&cursor);
172                     return;
173             }
174
175             for(i = 0; vgacur[i]; i++){
176                     if(strcmp(field[1], vgacur[i]->name))
177                             continue;
178                     lock(&cursor);
179                     if(scr->cur && scr->cur->disable)
180                             scr->cur->disable(scr);
181                     scr->cur = vgacur[i];
182                     if(scr->cur->enable)
183                             scr->cur->enable(scr);
184                     unlock(&cursor);
185                     return;
186             }
187     }
188     else if(strcmp(field[0], "type") == 0){
189             if(n < 2)
190                     error(Ebadarg);
191
192             for(i = 0; vgadev[i]; i++){
193                     if(strcmp(field[1], vgadev[i]->name))
194                             continue;
195                     if(scr->dev && scr->dev->disable)
196                             scr->dev->disable(scr);
197                     scr->dev = vgadev[i];
198                     if(scr->dev->enable)
199                             scr->dev->enable(scr);
200                     return;
201             }
202     }
203     else if(strcmp(field[0], "size") == 0){
204             if(n < 3)
205                     error(Ebadarg);
206             if(drawhasclients())
207                     error(Ebusy);
208
209             x = strtoul(field[1], &p, 0);
210             if(x == 0 || x > 2048)
211                     error(Ebadarg);
212             if(*p)
213                     p++;
214
215             y = strtoul(p, &p, 0);
216             if(y == 0 || y > 2048)
217                     error(Ebadarg);
218             if(*p)
219                     p++;
220
221             z = strtoul(p, &p, 0);
222
223             chanstr = field[2];
224             if((chan = strtochan(chanstr)) == 0)
225                     error("bad channel");
226
227             if(chantodepth(chan) != z)
228                     error("depth, channel do not match");
229
230             cursoroff(1);
231             deletescreenimage();
232             if(screensize(x, y, z, chan))
233                     error(Egreg);
234             vgascreenwin(scr);
235             cursoron(1);
236             return;
237     }
238     else if(strcmp(field[0], "actualsize") == 0){
239             if(scr->gscreen == nil)
240                     error("set the screen size first");
241
242             if(n < 2)
243                     error(Ebadarg);
244             x = strtoul(field[1], &p, 0);
245             if(x == 0 || x > 2048)
246                     error(Ebadarg);
247             if(*p)
248                     p++;
249
250             y = strtoul(p, nil, 0);
251             if(y == 0 || y > 2048)
252                     error(Ebadarg);
253
254             if(x > scr->gscreen->r.max.x || y > scr->gscreen->r.max.y)
255                     error("physical screen bigger than virtual");
256
257             r = Rect(0,0,x,y);
258             if(!eqrect(r, scr->gscreen->r)){
259                     if(scr->cur == nil || scr->cur->doespanning == 0)
260                             error("virtual screen not supported");
261             }
262
263             physgscreenr = r;
264             return;
265     }
266     else if(strcmp(field[0], "palettedepth") == 0){
267             if(n < 2)
268                     error(Ebadarg);
269
270             x = strtoul(field[1], &p, 0);
271             if(x != 8 && x != 6)
272                     error(Ebadarg);
```

```
273
274             scr->palettedepth = x;
275             return;
276     }
277     else if(strcmp(field[0], "drawinit") == 0){
278             if(scr && scr->dev && scr->dev->drawinit)
279                     scr->dev->drawinit(scr);
280             return;
281     }
282     else if(strcmp(field[0], "linear") == 0){
283             if(n < 2)
284                     error(Ebadarg);
285
286             size = strtoul(field[1], 0, 0);
287             if(n < 3)
288                     align = 0;
289             else
290                     align = strtoul(field[2], 0, 0);
291             if(screenaperture(size, align))
292                     error("not enough free address space");
293             return;
294     }
295   /*        else if(strcmp(field[0], "memset") == 0){
296             if(n < 4)
297                     error(Ebadarg);
298             memset((void*)strtoul(field[1], 0, 0), atoi(field[2]), atoi(field[3]));
299             return;
300     }
301   */
302     else if(strcmp(field[0], "blank") == 0){
303             if(n < 1)
304                     error(Ebadarg);
305             drawblankscreen(1);
306             return;
307     }
308     else if(strcmp(field[0], "blanktime") == 0){
309             if(n < 2)
310                     error(Ebadarg);
311             blanktime = strtoul(field[1], 0, 0);
312             return;
313     }
314     else if(strcmp(field[0], "hwaccel") == 0){
315             if(n < 2)
316                     error(Ebadarg);
317             if(strcmp(field[1], "on") == 0)
318                     hwaccel = 1;
319             else if(strcmp(field[1], "off") == 0)
320                     hwaccel = 0;
321             else
322                     error(Ebadarg);
323             return;
324     }
325     else if(strcmp(field[0], "hwblank") == 0){
326             if(n < 2)
327                     error(Ebadarg);
328             if(strcmp(field[1], "on") == 0)
329                     hwblank = 1;
330             else if(strcmp(field[1], "off") == 0)
331                     hwblank = 0;
332             else
333                     error(Ebadarg);
334             return;
335     }
336
337     error(Ebadarg);
338   }
339
340   static long
341   vgawrite(Chan* c, void* a, long n, vlong off)
342   {
343     char *p;
344     ulong offset = off;
345
346     switch(c->qid.path & ~CHDIR){
347
348     case Qdir:
349             error(Eperm);
350
351     case Qvgactl:
352             if(offset || n >= READSTR)
353                     error(Ebadarg);
354             p = malloc(READSTR);
355             if(waserror()){
356                     free(p);
357                     nexterror();
358             }
359             memmove(p, a, n);
360             p[n] = 0;
361             vgactl(p);
362             poperror();
363             free(p);
364             return n;
365
366     default:
367             error(Egreg);
368             break;
369     }
370
371     return 0;
372   }
373
374   Dev vgadevtab = {
375     'v',
376     "vga",
377
378     vgareset,
379     devinit,
380     vgaattach,
381     devclone,
382     vgawalk,
383     vgastat,
384     vgaopen,
385     devcreate,
386     vgaclose,
387     vgaread,
388     devbread,
389     vgawrite,
390     devbwrite,
391     devremove,
392     devwstat,
393   };
```

```
#include "../port/portfns.h"

void      aamloop(int);
void      addconf(char*, char*);
//void    addscsilink(char*, Scsiio (*)(int, ISAConf*));
void      archinit(void);
void      bootargs(ulong);
int       cistrcmp(char*, char*);
int       cistrncmp(char*, char*, int);
#define   clearmmucache()                         /* x86 doesn't have one */
void      clockintr(Ureg*, void*);
void      (*coherence)(void);
void      cpuid(char*, int*, int*);
int       cpuidentify(void);
void      cpuidprint(void);
void      delay(int);
int       dmacount(int);
int       dmadone(int);
void      dmaend(int);
int       dmainit(int, int);
long      dmasetup(int, void*, long, int);
#define   evenaddr(x)                             /* x86 doesn't care */
void      fpenv(FPsave*);
void      fpinit(void);
void      fpoff(void);
void      fprestore(FPsave*);
void      fpsave(FPsave*);
ulong     fpstatus(void);
ulong     getcr0(void);
ulong     getcr2(void);
ulong     getcr3(void);
ulong     getcr4(void);
char*     getconf(char*);
int       i8042auxcmd(int);
void      i8042auxenable(void (*)(int, int));
void      i8042reset(void);
void      i8253init(int, int);
void      i8253enable(void);
uvlong    i8253read(uvlong*);
void      i8259init(void);
int       i8259enable(Vctl*);
void      idle(void);
#define   idlehands()                     /* nothing to do in the runproc */
int       inb(int);
void      insb(int, void*, int);
ushort    ins(int);
void      inss(int, void*, int);
ulong     inl(int);
void      insl(int, void*, int);
void      intrenable(int, void (*)(Ureg*, void*), void*, int, char*);
void      iofree(int);
void      ioinit(void);
int       iounused(int, int);
int       ioalloc(int, int, int, char*);
int       iprint(char*, ...);
int       irqallocread(char*, long, vlong);
int       isaconfig(char*, int, ISAConf*);
void      kbdinit(void);
void      lgdt(ushort[3]);
void      lidt(ushort[3]);
void      links(void);
void      ltr(ulong);
void      mathinit(void);
void      meminit(ulong);
#define mmuflushtlb(pdb) putcr3(pdb)
void      mmuinit(void);
ulong     mmukmap(ulong, ulong, int);
int       mmukmapsync(ulong);
#define   mmunewpage(x)
ulong*    mmuwalk(ulong*, ulong, int, int);
void      ns16552install(void);
void      ns16552special(int, int, Queue**, Queue**, int (*)(Queue*, int));
uchar     nvramread(int);
void      nvramwrite(int, uchar);
void      outb(int, int);
void      outsb(int, void*, int);
void      outs(int, ushort);
void      outss(int, void*, int);
void      outl(int, ulong);
void      outsl(int, void*, int);
int       pcicfgr8(Pcidev*, int);
int       pcicfgr16(Pcidev*, int);
int       pcicfgr32(Pcidev*, int);
void      pcicfgw8(Pcidev*, int, int);
void      pcicfgw16(Pcidev*, int, int);
void      pcicfgw32(Pcidev*, int, int);
void      pcihinv(Pcidev*);
Pcidev* pcimatch(Pcidev*, int, int);
Pcidev* pcimatchtbdf(int);
void      pcireset(void);
void      pcisetbme(Pcidev*);
int       pcmcistuple(int, int, void*, int);
PCMmap*   pcmmap(int, ulong, int, int);
int       pcmspecial(char*, ISAConf*);
void      pcmspecialclose(int);
void      pcmunmap(int, PCMmap*);
void      printcpufreq(void);
#define   procrestore(p)
void      procsave(Proc*);
void      procsetup(Proc*);
void      putcr3(ulong);
void      putcr4(ulong);
void      rdmsr(int, vlong*);
void      screeninit(void);
int       screenprint(char*, ...);                /* debugging */
void      (*screenputs)(char*, int);
void      touser(void*);
void      trapenable(int, void (*)(Ureg*, void*), void*, char*);
void      trapinit(void);
int       tas(void*);
ulong     umbmalloc(ulong, int, int);
void      umbfree(ulong, int);
ulong     umbrwmalloc(ulong, int, int);
void      umbrwfree(ulong, int);
ulong     upamalloc(ulong, int, int);
void      upafree(ulong, int);
void      vectortable(void);
void      wrmsr(int, vlong);
void      wbflush(void);
int       xchgw(ushort*, int);
ulong     TK2MS(ulong);                   /* ticks to milliseconds */

#define   waserror()      (up->nerrlab++, setlabel(&up->errlab[up->nerrlab-1]))
#define KADDR(a)  ((void*)((ulong)(a)|KZERO))
#define PADDR(a)  ((ulong)(a)&~KZERO)

#define   dcflush(a, b)
```

```
#include "u.h"
#include "../port/lib.h"
#include "mem.h"
#include "dat.h"
#include "fns.h"
#include "io.h"

/*
 *  8253 timer
 */
enum
{
  T0cntr= 0x40,           /* counter ports */
  T1cntr= 0x41,           /* ... */
  T2cntr= 0x42,           /* ... */
  Tmode=  0x43,           /* mode port */

  /* commands */
  Latch0= 0x00,           /* latch counter 0's value */
  Load0=  0x30,           /* load counter 0 with 2 bytes */

  /* modes */
  Square= 0x36,           /* periodic square wave */
  Trigger= 0x30,          /* interrupt on terminal count */

  Freq=   1193182,        /* Real clock frequency */
};

void
i8253init(int aalcycles, int havecycleclock)
{
  int cpufreq, loops, incr, x, y;
  vlong a, b;
  static int initialised;

  if(initialised == 0){
          initialised = 1;
          ioalloc(T0cntr, 4, 0, "i8253");
          /*
           *  set clock for 1/HZ seconds
           */
          outb(Tmode, Load0|Square);
          outb(T0cntr, (Freq/HZ));        /* low byte */
          outb(T0cntr, (Freq/HZ)>>8);     /* high byte */

          /*
           * Introduce a little delay to make sure the count is
           * latched and the timer is counting down; with a fast
           * enough processor this may not be the case.
           * The i8254 (which this probably is) has a read-back
           * command which can be used to make sure the counting
           * register has been written into the counting element.
           */
          x = (Freq/HZ);
          for(loops = 0; loops < 100000 && x >= (Freq/HZ); loops++){
                  outb(Tmode, Latch0);
                  x = inb(T0cntr);
                  x |= inb(T0cntr)<<8;
          }
  }

  /* find biggest loop that doesn't wrap */
  incr = 16000000/(aalcycles*HZ*2);
  x = 2000;
  for(loops = incr; loops < 64*1024; loops += incr) {

          /*
           *  measure time for the loop
           *
           *                      MOVL    loops,CX
           *      aaml1:          AAM
           *                      LOOP    aaml1
           *
           *  the time for the loop should be independent of external
           *  cache and memory system since it fits in the execution
           *  prefetch buffer.
           *
           */
          outb(Tmode, Latch0);
          if(havecycleclock)
                  rdmsr(0x10, &a);
          x = inb(T0cntr);
          x |= inb(T0cntr)<<8;
          aamloop(loops);
          outb(Tmode, Latch0);
          if(havecycleclock)
                  rdmsr(0x10, &b);
          y = inb(T0cntr);
          y |= inb(T0cntr)<<8;
          x -= y;

          if(x < 0)
                  x += Freq/HZ;

          if(x > Freq/(3*HZ))
                  break;
  }

  /*
   *  figure out clock frequency and a loop multiplier for delay().
   *  n.b. counter goes up by 2*Freq
   */
  cpufreq = loops*((aalcycles*2*Freq)/x);
  m->loopconst = (cpufreq/1000)/aalcycles;        /* AAM+LOOP's for 1 ms */

  if(havecycleclock){

          /* counter goes up by 2*Freq */
          b = (b-a)<<1;
          b *= Freq;
          b /= x;

          /*
           *  round to the nearest megahz
           */
          m->cpumhz = (b+500000)/1000000L;
          m->cpuhz = b;
  } else {
          /*
           *  add in possible 0.5% error and convert to MHz
           */
          m->cpumhz = (cpufreq + cpufreq/200)/1000000;
          m->cpuhz = cpufreq;
  }
}

void
i8253enable(void)
{
  intrenable(IrqCLOCK, clockintr, 0, BUSUNKNOWN, "clock");
}

/*
 *  return time elapsed since clock start in
 *  100 times hz
 */
```

```
137   uvlong
138   i8253read(uvlong *hz)
139   {
140     if(hz)
141             *hz = HZ*100;
142     return m->ticks*100;
143   }
```

```
#include "u.h"
#include "../port/lib.h"
#include "mem.h"
#include "dat.h"
#include "fns.h"
#include "io.h"

/*
 *  8259 interrupt controllers
 */
enum
{
  Int0ctl=        0x20,           /* control port (ICW1, OCW2, OCW3) */
  Int0aux=        0x21,           /* everything else (ICW2, ICW3, ICW4, OCW1) */
  Int1ctl=        0xA0,           /* control port */
  Int1aux=        0xA1,           /* everything else (ICW2, ICW3, ICW4, OCW1) */

  Icw1=           0x10,           /* select bit in ctl register */
  Ocw2=           0x00,
  Ocw3=           0x08,

  EOI=            0x20,           /* non-specific end of interrupt */

  Elcr1=          0x4D0,          /* Edge/Level Triggered Register */
  Elcr2=          0x4D1,
};

static Lock i8259lock;
static int i8259mask = 0xFFFF;          /* disabled interrupts */
int i8259elcr;                          /* mask of level-triggered interrupts */

void
i8259init(void)
{
  int x;

  ioalloc(Int0ctl, 2, 0, "i8259.0");
  ioalloc(Int1ctl, 2, 0, "i8259.1");
  ilock(&i8259lock);

  /*
   *  Set up the first 8259 interrupt processor.
   *  Make 8259 interrupts start at CPU vector VectorPIC.
   *  Set the 8259 as master with edge triggered
   *  input with fully nested interrupts.
   */
  outb(Int0ctl, (1<<4)|(0<<3)|(1<<0));  /* ICW1 - master, edge triggered,
                                           ICW4 will be sent */
  outb(Int0aux, VectorPIC);             /* ICW2 - interrupt vector offset */
  outb(Int0aux, 0x04);                  /* ICW3 - have slave on level 2 */
  outb(Int0aux, 0x01);                  /* ICW4 - 8086 mode, not buffered */

  /*
   *  Set up the second 8259 interrupt processor.
   *  Make 8259 interrupts start at CPU vector VectorPIC+8.
   *  Set the 8259 as slave with edge triggered
   *  input with fully nested interrupts.
   */
  outb(Int1ctl, (1<<4)|(0<<3)|(1<<0));  /* ICW1 - master, edge triggered,
                                           ICW4 will be sent */
  outb(Int1aux, VectorPIC+8);           /* ICW2 - interrupt vector offset */
  outb(Int1aux, 0x02);                  /* ICW3 - I am a slave on level 2 */
  outb(Int1aux, 0x01);                  /* ICW4 - 8086 mode, not buffered */
  outb(Int1aux, (i8259mask>>8) & 0xFF);

  /*
   *  pass #2 8259 interrupts to #1
   */
  i8259mask &= ~0x04;
  outb(Int0aux, i8259mask & 0xFF);

  /*
   * Set Ocw3 to return the ISR when ctl read.
   * After initialisation status read is set to IRR.
   * Read IRR first to possibly deassert an outstanding
   * interrupt.
   */
  inb(Int0ctl);
  outb(Int0ctl, Ocw3|0x03);
  inb(Int1ctl);
  outb(Int1ctl, Ocw3|0x03);

  /*
   * Check for Edge/Level register.
   * This check may not work for all chipsets.
   * First try a non-intrusive test - the bits for
   * IRQs 13, 8, 2, 1 and 0 must be edge (0). If
   * that's OK try a R/W test.
   */
  x = (inb(Elcr2)<<8)|inb(Elcr1);
  if(!(x & 0x2107)){
          outb(Elcr1, 0);
          if(inb(Elcr1) == 0){
                  outb(Elcr1, 0x20);
                  if(inb(Elcr1) == 0x20)
                          i8259elcr = x;
                  outb(Elcr1, x & 0xFF);
                  //print("ELCR: %4.4uX\n", i8259elcr);
          }
  }
  iunlock(&i8259lock);
}

int
i8259isr(int vno)
{
  int irq, isr;

  if(vno < VectorPIC || vno > VectorPIC+MaxIrqPIC)
          return 0;
  irq = vno-VectorPIC;

  /*
   *  tell the 8259 that we're done with the
   *  highest level interrupt (interrupts are still
   *  off at this point)
   */
  ilock(&i8259lock);
  isr = inb(Int0ctl);
  outb(Int0ctl, EOI);
  if(irq >= 8){
          isr |= inb(Int1ctl)<<8;
          outb(Int1ctl, EOI);
  }
  iunlock(&i8259lock);

  return isr & (1<<irq);
}

int
i8259enable(Vctl* v)
{
  int irq, irqbit;

  /*
   * Given an IRQ, enable the corresponding interrupt in the i8259
```

```
137      * and return the vector to be used. The i8259 is set to use a fixed
138      * range of vectors starting at VectorPIC.
139      */
140     irq = v->irq;
141     if(irq < 0 || irq > MaxIrqPIC){
142             print("i8259enable: irq %d out of range\n", irq);
143             return -1;
144     }
145     irqbit = 1<<irq;
146
147     ilock(&i8259lock);
148     if(!(i8259mask & irqbit) && !(i8259elcr & irqbit)){
149             print("i8259enable: irq %d shared but not level\n", irq);
150             iunlock(&i8259lock);
151             return -1;
152     }
153     i8259mask &= ~irqbit;
154     if(irq < 8)
155             outb(Int0aux, i8259mask & 0xFF);
156     else
157             outb(Int1aux, (i8259mask>>8) & 0xFF);
158
159     if(i8259elcr & irqbit)
160             v->eoi = i8259isr;
161     else
162             v->isr = i8259isr;
163     iunlock(&i8259lock);
164
165     return VectorPIC+irq;
166   }
```

```
#define X86STEPPING(x)    ((x) & 0x0F)
#define X86MODEL(x)       (((x)>>4) & 0x0F)
#define X86FAMILY(x)      (((x)>>8) & 0x0F)

enum {
  VectorNMI       = 2,            /* non-maskable interrupt */
  VectorBPT       = 3,            /* breakpoint */
  VectorUD        = 6,            /* invalid opcode exception */
  VectorCNA       = 7,            /* coprocessor not available */
  VectorCSO       = 9,            /* coprocessor segment overrun */
  VectorPF        = 14,           /* page fault */
  VectorCERR      = 16,           /* coprocessor error */

  VectorPIC       = 32,           /* external i8259 interrupts */
  IrqCLOCK        = 0,
  IrqKBD          = 1,
  IrqUART1        = 3,
  IrqUART0        = 4,
  IrqPCMCIA       = 5,
  IrqFLOPPY       = 6,
  IrqLPT          = 7,
  IrqIRQ7         = 7,
  IrqAUX          = 12,           /* PS/2 port */
  IrqIRQ13        = 13,           /* coprocessor on 386 */
  IrqATA0         = 14,
  IrqATA1         = 15,
  MaxIrqPIC       = 15,

  VectorLAPIC     = VectorPIC+16, /* local APIC interrupts */
  IrqLINT0        = 16,           /* LINT[01] must be offsets 0 and 1 */
  IrqLINT1        = 17,
  IrqTIMER        = 18,
  IrqERROR        = 19,
  IrqPCINT        = 20,
  IrqSPURIOUS     = 31,           /* must have bits [3-0] == 0x0F */
  MaxIrqLAPIC     = 31,

  VectorSYSCALL   = 64,

  VectorAPIC      = 65,           /* external APIC interrupts */
  MaxVectorAPIC   = 255,
};

typedef struct Vctl {
  Vctl*   next;                   /* handlers on this vector */

  char    name[NAMELEN];          /* of driver */
  int     isintr;                 /* interrupt or fault/trap */
  int     irq;
  int     tbdf;
  int     (*isr)(int);            /* get isr bit for this irq */
  int     (*eoi)(int);            /* eoi */

  void    (*f)(Ureg*, void*);     /* handler to call */
  void*   a;                      /* argument to call it with */
} Vctl;

enum {
  BusCBUS         = 0,            /* Corollary CBUS */
  BusCBUSII,                      /* Corollary CBUS II */
  BusEISA,                        /* Extended ISA */
  BusFUTURE,                      /* IEEE Futurebus */
  BusINTERN,                      /* Internal bus */
  BusISA,                         /* Industry Standard Architecture */
  BusMBI,                         /* Multibus I */
  BusMBII,                        /* Multibus II */
  BusMCA,                         /* Micro Channel Architecture */
  BusMPI,                         /* MPI */
  BusMPSA,                        /* MPSA */
  BusNUBUS,                       /* Apple Macintosh NuBus */
  BusPCI,                         /* Peripheral Component Interconnect */
  BusPCMCIA,                      /* PC Memory Card International Association */
  BusTC,                          /* DEC TurboChannel */
  BusVL,                          /* VESA Local bus */
  BusVME,                         /* VMEbus */
  BusXPRESS,                      /* Express System Bus */
};

#define MKBUS(t,b,d,f)    (((t)<<24)|(((b)&0xFF)<<16)|(((d)&0x1F)<<11)|(((f)&0x07)<<8))
#define BUSFNO(tbdf)      (((tbdf)>>8)&0x07)
#define BUSDNO(tbdf)      (((tbdf)>>11)&0x1F)
#define BUSBNO(tbdf)      (((tbdf)>>16)&0xFF)
#define BUSTYPE(tbdf)     ((tbdf)>>24)
#define BUSBDF(tbdf)      ((tbdf)&0x00FFFF00)
#define BUSUNKNOWN        (-1)

enum {
  MaxEISA         = 16,
  CfgEISA         = 0xC80,
};

/*
 * PCI support code.
 */
enum {                                  /* type 0 and type 1 pre-defined header */
  PciVID          = 0x00,         /* vendor ID */
  PciDID          = 0x02,         /* device ID */
  PciPCR          = 0x04,         /* command */
  PciPSR          = 0x06,         /* status */
  PciRID          = 0x08,         /* revision ID */
  PciCCRp         = 0x09,         /* programming interface class code */
  PciCCRu         = 0x0A,         /* sub-class code */
  PciCCRb         = 0x0B,         /* base class code */
  PciCLS          = 0x0C,         /* cache line size */
  PciLTR          = 0x0D,         /* latency timer */
  PciHDT          = 0x0E,         /* header type */
  PciBST          = 0x0F,         /* BIST */

  PciBAR0         = 0x10,         /* base address */
  PciBAR1         = 0x14,

  PciINTL         = 0x3C,         /* interrupt line */
  PciINTP         = 0x3D,         /* interrupt pin */
};

enum {                                  /* type 0 pre-defined header */
  PciBAR2         = 0x18,
  PciBAR3         = 0x1C,
  PciBAR4         = 0x20,
  PciBAR5         = 0x24,
  PciCIS          = 0x28,         /* cardbus CIS pointer */
  PciSVID         = 0x2C,         /* subsystem vendor ID */
  PciSID          = 0x2E,         /* cardbus CIS pointer */
  PciEBAR0        = 0x30,         /* expansion ROM base address */
  PciMGNT         = 0x3E,         /* burst period length */
  PciMLT          = 0x3F,         /* maximum latency between bursts */
};

enum {                                  /* type 1 pre-defined header */
  PciPBN          = 0x18,         /* primary bus number */
  PciSBN          = 0x19,         /* secondary bus number */
  PciUBN          = 0x1A,         /* subordinate bus number */
  PciSLTR         = 0x1B,         /* secondary latency timer */
  PciIBR          = 0x1C,         /* I/O base */
  PciILR          = 0x1D,         /* I/O limit */
  PciSPSR         = 0x1E,         /* secondary status */
```

```
137     PciMBR          = 0x20,         /* memory base */
138     PciMLR          = 0x22,         /* memory limit */
139     PciPMBR         = 0x24,         /* prefetchable memory base */
140     PciPMLR         = 0x26,         /* prefetchable memory limit */
141     PciPUBR         = 0x28,         /* prefetchable base upper 32 bits */
142     PciPULR         = 0x2C,         /* prefetchable limit upper 32 bits */
143     PciIUBR         = 0x30,         /* I/O base upper 16 bits */
144     PciIULR         = 0x32,         /* I/O limit upper 16 bits */
145     PciEBAR1        = 0x28,         /* expansion ROM base address */
146     PciBCR          = 0x3E,         /* bridge control register */
147   };
148
149   enum {                                    /* type 2 pre-defined header */
150     PciCBExCA       = 0x10,
151     PciCBSPSR       = 0x16,
152     PciCBPBN        = 0x18,         /* primary bus number */
153     PciCBSBN        = 0x19,         /* secondary bus number */
154     PciCBUBN        = 0x1A,         /* subordinate bus number */
155     PciCBSLTR       = 0x1B,         /* secondary latency timer */
156     PciCBMBR0       = 0x1C,
157     PciCBMLR0       = 0x20,
158     PciCBMBR1       = 0x24,
159     PciCBMLR1       = 0x28,
160     PciCBIBR0       = 0x2C,         /* I/O base */
161     PciCBILR0       = 0x30,         /* I/O limit */
162     PciCBIBR1       = 0x34,         /* I/O base */
163     PciCBILR1       = 0x38,         /* I/O limit */
164     PciCBSVID       = 0x40,         /* subsystem vendor ID */
165     PciCBSID        = 0x42,         /* subsystem ID */
166     PciCBLMBAR      = 0x44,         /* legacy mode base address */
167   };
168
169   typedef struct Pcisiz Pcisiz;
170   struct Pcisiz
171   {
172     Pcidev* dev;
173     int     siz;
174     int     bar;
175   };
176
177   typedef struct Pcidev Pcidev;
178   struct Pcidev
179   {
180     int     tbdf;                   /* type+bus+device+function */
181     ushort  vid;                    /* vendor ID */
182     ushort  did;                    /* device ID */
183
184     uchar   rid;
185     uchar   ccrp;
186     uchar   ccru;
187     uchar   ccrb;
188
189     struct {
190             ulong   bar;            /* base address */
191             int     size;
192     } mem[6];
193
194     uchar   intl;                   /* interrupt line */
195
196     Pcidev* list;
197     Pcidev* link;                   /* next device on this bno */
198
199     Pcidev* bridge;                 /* down a bus */
200     struct {
201             ulong   bar;
202             int     size;
203     } ioa, mema;
204     ulong   pcr;
205   };
206
207   #define PCIWINDOW 0
208   #define PCIWADDR(va)      (PADDR(va)+PCIWINDOW)
209   #define ISAWINDOW 0
210   #define ISAWADDR(va)      (PADDR(va)+ISAWINDOW)
211
212   /*
213    * PCMCIA support code.
214    */
215   /*
216    * Map between ISA memory space and PCMCIA card memory space.
217    */
218   struct PCMmap {
219     ulong   ca;                     /* card address */
220     ulong   cea;                    /* card end address */
221     ulong   isa;                    /* ISA address */
222     int     len;                    /* length of the ISA area */
223     int     attr;                   /* attribute memory */
224     int     ref;
225   };
226
227   /* SMBus transactions */
228   enum
229   {
230     SMBquick,               /* sends address only */
231
232     /* write */
233     SMBsend,                /* sends address and cmd */
234     SMBbytewrite,           /* sends address and cmd and 1 byte */
235     SMBwordwrite,           /* sends address and cmd and 2 bytes */
236
237     /* read */
238     SMBrecv,                /* sends address, recvs 1 byte */
239     SMBbyteread,            /* sends address and cmd, recv's byte */
240     SMBwordread,            /* sends address and cmd, recv's 2 bytes */
241   };
242
243   typedef struct SMBus SMBus;
244   struct SMBus {
245     QLock;          /* mutex */
246     Rendez  r;      /* rendezvous point for completion interrupts */
247     void    *arg;   /* implementation dependent */
248     ulong   base;   /* port or memory base of smbus */
249     int     busy;
250     void    (*transact)(SMBus*, int, int, int, uchar*);
251   };
```

```
#include  "u.h"
#include  "../port/lib.h"
#include  "mem.h"
#include  "dat.h"
#include  "fns.h"
#include  "io.h"
#include  "../port/error.h"

enum {
  Data=           0x60,           /* data port */

  Status=         0x64,           /* status port */
   Inready=       0x01,           /*  input character ready */
   Outbusy=       0x02,           /*  output busy */
   Sysflag=       0x04,           /*  system flag */
   Cmddata=       0x08,           /*  cmd==0, data==1 */
   Inhibit=       0x10,           /*  keyboard/mouse inhibited */
   Minready=      0x20,           /*  mouse character ready */
   Rtimeout=      0x40,           /*  general timeout */
   Parity=        0x80,

  Cmd=            0x64,           /* command port (write only) */

  Spec=           0x80,

  PF=             Spec|0x20,      /* num pad function key */
  View=           Spec|0x00,      /* view (shift window up) */
  KF=             0xF000, /* function key (begin Unicode private space) */
  Shift=          Spec|0x60,
  Break=          Spec|0x61,
  Ctrl=           Spec|0x62,
  Latin=          Spec|0x63,
  Caps=           Spec|0x64,
  Num=            Spec|0x65,
  Middle=         Spec|0x66,
  No=             0x00,           /* peter */

  Home=           KF|13,
  Up=             KF|14,
  Pgup=           KF|15,
  Print=          KF|16,
  Left=           KF|17,
  Right=          KF|18,
  End=            '\r',
  Down=           View,
  Pgdown=         KF|19,
  Ins=            KF|20,
  Del=            0x7F,
  Scroll=         KF|21,
};

/*
 * The codes at 0x79 and 0x81 are produed by the PFU Happy Hacking keyboard.
 * A 'standard' keyboard doesn't produce anything above 0x58.
 */
Rune kbtab[] =
{
[0x00]    No,     0x1b,   '1',    '2',    '3',    '4',    '5',    '6',
[0x08]    '7',    '8',    '9',    '0',    '-',    '=',    '\b',   '\t',
[0x10]    'q',    'w',    'e',    'r',    't',    'y',    'u',    'i',
[0x18]    'o',    'p',    '[',    ']',    '\n',   Ctrl,   'a',    's',
[0x20]    'd',    'f',    'g',    'h',    'j',    'k',    'l',    ';',
[0x28]    '\'',   '`',    Shift,  '\\',   'z',    'x',    'c',    'v',
[0x30]    'b',    'n',    'm',    ',',    '.',    '/',    Shift,  '*',
[0x38]    Latin,  ' ',    Ctrl,   KF|1,   KF|2,   KF|3,   KF|4,   KF|5,
[0x40]    KF|6,   KF|7,   KF|8,   KF|9,   KF|10,  Num,    Scroll, '7',
[0x48]    '8',    '9',    '-',    '4',    '5',    '6',    '+',    '1',
[0x50]    '2',    '3',    '0',    '.',    No,     No,     No,     KF|11,
[0x58]    KF|12,  No,     No,     No,     No,     No,     No,     No,
[0x60]    No,     No,     No,     No,     No,     No,     No,     No,
[0x68]    No,     No,     No,     No,     No,     No,     No,     No,
[0x70]    No,     No,     No,     No,     No,     No,     No,     No,
[0x78]    No,     View,   No,     Up,     No,     No,     No,     No,
};

Rune kbtabshift[] =
{
[0x00]    No,     0x1b,   '!',    '@',    '#',    '$',    '%',    '^',
[0x08]    '&',    '*',    '(',    ')',    '_',    '+',    '\b',   '\t',
[0x10]    'Q',    'W',    'E',    'R',    'T',    'Y',    'U',    'I',
[0x18]    'O',    'P',    '{',    '}',    '\n',   Ctrl,   'A',    'S',
[0x20]    'D',    'F',    'G',    'H',    'J',    'K',    'L',    ':',
[0x28]    '"',    '~',    Shift,  '|',    'Z',    'X',    'C',    'V',
[0x30]    'B',    'N',    'M',    '<',    '>',    '?',    Shift,  '*',
[0x38]    Latin,  ' ',    Ctrl,   KF|1,   KF|2,   KF|3,   KF|4,   KF|5,
[0x40]    KF|6,   KF|7,   KF|8,   KF|9,   KF|10,  Num,    Scroll, '7',
[0x48]    '8',    '9',    '-',    '4',    '5',    '6',    '+',    '1',
[0x50]    '2',    '3',    '0',    '.',    No,     No,     No,     KF|11,
[0x58]    KF|12,  No,     No,     No,     No,     No,     No,     No,
[0x60]    No,     No,     No,     No,     No,     No,     No,     No,
[0x68]    No,     No,     No,     No,     No,     No,     No,     No,
[0x70]    No,     No,     No,     No,     No,     No,     No,     No,
[0x78]    No,     Up,     No,     Up,     No,     No,     No,     No,
};

Rune kbtabesc1[] =
{
[0x00]    No,     No,     No,     No,     No,     No,     No,     No,
[0x08]    No,     No,     No,     No,     No,     No,     No,     No,
[0x10]    No,     No,     No,     No,     No,     No,     No,     No,
[0x18]    No,     No,     No,     No,     '\n',   Ctrl,   No,     No,
[0x20]    No,     No,     No,     No,     No,     No,     No,     No,
[0x28]    No,     No,     Shift,  No,     No,     No,     No,     No,
[0x30]    No,     No,     No,     No,     '/',    No,     Print,
[0x38]    Latin,  No,     No,     No,     No,     No,     No,     No,
[0x40]    No,     No,     No,     No,     No,     No,     Break,  Home,
[0x48]    Up,     Pgup,   No,     Left,   No,     Right,  No,     End,
[0x50]    Down,   Pgdown, Ins,    Del,    No,     No,     No,     No,
[0x58]    No,     No,     No,     No,     No,     No,     No,     No,
[0x60]    No,     No,     No,     No,     No,     No,     No,     No,
[0x68]    No,     No,     No,     No,     No,     No,     No,     No,
[0x70]    No,     No,     No,     No,     No,     No,     No,     No,
[0x78]    No,     Up,     No,     No,     No,     No,     No,     No,
};

enum
{
  /* controller command byte */
  Cscs1=          (1<<6),         /* scan code set 1 */
  Cauxdis=        (1<<5),         /* mouse disable */
  Ckbddis=        (1<<4),         /* kbd disable */
  Csf=            (1<<2),         /* system flag */
  Cauxint=        (1<<1),         /* mouse interrupt enable */
  Ckbdint=        (1<<0),         /* kbd interrupt enable */
};

static Lock i8042lock;
static uchar ccc;
static void (*auxputc)(int, int);

/*
 *  wait for output no longer busy
 */
static int
outready(void)
{
```

```
137     int tries;
138
139     for(tries = 0; (inb(Status) & Outbusy); tries++){
140             if(tries > 500)
141                     return -1;
142             delay(2);
143     }
144     return 0;
145   }
146
147   /*
148    *  wait for input
149    */
150   static int
151   inready(void)
152   {
153     int tries;
154
155     for(tries = 0; !(inb(Status) & Inready); tries++){
156             if(tries > 500)
157                     return -1;
158             delay(2);
159     }
160     return 0;
161   }
162
163   /*
164    *  ask 8042 to reset the machine
165    */
166   void
167   i8042reset(void)
168   {
169     ushort *s = KADDR(0x472);
170     int i, x;
171
172     *s = 0x1234;            /* BIOS warm-boot flag */
173
174     /*
175      *  newer reset the machine command
176      */
177     outready();
178     outb(Cmd, 0xFE);
179     outready();
180
181     /*
182      *  Pulse it by hand (old somewhat reliable)
183      */
184     x = 0xDF;
185     for(i = 0; i < 5; i++){
186             x ^= 1;
187             outready();
188             outb(Cmd, 0xD1);
189             outready();
190             outb(Data, x);  /* toggle reset */
191             delay(100);
192     }
193   }
194
195   int
196   i8042auxcmd(int cmd)
197   {
198     unsigned int c;
199     int tries;
200
201     c = 0;
202     tries = 0;
203
204     ilock(&i8042lock);
205     do{
206             if(tries++ > 2)
207                     break;
208             if(outready() < 0)
209                     break;
210             outb(Cmd, 0xD4);
211             if(outready() < 0)
212                     break;
213             outb(Data, cmd);
214             if(outready() < 0)
215                     break;
216             if(inready() < 0)
217                     break;
218             c = inb(Data);
219     } while(c == 0xFE || c == 0);
220     iunlock(&i8042lock);
221
222     if(c != 0xFA){
223             print("i8042: %2.2ux returned to the %2.2ux command\n", c, cmd);
224             return -1;
225     }
226     return 0;
227   }
228
229   /*
230    *  keyboard interrupt
231    */
232   static void
233   i8042intr(Ureg*, void*)
234   {
235     int s, c, i;
236     static int esc1, esc2;
237     static int alt, caps, ctl, num, shift;
238     static int collecting, nk;
239     static Rune kc[5];
240     int keyup;
241
242     /*
243      *  get status
244      */
245     lock(&i8042lock);
246     s = inb(Status);
247     if(!(s&Inready)){
248             unlock(&i8042lock);
249             return;
250     }
251
252     /*
253      *  get the character
254      */
255     c = inb(Data);
256     unlock(&i8042lock);
257
258     /*
259      *  if it's the aux port...
260      */
261     if(s & Minready){
262             if(auxputc != nil)
263                     auxputc(c, shift);
264             return;
265     }
266
267     /*
268      *  e0's is the first of a 2 character sequence
269      */
270     if(c == 0xe0){
271             esc1 = 1;
272             return;
```

```
273     } else if(c == 0xe1){
274             esc2 = 2;
275             return;
276     }
277
278     keyup = c&0x80;
279     c &= 0x7f;
280     if(c > sizeof kbtab){
281             c |= keyup;
282             if(c != 0xFF)   /* these come fairly often: CAPSLOCK U Y */
283                     print("unknown key %ux\n", c);
284             return;
285     }
286
287     if(esc1){
288             c = kbtabesc1[c];
289             esc1 = 0;
290     } else if(esc2){
291             esc2--;
292             return;
293     } else if(shift)
294             c = kbtabshift[c];
295     else
296             c = kbtab[c];
297
298     if(caps && c<='z' && c>='a')
299             c += 'A' - 'a';
300
301     /*
302      *  keyup only important for shifts
303      */
304     if(keyup){
305             switch(c){
306             case Latin:
307                     alt = 0;
308                     break;
309             case Shift:
310                     shift = 0;
311                     break;
312             case Ctrl:
313                     ctl = 0;
314                     break;
315             }
316             return;
317     }
318
319     /*
320      *  normal character
321      */
322     if(!(c & (Spec|KF))){
323             if(ctl){
324                     if(alt && c == Del)
325                             exit(0);
326                     c &= 0x1f;
327             }
328             if(!collecting){
329                     kbdputc(kbdq, c);
330                     return;
331             }
332             kc[nk++] = c;
333             c = latin1(kc, nk);
334             if(c < -1)      /* need more keystrokes */
335                     return;
336             if(c != -1)     /* valid sequence */
337                     kbdputc(kbdq, c);
338             else    /* dump characters */
339                     for(i=0; i<nk; i++)
340                             kbdputc(kbdq, kc[i]);
341             nk = 0;
342             collecting = 0;
343             return;
344     } else {
345             switch(c){
346             case Caps:
347                     caps ^= 1;
348                     return;
349             case Num:
350                     num ^= 1;
351                     return;
352             case Shift:
353                     shift = 1;
354                     return;
355             case Latin:
356                     alt = 1;
357                     collecting = 1;
358                     nk = 0;
359                     return;
360             case Ctrl:
361                     ctl = 1;
362                     return;
363             }
364     }
365     kbdputc(kbdq, c);
366   }
367
368   void
369   i8042auxenable(void (*putc)(int, int))
370   {
371     char *err = "i8042: aux init failed\n";
372
373     /* enable kbd/aux xfers and interrupts */
374     ccc &= ~Cauxdis;
375     ccc |= Cauxint;
376
377     ilock(&i8042lock);
378     if(outready() < 0)
379             print(err);
380     outb(Cmd, 0x60);                        /* write control register */
381     if(outready() < 0)
382             print(err);
383     outb(Data, ccc);
384     if(outready() < 0)
385             print(err);
386     outb(Cmd, 0xA8);                        /* auxilliary device enable */
387     if(outready() < 0){
388             iunlock(&i8042lock);
389             return;
390     }
391     auxputc = putc;
392     intrenable(IrqAUX, i8042intr, 0, BUSUNKNOWN, "kbdaux");
393     iunlock(&i8042lock);
394   }
395
396   void
397   kbdinit(void)
398   {
399     int c;
400
401     kbdq = qopen(4*1024, 0, 0, 0);
402     if(kbdq == nil)
403             panic("kbdinit");
404     qnoblock(kbdq, 1);
405
406     ioalloc(Data, 1, 0, "kbd");
407     ioalloc(Cmd, 1, 0, "kbd");
408
```

```
409     intrenable(IrqKBD, i8042intr, 0, BUSUNKNOWN, "kbd");
410
411     /* wait for a quiescent controller */
412     while((c = inb(Status)) & (Outbusy | Inready))
413             if(c & Inready)
414                     inb(Data);
415
416     /* get current controller command byte */
417     outb(Cmd, 0x20);
418     if(inready() < 0){
419             print("kbdinit: can't read ccc\n");
420             ccc = 0;
421     } else
422             ccc = inb(Data);
423
424     /* enable kbd xfers and interrupts */
425     /* disable mouse */
426     ccc &= ~Ckbddis;
427     ccc |= Csf | Ckbdint | Cscs1;
428     if(outready() < 0)
429             print("kbd init failed\n");
430     outb(Cmd, 0x60);
431     if(outready() < 0)
432             print("kbd init failed\n");
433     outb(Data, ccc);
434     outready();
435   }
```

```
#include  "u.h"
#include  "../port/lib.h"
#include  "mem.h"
#include  "dat.h"
#include  "fns.h"
#include  "io.h"
#include  "ureg.h"
#include  "init.h"
#include  "pool.h"

Mach *m;

static  uchar *sp;        /* stack pointer for /boot */

/*
 * Where configuration info is left for the loaded programme.
 * This will turn into a structure as more is done by the boot loader
 * (e.g. why parse the .ini file twice?).
 * There are 1024 bytes available at CONFADDR.
 */
#define BOOTLINE  ((char*)CONFADDR)
#define BOOTLINELEN       64
#define BOOTARGS  ((char*)(CONFADDR+BOOTLINELEN))
#define   BOOTARGSLEN     (1024-BOOTLINELEN)
#define   MAXCONF         32

char bootdisk[NAMELEN];
char *confname[MAXCONF];
char *confval[MAXCONF];
int nconf;

extern void ns16552install(void); /* botch: config */

static int isoldbcom;

static int
getcfields(char* lp, char** fields, int n, char* sep)
{
  int i;

  for(i = 0; lp && *lp && i < n; i++){
          while(*lp && strchr(sep, *lp) != 0)
                  *lp++ = 0;
          if(*lp == 0)
                  break;
          fields[i] = lp;
          while(*lp && strchr(sep, *lp) == 0){
                  if(*lp == '\\' && *(lp+1) == '\n')
                          *lp++ = ' ';
                  lp++;
          }
  }

  return i;
}

static void
options(void)
{
  uchar *bda;
  long i, n;
  char *cp, *line[MAXCONF], *p, *q;

  if(strncmp(BOOTARGS, "ZORT 0\r\n", 8)){
          isoldbcom = 1;

          memmove(BOOTARGS, KADDR(1024), BOOTARGSLEN);
          memmove(BOOTLINE, KADDR(0x100), BOOTLINELEN);

          bda = KADDR(0x400);
          bda[0x13] = 639;
          bda[0x14] = 639>>8;
  }

  /*
   *  parse configuration args from dos file plan9.ini
   */
  cp = BOOTARGS;  /* where b.com leaves its config */
  cp[BOOTARGSLEN-1] = 0;

  /*
   * Strip out '\r', change '\t' -> ' '.
   */
  p = cp;
  for(q = cp; *q; q++){
          if(*q == '\r')
                  continue;
          if(*q == '\t')
                  *q = ' ';
          *p++ = *q;
  }
  *p = 0;

  n = getcfields(cp, line, MAXCONF, "\n");
  for(i = 0; i < n; i++){
          if(*line[i] == '#')
                  continue;
          cp = strchr(line[i], '=');
          if(cp == 0)
                  continue;
          *cp++ = 0;
          if(cp - line[i] >= NAMELEN+1)
                  *(line[i]+NAMELEN-1) = 0;
          confname[nconf] = line[i];
          confval[nconf] = cp;
          nconf++;
  }
}

void
main(void)
{
  outb(0x3F2, 0x00);                      /* botch: turn off the floppy motor */

  /*
   * There is a little leeway here in the ordering but care must be
   * taken with dependencies:
   *      function                dependencies
   *      ========                ============
   *      machinit                depends on: m->machno, m->pdb
   *      cpuidentify             depends on: m
   *      confinit                calls: meminit
   *      meminit                 depends on: cpuidentify (needs to know processor
   *                                type for caching, etc.)
   *      archinit                depends on: meminit (MP config table may be at the
   *                                top of system physical memory);
   *                              conf.nmach (not critical, mpinit will check);
   *      arch->intrinit          depends on: trapinit
   */
  conf.nmach = 1;
  MACHP(0) = (Mach*)CPU0MACH;
  m->pdb = (ulong*)CPU0PDB;
  machinit();
  ioinit();
  active.machs = 1;
  active.exiting = 0;
```

```
137     options();
138     screeninit();
139     cpuidentify();
140     confinit();
141     archinit();
142     xinit();
143     trapinit();
144     printinit();
145     cpuidprint();
146     if(isoldbcom)
147             print("    ****OLD B.COM - UPGRADE****\n");
148     mmuinit();
149     if(arch->intrinit)
150             arch->intrinit();
151     ns16552install();                       /* botch: config */
152     mathinit();
153     kbdinit();
154     if(arch->clockenable)
155             arch->clockenable();
156     procinit0();
157     initseg();
158     links();
159   conf.monitor = 1;
160     chandevreset();
161     pageinit();
162     swapinit();
163     userinit();
164     schedinit();
165   }
166
167   void
168   machinit(void)
169   {
170     int machno;
171     ulong *pdb;
172
173     machno = m->machno;
174     pdb = m->pdb;
175     memset(m, 0, sizeof(Mach));
176     m->machno = machno;
177     m->pdb = pdb;
178   }
179
180   void
181   ksetterm(char *f)
182   {
183     char buf[2*NAMELEN];
184
185     sprint(buf, f, conffile);
186     ksetenv("terminal", buf);
187   }
188
189   void
190   init0(void)
191   {
192     int i;
193     char tstr[32];
194
195     up->nerrlab = 0;
196
197     spllo();
198
199     /*
200      * These are o.k. because rootinit is null.
201      * Then early kproc's will have a root and dot.
202      */
203     up->slash = namec("#/", Atodir, 0, 0);
204     cnameclose(up->slash->name);
205     up->slash->name = newcname("/");
206     up->dot = cclone(up->slash, 0);
207
208     chandevinit();
209
210     if(!waserror()){
211             strcpy(tstr, arch->id);
212             strcat(tstr, " %s");
213             ksetterm(tstr);
214             ksetenv("cputype", "386");
215             if(cpuserver)
216                     ksetenv("service", "cpu");
217             else
218                     ksetenv("service", "terminal");
219             for(i = 0; i < nconf; i++)
220                     if(confname[i] && confname[i][0] != '*')
221                             ksetenv(confname[i], confval[i]);
222             poperror();
223     }
224     kproc("alarm", alarmkproc, 0);
225     touser(sp);
226   }
227
228   void
229   userinit(void)
230   {
231     Proc *p;
232     Segment *s;
233     KMap *k;
234     Page *pg;
235
236     p = newproc();
237     p->pgrp = newpgrp();
238     p->egrp = smalloc(sizeof(Egrp));
239     p->egrp->ref = 1;
240     p->fgrp = dupfgrp(nil);
241     p->rgrp = newrgrp();
242     p->procmode = 0640;
243
244     strcpy(p->text, "*init*");
245     strcpy(p->user, eve);
246     p->fpstate = FPinit;
247     fpoff();
248
249     /*
250      * Kernel Stack
251      *
252      * N.B. The -12 for the stack pointer is important.
253      *      4 bytes for gotolabel's return PC
254      */
255     p->sched.pc = (ulong)init0;
256     p->sched.sp = (ulong)p->kstack+KSTACK-(1+MAXSYSARG)*BY2WD;
257
258     /*
259      * User Stack
260      */
261     s = newseg(SG_STACK, USTKTOP-USTKSIZE, USTKSIZE/BY2PG);
262     p->seg[SSEG] = s;
263     pg = newpage(1, 0, USTKTOP-BY2PG);
264     segpage(s, pg);
265     k = kmap(pg);
266     bootargs(VA(k));
267     kunmap(k);
268
269     /*
270      * Text
271      */
272     s = newseg(SG_TEXT, UTZERO, 1);
```

```
    s->flushme++;
    p->seg[TSEG] = s;
    pg = newpage(1, 0, UTZERO);
    memset(pg->cachectl, PG_TXTFLUSH, sizeof(pg->cachectl));
    segpage(s, pg);
    k = kmap(s->map[0]->pages[0]);
    memmove((ulong*)VA(k), initcode, sizeof initcode);
    kunmap(k);

    ready(p);
}

uchar *
pusharg(char *p)
{
  int n;

  n = strlen(p)+1;
  sp -= n;
  memmove(sp, p, n);
  return sp;
}

void
bootargs(ulong base)
{
  int i, ac;
  uchar *av[32];
  uchar **lsp;
  char *cp = BOOTLINE;
  char buf[64];

  sp = (uchar*)base + BY2PG - MAXSYSARG*BY2WD;

  ac = 0;
  av[ac++] = pusharg("/386/9dos");
  cp[BOOTLINELEN-1] = 0;
  buf[0] = 0;
  if(strncmp(cp, "fd", 2) == 0){
          sprint(buf, "local!#f/fd%lddisk", strtol(cp+2, 0, 0));
          av[ac++] = pusharg(buf);
  } else if(strncmp(cp, "sd", 2) == 0){
          sprint(buf, "local!#S/sd%c%c/fs", *(cp+2), *(cp+3));
          av[ac++] = pusharg(buf);
  } else if(strncmp(cp, "ether", 5) == 0)
          av[ac++] = pusharg("-n");
  if(buf[0]){
          cp = strchr(buf, '!');
          if(cp){
                  strcpy(bootdisk, cp+1);
                  addconf("bootdisk", bootdisk);
          }
  }

  /* 4 byte word align stack */
  sp = (uchar*)((ulong)sp & ~3);

  /* build argc, argv on stack */
  sp -= (ac+1)*sizeof(sp);
  lsp = (uchar**)sp;
  for(i = 0; i < ac; i++)
          *lsp++ = av[i] + ((USTKTOP - BY2PG) - base);
  *lsp = 0;
  sp += (USTKTOP - BY2PG) - base - sizeof(ulong);
}

Conf      conf;

void
addconf(char *name, char *val)
{
  if(nconf >= MAXCONF)
          return;
  confname[nconf] = name;
  confval[nconf] = val;
  nconf++;
}

char*
getconf(char *name)
{
  int i;

  for(i = 0; i < nconf; i++)
          if(cistrcmp(confname[i], name) == 0)
                  return confval[i];
  return 0;
}

void
confinit(void)
{
  char *p;
  int userpcnt;
  ulong kpages, maxmem;

  if(p = getconf("*maxmem"))
          maxmem = strtoul(p, 0, 0);
  else
          maxmem = 0;
  if(p = getconf("*kernelpercent"))
          userpcnt = 100 - strtol(p, 0, 0);
  else
          userpcnt = 0;

  meminit(maxmem);

  conf.npage = conf.npage0 + conf.npage1;

  conf.nproc = 100 + ((conf.npage*BY2PG)/MB)*5;
  if(cpuserver)
          conf.nproc *= 3;
  if(conf.nproc > 2000)
          conf.nproc = 2000;
  conf.nimage = 200;
  conf.nswap = conf.nproc*80;
  conf.nswppo = 4096;

  if(cpuserver) {
          if(userpcnt < 10)
                  userpcnt = 70;
          kpages = conf.npage - (conf.npage*userpcnt)/100;

          /*
           * Hack for the big boys. Only good while physmem < 4GB.
           * Give the kernel a max. of 16MB + enough to allocate the
           * page pool.
           * This is an overestimate as conf.upages < conf.npages.
           * The patch of nimage is a band-aid, scanning the whole
           * page list in imagereclaim just takes too long.
           */
          if(kpages > (16*MB + conf.npage*sizeof(Page))/BY2PG){
                  kpages = (16*MB + conf.npage*sizeof(Page))/BY2PG;
                  conf.nimage = 2000;
                  kpages += (conf.nproc*KSTACK)/BY2PG;
          }
```

```
409     } else {
410             if(userpcnt < 10) {
411                     if(conf.npage*BY2PG < 16*MB)
412                             userpcnt = 40;
413                     else
414                             userpcnt = 60;
415             }
416             kpages = conf.npage - (conf.npage*userpcnt)/100;
417
418             /*
419              * Make sure terminals with low memory get at least
420              * 4MB on the first Image chunk allocation.
421              */
422             if(conf.npage*BY2PG < 16*MB)
423                     imagmem->minarena = 4*1024*1024;
424     }
425     conf.upages = conf.npage - kpages;
426     conf.ialloc = (kpages/2)*BY2PG;
427
428     /*
429      * Guess how much is taken by the large permanent
430      * datastructures. Mntcache and Mntrpc are not accounted for
431      * (probably ~300KB).
432      */
433     kpages *= BY2PG;
434     kpages -= conf.upages*sizeof(Page)
435             + conf.nproc*sizeof(Proc)
436             + conf.nimage*sizeof(Image)
437             + conf.nswap
438             + conf.nswppo*sizeof(Page);
439     mainmem->maxsize = kpages;
440     if(!cpuserver){
441             /*
442              * give terminals lots of image memory, too; the dynamic
443              * allocation will balance the load properly, hopefully.
444              * be careful with 32-bit overflow.
445              */
446             imagmem->maxsize = kpages;
447     }
448   }
449
450   static char* mathmsg[] =
451   {
452     "invalid",
453     "denormalized",
454     "div-by-zero",
455     "overflow",
456     "underflow",
457     "precision",
458     "stack",
459     "error",
460   };
461
462   static void
463   mathnote(void)
464   {
465     int i;
466     ulong status;
467     char *msg, note[ERRLEN];
468
469     status = up->fpsave.status;
470
471     /*
472      * Some attention should probably be paid here to the
473      * exception masks and error summary.
474      */
475     msg = "unknown";
476     for(i = 0; i < 8; i++){
477             if(!((1<<i) & status))
478                     continue;
479             msg = mathmsg[i];
480             break;
481     }
482     sprint(note, "sys: fp: %s fppc=0x%lux", msg, up->fpsave.pc);
483     postnote(up, 1, note, NDebug);
484   }
485
486   /*
487    *  math coprocessor error
488    */
489   static void
490   matherror(Ureg *ur, void*)
491   {
492     /*
493      *  a write cycle to port 0xF0 clears the interrupt latch attached
494      *  to the error# line from the 387
495      */
496     if(!(m->cpuiddx & 0x01))
497             outb(0xF0, 0xFF);
498
499     /*
500      *  save floating point state to check out error
501      */
502     fpenv(&up->fpsave);
503     mathnote();
504
505     if(ur->pc & KZERO)
506             panic("fp: status %lux fppc=0x%lux pc=0x%lux",
507                     up->fpsave.status, up->fpsave.pc, ur->pc);
508   }
509
510   /*
511    *  math coprocessor emulation fault
512    */
513   static void
514   mathemu(Ureg*, void*)
515   {
516     switch(up->fpstate){
517     case FPinit:
518             fpinit();
519             up->fpstate = FPactive;
520             break;
521     case FPinactive:
522             /*
523              * Before restoring the state, check for any pending
524              * exceptions, there's no way to restore the state without
525              * generating an unmasked exception.
526              * More attention should probably be paid here to the
527              * exception masks and error summary.
528              */
529             if((up->fpsave.status & ~up->fpsave.control) & 0x07F){
530                     mathnote();
531                     break;
532             }
533             fprestore(&up->fpsave);
534             up->fpstate = FPactive;
535             break;
536     case FPactive:
537             panic("math emu", 0);
538             break;
539     }
540   }
541
542   /*
543    *  math coprocessor segment overrun
544    */
```

```
static void
mathover(Ureg*, void*)
{
  pexit("math overrun", 0);
}

void
mathinit(void)
{
  trapenable(VectorCERR, matherror, 0, "matherror");
  if(X86FAMILY(m->cpuidax) == 3)
          intrenable(IrqIRQ13, matherror, 0, BUSUNKNOWN, "matherror");
  trapenable(VectorCNA, mathemu, 0, "mathemu");
  trapenable(VectorCSO, mathover, 0, "mathover");
}

/*
 *  set up floating point for a new process
 */
void
procsetup(Proc *p)
{
  p->fpstate = FPinit;
  fpoff();
}

/*
 *  Save the mach dependent part of the process state.
 */
void
procsave(Proc *p)
{
  if(p->fpstate == FPactive){
          if(p->state == Moribund)
                  fpoff();
          else{
                  /*
                   * Fpsave() stores without handling pending
                   * unmasked exeptions. Postnote() can't be called
                   * here as sleep() already has up->rlock, so
                   * the handling of pending exceptions is delayed
                   * until the process runs again and generates an
                   * emulation fault to activate the FPU.
                   */
                  fpsave(&up->fpsave);
          }
          p->fpstate = FPinactive;
  }

  /*
   * Switch to the prototype page tables for this processor.
   * While this processor is in the scheduler, the process could run
   * on another processor and exit, returning the page tables to
   * the free list where they could be reallocated and overwritten.
   * When this processor eventually has to get an entry from the
   * trashed page tables it will crash.
   */
  mmuflushtlb(PADDR(m->pdb));
}

void
exit(int ispanic)
{
  int ms, once;

  lock(&active);
  if(ispanic)
          active.ispanic = ispanic;
  else if(m->machno == 0 && (active.machs & (1<<m->machno)) == 0)
          active.ispanic = 0;
  once = active.machs & (1<<m->machno);
  active.machs &= ~(1<<m->machno);
  active.exiting = 1;
  unlock(&active);

  if(once)
          print("cpu%d: exiting\n", m->machno);
  spllo();
  for(ms = 5*1000; ms > 0; ms -= TK2MS(2)){
          delay(TK2MS(2));
          if(active.machs == 0 && consactive() == 0)
                  break;
  }

  if(active.ispanic && m->machno == 0){
          if(cpuserver)
                  delay(10000);
          else
                  for(;;);
  }
  else
          delay(1000);

  arch->reset();
}

int
isaconfig(char *class, int ctlrno, ISAConf *isa)
{
  char cc[NAMELEN], *p, *q, *r;
  int n;

  sprint(cc, "%s%d", class, ctlrno);
  for(n = 0; n < nconf; n++){
          if(cistrncmp(confname[n], cc, NAMELEN))
                  continue;
          isa->nopt = 0;
          p = confval[n];
          while(*p){
                  while(*p == ' ' || *p == '\t')
                          p++;
                  if(*p == '\0')
                          break;
                  if(cistrncmp(p, "type=", 5) == 0){
                          p += 5;
                          for(q = isa->type; q < &isa->type[NAMELEN-1]; q++){
                                  if(*p == '\0' || *p == ' ' || *p == '\t')
                                          break;
                                  *q = *p++;
                          }
                          *q = '\0';
                  }
                  else if(cistrncmp(p, "port=", 5) == 0)
                          isa->port = strtoul(p+5, &p, 0);
                  else if(cistrncmp(p, "irq=", 4) == 0)
                          isa->irq = strtoul(p+4, &p, 0);
                  else if(cistrncmp(p, "dma=", 4) == 0)
                          isa->dma = strtoul(p+4, &p, 0);
                  else if(cistrncmp(p, "mem=", 4) == 0)
                          isa->mem = strtoul(p+4, &p, 0);
                  else if(cistrncmp(p, "size=", 5) == 0)
                          isa->size = strtoul(p+5, &p, 0);
                  else if(cistrncmp(p, "freq=", 5) == 0)
                          isa->freq = strtoul(p+5, &p, 0);
                  else if(isa->nopt < NISAOPT){
                          r = isa->opt[isa->nopt];
```

```
                        while(*p && *p != ' ' && *p != '\t'){
                                *r++ = *p++;
                                if(r-isa->opt[isa->nopt] >= ISAOPTLEN-1)
                                        break;
                        }
                        *r = '\0';
                        isa->nopt++;
                }
                while(*p && *p != ' ' && *p != '\t')
                        p++;
        }
        return 1;
  }
  return 0;
}

/*
int
iprint(char *fmt, ...)
{
  char buf[PRINTSIZE];
  int n;
  va_list arg;

  va_start(arg, fmt);
  n = doprint(buf, buf+sizeof(buf), fmt, arg) - buf;
  va_end(arg);

  screenputs(buf, n);

  return n;
}
*/

int
cistrcmp(char *a, char *b)
{
  int ac, bc;

  for(;;){
        ac = *a++;
        bc = *b++;

        if(ac >= 'A' && ac <= 'Z')
                ac = 'a' + (ac - 'A');
        if(bc >= 'A' && bc <= 'Z')
                bc = 'a' + (bc - 'A');
        ac -= bc;
        if(ac)
                return ac;
        if(bc == 0)
                break;
  }
  return 0;
}

int
cistrncmp(char *a, char *b, int n)
{
  unsigned ac, bc;

  while(n > 0){
        ac = *a++;
        bc = *b++;
        n--;

        if(ac >= 'A' && ac <= 'Z')
                ac = 'a' + (ac - 'A');
        if(bc >= 'A' && bc <= 'Z')
                bc = 'a' + (bc - 'A');

        ac -= bc;
        if(ac)
                return ac;
        if(bc == 0)
                break;
  }

  return 0;
}
```

```
/*
 * Memory and machine-specific definitions.  Used in C and assembler.
 */

/*
 * Sizes
 */
#define   BI2BY           8                       /* bits per byte */
#define BI2WD             32                      /* bits per word */
#define   BY2WD           4                       /* bytes per word */
#define   BY2V            8                       /* bytes per double word */
#define   BY2PG           4096                    /* bytes per page */
#define   WD2PG           (BY2PG/BY2WD)           /* words per page */
#define   PGSHIFT         12                      /* log(BY2PG) */
#define ROUND(s, sz)      (((s)+((sz)-1))&~((sz)-1))
#define PGROUND(s)        ROUND(s, BY2PG)
#define BLOCKALIGN        8

#define   MAXMACH         8                       /* max # cpus system can run */
#define KSTACK            4096                    /* Size of kernel stack */

/*
 * Time
 */
#define   HZ              (82)                    /* clock frequency */
#define   MS2HZ           (1000/HZ)               /* millisec per clock tick */
#define   TK2SEC(t)       ((t)/HZ)                /* ticks to seconds */
#define   MS2TK(t)        (((t)*HZ)/1000)         /* milliseconds to ticks */

/*
 * Fundamental addresses
 */
#define IDTADDR           0x80000800              /* idt */
#define APBOOTSTRAP       0x80001000              /* AP bootstrap code */
#define CONFADDR  0x80001200              /* info passed from boot loader */
#define CPU0PDB           0x80002000              /* bootstrap processor PDB */
#define CPU0PTE           0x80003000              /* bootstrap processor PTE's for 0-4MB */
#define MACHADDR  0x80004000              /* as seen by current processor */
#define CPU0MACH  0x80005000              /* Mach for bootstrap processor */
#define   MACHSIZE        BY2PG

/*
 *  Address spaces
 *
 *  User is at 0-2GB
 *  Kernel is at 2GB-4GB
 */
#define   UZERO           0                       /* base of user address space */
#define   UTZERO          (UZERO+BY2PG)           /* first address in user text */
#define   KZERO           0x80000000              /* base of kernel address space */
#define   KTZERO          0x80100000              /* first address in kernel text */
#define   USTKTOP         (KZERO-BY2PG)           /* byte just beyond user stack */
#define   USTKSIZE        (16*1024*1024)          /* size of user stack */
#define   TSTKTOP         (USTKTOP-USTKSIZE)      /* end of new stack in sysexec */
#define TSTKSIZ   100

/*
 *  known x86 segments (in GDT) and their selectors
 */
#define   NULLSEG 0       /* null segment */
#define   KDSEG   1       /* kernel data/stack */
#define   KESEG   2       /* kernel executable */
#define   UDSEG   3       /* user data/stack */
#define   UESEG   4       /* user executable */
#define TSSSEG    5       /* task segment */

#define SELGDT    (0<<2)  /* selector is in gdt */
#define   SELLDT  (1<<2)  /* selector is in ldt */

#define SELECTOR(i, t, p) (((i)<<3) | (t) | (p))

#define NULLSEL   SELECTOR(NULLSEG, SELGDT, 0)
#define KESEL     SELECTOR(KESEG, SELGDT, 0)
#define KDSEL     SELECTOR(KDSEG, SELGDT, 0)
#define UESEL     SELECTOR(UESEG, SELGDT, 3)
#define UDSEL     SELECTOR(UDSEG, SELGDT, 3)
#define TSSSEL    SELECTOR(TSSSEG, SELGDT, 0)

/*
 *  fields in segment descriptors
 */
#define SEGDATA   (0x10<<8)       /* data/stack segment */
#define SEGEXEC   (0x18<<8)       /* executable segment */
#define   SEGTSS  (0x9<<8)        /* TSS segment */
#define SEGCG     (0x0C<<8)       /* call gate */
#define   SEGIG   (0x0E<<8)       /* interrupt gate */
#define SEGTG     (0x0F<<8)       /* trap gate */
#define SEGTYPE   (0x1F<<8)

#define SEGP      (1<<15)         /* segment present */
#define SEGPL(x) ((x)<<13)        /* priority level */
#define SEGB      (1<<22)         /* granularity 1==4k (for expand-down) */
#define SEGG      (1<<23)         /* granularity 1==4k (for other) */
#define SEGE      (1<<10)         /* expand down */
#define SEGW      (1<<9)          /* writable (for data/stack) */
#define   SEGR    (1<<9)          /* readable (for code) */
#define SEGD      (1<<22)         /* default 1==32bit (for code) */

/*
 *  virtual MMU
 */
#define PTEMAPMEM (1024*1024)
#define   PTEPERTAB       (PTEMAPMEM/BY2PG)
#define SEGMAPSIZE        1984
#define SSEGMAPSIZE       16
#define PPN(x)            ((x)&~(BY2PG-1))

/*
 *  physical MMU
 */
#define   PTEVALID        (1<<0)
#define   PTEWT           (1<<3)
#define   PTEUNCACHED     (1<<4)
#define   PTEWRITE        (1<<1)
#define   PTERONLY        (0<<1)
#define   PTEKERNEL       (0<<2)
#define   PTEUSER         (1<<2)
#define   PTESIZE         (1<<7)

#define getpgcolor(a)     0
```

```
/*
 * Size memory and create the kernel page-tables on the fly while doing so.
 * Called from main(), this code should only be run by the bootstrap processor.
 */
#include "u.h"
#include "../port/lib.h"
#include "mem.h"
#include "dat.h"
#include "fns.h"
#include "io.h"

#define MEMDEBUG	0

#define PDX(va)		((((ulong)(va))>>22) & 0x03FF)
#define PTX(va)		((((ulong)(va))>>12) & 0x03FF)

enum {
	MemUPA		= 0,		/* unbacked physical address */
	MemRAM		= 1,		/* physical memory */
	MemUMB		= 2,		/* upper memory block (<16MB) */
	NMemType	= 3,

	KB		= 1024,

	MemMinMB	= 4,		/* minimum physical memory (<=4MB) */
	MemMaxMB	= 768,		/* maximum physical memory to check */

	NMemBase	= 10,
};

typedef struct {
	int	size;
	ulong	addr;
} Map;

typedef struct {
	char*	name;
	Map*	map;
	Map*	mapend;

	Lock;
} RMap;

static Map mapupa[16];
static RMap rmapupa = {
	"unallocated unbacked physical memory",
	mapupa,
	&mapupa[nelem(mapupa)-1],
};

static Map xmapupa[16];
static RMap xrmapupa = {
	"unbacked physical memory",
	xmapupa,
	&xmapupa[nelem(xmapupa)-1],
};

static Map mapram[16];
static RMap rmapram = {
	"physical memory",
	mapram,
	&mapram[nelem(mapram)-1],
};

static Map mapumb[64];
static RMap rmapumb = {
	"upper memory block",
	mapumb,
	&mapumb[nelem(mapumb)-1],
};

static Map mapumbrw[16];
static RMap rmapumbrw = {
	"UMB device memory",
	mapumbrw,
	&mapumbrw[nelem(mapumbrw)-1],
};

void
memdebug(void)
{
	Map *mp;
	ulong maxpa, maxpa1, maxpa2;

	if(MEMDEBUG == 0)
		return;

	maxpa = (nvramread(0x18)<<8)|nvramread(0x17);
	maxpa1 = (nvramread(0x31)<<8)|nvramread(0x30);
	maxpa2 = (nvramread(0x16)<<8)|nvramread(0x15);
	print("maxpa = %luX -> %luX, maxpa1 = %luX maxpa2 = %luX\n",
		maxpa, MB+maxpa*KB, maxpa1, maxpa2);

	for(mp = rmapram.map; mp->size; mp++)
		print("%8.8luX %8.8uX %8.8luX\n", mp->addr, mp->size, mp->addr+mp->size);
	for(mp = rmapumb.map; mp->size; mp++)
		print("%8.8luX %8.8uX %8.8luX\n", mp->addr, mp->size, mp->addr+mp->size);
	for(mp = rmapumbrw.map; mp->size; mp++)
		print("%8.8luX %8.8uX %8.8luX\n", mp->addr, mp->size, mp->addr+mp->size);
	for(mp = rmapupa.map; mp->size; mp++)
		print("%8.8luX %8.8uX %8.8luX\n", mp->addr, mp->size, mp->addr+mp->size);
}

void
mapfree(RMap* rmap, ulong addr, ulong size)
{
	Map *mp;
	ulong t;

	if(size <= 0)
		return;

	lock(rmap);
	for(mp = rmap->map; mp->addr <= addr && mp->size; mp++)
		;

	if(mp > rmap->map && (mp-1)->addr+(mp-1)->size == addr){
		(mp-1)->size += size;
		if(addr+size == mp->addr){
			(mp-1)->size += mp->size;
			while(mp->size){
				mp++;
				(mp-1)->addr = mp->addr;
				(mp-1)->size = mp->size;
			}
		}
	}
	else{
		if(addr+size == mp->addr && mp->size){
			mp->addr -= size;
			mp->size += size;
		}
		else do{
			if(mp >= rmap->mapend){
				print("mapfree: %s: losing 0x%luX, %ld\n",
					rmap->name, addr, size);
```

```
137                             break;
138                     }
139                     t = mp->addr;
140                     mp->addr = addr;
141                     addr = t;
142                     t = mp->size;
143                     mp->size = size;
144                     mp++;
145             }while(size = t);
146     }
147     unlock(rmap);
148   }
149
150   ulong
151   mapalloc(RMap* rmap, ulong addr, int size, int align)
152   {
153     Map *mp;
154     ulong maddr, oaddr;
155
156     lock(rmap);
157     for(mp = rmap->map; mp->size; mp++){
158             maddr = mp->addr;
159
160             if(addr){
161                     /*
162                      * A specific address range has been given:
163                      *   if the current map entry is greater then
164                      *   the address is not in the map;
165                      *   if the current map entry does not overlap
166                      *   the beginning of the requested range then
167                      *   continue on to the next map entry;
168                      *   if the current map entry does not entirely
169                      *   contain the requested range then the range
170                      *   is not in the map.
171                      */
172                     if(maddr > addr)
173                             break;
174                     if(maddr+mp->size < addr)
175                             continue;
176                     if(addr+size > maddr+mp->size)
177                             break;
178                     maddr = addr;
179             }
180
181             if(align > 0)
182                     maddr = ((maddr+align-1)/align)*align;
183             if(mp->addr+mp->size-maddr < size)
184                     continue;
185
186             oaddr = mp->addr;
187             mp->addr = maddr+size;
188             mp->size -= maddr-oaddr+size;
189             if(mp->size == 0){
190                     do{
191                             mp++;
192                             (mp-1)->addr = mp->addr;
193                     }while((mp-1)->size = mp->size);
194             }
195
196             unlock(rmap);
197             if(oaddr != maddr)
198                     mapfree(rmap, oaddr, maddr-oaddr);
199
200             return maddr;
201     }
202     unlock(rmap);
203
204     return 0;
205   }
206
207   static void
208   umbscan(void)
209   {
210     uchar *p;
211
212     /*
213      * Scan the Upper Memory Blocks (0xA0000->0xF0000) for pieces
214      * which aren't used; they can be used later for devices which
215      * want to allocate some virtual address space.
216      * Check for two things:
217      * 1) device BIOS ROM. This should start with a two-byte header
218      *    of 0x55 0xAA, followed by a byte giving the size of the ROM
219      *    in 512-byte chunks. These ROM's must start on a 2KB boundary.
220      * 2) device memory. This is read-write.
221      * There are some assumptions: there's VGA memory at 0xA0000 and
222      * the VGA BIOS ROM is at 0xC0000. Also, if there's no ROM signature
223      * at 0xE0000 then the whole 64KB up to 0xF0000 is theoretically up
224      * for grabs; check anyway.
225      */
226     p = KADDR(0xC0000);
227     while(p < (uchar*)KADDR(0xE0000)){
228             p[0] = 0xCC;
229             p[2*KB-1] = 0xCC;
230             if(p[0] != 0xCC || p[2*KB-1] != 0xCC){
231                     p[0] = 0x55;
232                     p[1] = 0xAA;
233                     p[2] = 4;
234                     if(p[0] == 0x55 && p[1] == 0xAA){
235                             p += p[2]*512;
236                             continue;
237                     }
238                     if(p[0] == 0xFF && p[1] == 0xFF)
239                             mapfree(&rmapumb, PADDR(p), 2*KB);
240             }
241             else
242                     mapfree(&rmapumbrw, PADDR(p), 2*KB);
243             p += 2*KB;
244     }
245
246     p = KADDR(0xE0000);
247     if(p[0] != 0x55 || p[1] != 0xAA){
248             p[0] = 0xCC;
249             p[64*KB-1] = 0xCC;
250             if(p[0] != 0xCC && p[64*KB-1] != 0xCC)
251                     mapfree(&rmapumb, PADDR(p), 64*KB);
252     }
253   }
254
255   static void
256   ramscan(ulong maxmem)
257   {
258     ulong *k0, kzero, map, maxpa, pa, *pte, *table, *va, x;
259     int nvalid[NMemType];
260     uchar *bda;
261
262     /*
263      * The bootstrap code has has created a prototype page
264      * table which maps the first MemMinMB of physical memory to KZERO.
265      * The page directory is at m->pdb and the first page of
266      * free memory is after the per-processor MMU information.
267      */
268     /*
269      * Initialise the memory bank information for conventional memory
270      * (i.e. less than 640KB). The base is the first location after the
271      * bootstrap processor MMU information and the limit is obtained from
272      * the BIOS data area.
```

```
	 */
	x = PADDR(CPU0MACH+BY2PG);
	bda = (uchar*)KADDR(0x400);
	mapfree(&rmapram, x, ((bda[0x14]<<8)|bda[0x13])*KB-x);

	x = PADDR(PGROUND((ulong)end));
	pa = MemMinMB*MB;
	mapfree(&rmapram, x, pa-x);

	/*
	 * Check if the extended memory size can be obtained from the CMOS.
	 * If it's 0 then it's either not known or >= 64MB. Always check
	 * at least 24MB in case there's a memory gap (up to 8MB) below 16MB;
	 * in this case the memory from the gap is remapped to the top of
	 * memory.
	 * The value in CMOS is supposed to be the number of KB above 1MB.
	 */
	if(maxmem == 0){
		x = (nvramread(0x18)<<8)|nvramread(0x17);
		if(x == 0 || x >= (63*KB))
			maxpa = MemMaxMB*MB;
		else
			maxpa = MB+x*KB;
		if(maxpa < 24*MB)
			maxpa = 24*MB;
		maxmem = MemMaxMB*MB;
	}
	else
		maxpa = maxmem;

	/*
	 * March up memory from MemMinMB to maxpa 1MB at a time,
	 * mapping the first page and checking the page can
	 * be written and read correctly. The page tables are created here
	 * on the fly, allocating from low memory as necessary.
	 */
	k0 = (ulong*)KADDR(0);
	kzero = *k0;
	map = 0;
	x = 0x12345678;
	memset(nvalid, 0, sizeof(nvalid));
	while(pa < maxpa){
		/*
		 * Map the page. Use mapalloc(&rmapram, ...) to make
		 * the page table if necessary, it will be returned to the
		 * pool later if it isn't needed.
		 */
		va = KADDR(pa);
		table = &m->pdb[PDX(va)];
		if(*table == 0){
			if(map == 0 && (map = mapalloc(&rmapram, 0, BY2PG, BY2PG)) == 0)
				break;
			memset(KADDR(map), 0, BY2PG);
			*table = map|PTEWRITE|PTEVALID;
			memset(nvalid, 0, sizeof(nvalid));
		}
		table = KADDR(PPN(*table));
		pte = &table[PTX(va)];

		*pte = pa|PTEWRITE|PTEUNCACHED|PTEVALID;
		mmuflushtlb(PADDR(m->pdb));

		/*
		 * Write a pattern to the page and write a different
		 * pattern to a possible mirror at KZER0. If the data
		 * reads back correctly the chunk is some type of RAM (possibly
		 * a linearly-mapped VGA framebuffer, for instance...) and
		 * can be cleared and added to the memory pool. If not, the
		 * chunk is marked uncached and added to the UMB pool if <16MB
		 * or is marked invalid and added to the UPA pool.
		 */
		*va = x;
		*k0 = ~x;
		if(*va == x){
			nvalid[MemRAM] += MB/BY2PG;
			mapfree(&rmapram, pa, MB);

			do{
				*pte++ = pa|PTEWRITE|PTEVALID;
				pa += BY2PG;
			}while(pa % MB);
			mmuflushtlb(PADDR(m->pdb));
//				memset(va, 0, MB);
		}
		else if(pa < 16*MB){
			nvalid[MemUMB] += MB/BY2PG;
			mapfree(&rmapumb, pa, MB);

			do{
				*pte++ = pa|PTEWRITE|PTEUNCACHED|PTEVALID;
				pa += BY2PG;
			}while(pa % MB);
		}
		else{
			nvalid[MemUPA] += MB/BY2PG;
			mapfree(&rmapupa, pa, MB);

			*pte = 0;
			pa += MB;
		}

		/*
		 * Done with this 4MB chunk, review the options:
		 * 1) not physical memory and >=16MB - invalidate the PDB entry;
		 * 2) physical memory - use the 4MB page extension if possible;
		 * 3) not physical memory and <16MB - use the 4MB page extension
		 *    if possible;
		 * 4) mixed or no 4MB page extension - commit the already
		 *    initialised space for the page table.
		 */
		if((pa % (4*MB)) == 0){
			table = &m->pdb[PDX(va)];
			if(nvalid[MemUPA] == (4*MB)/BY2PG)
				*table = 0;
			else if(nvalid[MemRAM] == (4*MB)/BY2PG && (m->cpuiddx & 0x08))
				*table = (pa - 4*MB)|PTESIZE|PTEWRITE|PTEVALID;
			else if(nvalid[MemUMB] == (4*MB)/BY2PG && (m->cpuiddx & 0x08))
				*table = (pa - 4*MB)|PTESIZE|PTEWRITE|PTEUNCACHED|PTEVALID;
			else
				map = 0;
		}

		mmuflushtlb(PADDR(m->pdb));
		x += 0x3141526;
	}
	if(map)
		mapfree(&rmapram, map, BY2PG);
	if(pa < maxmem)
		mapfree(&rmapupa, pa, maxmem-pa);
	if(maxmem < 0xFFE00000)
		mapfree(&rmapupa, maxmem, 0xFFE00000-maxmem);
	if(MEMDEBUG)
		print("maxmem %luX %luX\n", maxmem, 0xFFE00000-maxmem);
	*k0 = kzero;
}

```

```
409  void
410  meminit(ulong maxmem)
411  {
412    Map *mp, *xmp;
413    ulong pa, *pte;
414
415    /*
416     * Set special attributes for memory between 640KB and 1MB:
417     *   VGA memory is writethrough;
418     *   BIOS ROM's/UMB's are uncached;
419     * then scan for useful memory.
420     */
421    for(pa = 0xA0000; pa < 0xC0000; pa += BY2PG){
422            pte = mmuwalk(m->pdb, (ulong)KADDR(pa), 2, 0);
423            *pte |= PTEWT;
424    }
425    for(pa = 0xC0000; pa < 0x100000; pa += BY2PG){
426            pte = mmuwalk(m->pdb, (ulong)KADDR(pa), 2, 0);
427            *pte |= PTEUNCACHED;
428    }
429    mmuflushtlb(PADDR(m->pdb));
430
431    umbscan();
432    ramscan(maxmem);
433
434    /*
435     * Set the conf entries describing two banks of allocatable memory.
436     * Grab the first and largest entries in rmapram as left by ramscan().
437     *
438     * It would be nice to have more than 2 memory banks describable in conf.
439     */
440    mp = rmapram.map;
441    conf.base0 = mp->addr;
442    conf.npage0 = mp->size/BY2PG;
443    mp++;
444    for(xmp = 0; mp->size; mp++){
445            if(xmp == 0 || mp->size > xmp->size)
446                    xmp = mp;
447    }
448
449    if(xmp){
450            conf.base1 = xmp->addr;
451            conf.npage1 = xmp->size/BY2PG;
452    }
453    if(MEMDEBUG)
454            memdebug();
455  }
456
457  ulong
458  umbmalloc(ulong addr, int size, int align)
459  {
460    ulong a;
461
462    if(a = mapalloc(&rmapumb, addr, size, align))
463            return (ulong)KADDR(a);
464
465    return 0;
466  }
467
468  void
469  umbfree(ulong addr, int size)
470  {
471    mapfree(&rmapumb, PADDR(addr), size);
472  }
473
474  ulong
475  umbrwmalloc(ulong addr, int size, int align)
476  {
477    ulong a;
478    uchar *p;
479
480    if(a = mapalloc(&rmapumbrw, addr, size, align))
481            return(ulong)KADDR(a);
482
483    /*
484     * Perhaps the memory wasn't visible before
485     * the interface is initialised, so try again.
486     */
487    if((a = umbmalloc(addr, size, align)) == 0)
488            return 0;
489    p = (uchar*)a;
490    p[0] = 0xCC;
491    p[size-1] = 0xCC;
492    if(p[0] == 0xCC && p[size-1] == 0xCC)
493            return a;
494    umbfree(a, size);
495
496    return 0;
497  }
498
499  void
500  umbrwfree(ulong addr, int size)
501  {
502    mapfree(&rmapumbrw, PADDR(addr), size);
503  }
504
505  ulong
506  upamalloc(ulong pa, int size, int align)
507  {
508    ulong a, ae;
509
510    if(a = mapalloc(&xrmapupa, pa, size, align))
511            return a;
512
513    if((a = mapalloc(&rmapupa, pa, size, align)) == 0){
514            memdebug();
515            return 0;
516    }
517
518    /*
519     * Upamalloc is a request to map a range of physical addresses.
520     * Therefore, if pa is 0 mapalloc will choose the base address.
521     * Note, however, mmukmap is always asked to give a 1-to-1 mapping
522     * of va to pa.
523    ae = mmukmap(a, a, size);
524     * ...but for the moment go back to the old scheme for VLB cards.
525     */
526    ae = mmukmap(a, 0, size);
527
528    /*
529     * Should check here that it was all delivered
530     * and put it back and barf if not.
531     */
532    USED(ae);
533
534    /*
535     * Be very careful this returns a PHYSICAL address
536     * mapped 1-to-1 with the virtual address.
537     * If a < KZERO it's probably not a good idea to
538     * try KADDR(a)...
539     */
540    return a;
541  }
542
543  void
544  upafree(ulong pa, int size)
```

```
545   {
546     mapfree(&xrmapupa, pa, size);
547   }
548
```

```
#include  "u.h"
#include  "../port/lib.h"
#include  "mem.h"
#include  "dat.h"
#include  "fns.h"
#include  "io.h"

#define   DATASEGM(p)     { 0xFFFF, SEGG|SEGB|(0xF<<16)|SEGP|SEGPL(p)|SEGDATA|SEGW }
#define   EXECSEGM(p)     { 0xFFFF, SEGG|SEGD|(0xF<<16)|SEGP|SEGPL(p)|SEGEXEC|SEGR }
#define   TSSSEGM(b,p)    { ((b)<<16)|sizeof(Tss),\
                      ((b)&0xFF000000)|(((b)>>16)&0xFF)|SEGTSS|SEGPL(p)|SEGP }

Segdesc gdt[6] =
{
[NULLSEG] { 0, 0},              /* null descriptor */
[KDSEG]           DATASEGM(0),          /* kernel data/stack */
[KESEG]           EXECSEGM(0),          /* kernel code */
[UDSEG]           DATASEGM(3),          /* user data/stack */
[UESEG]           EXECSEGM(3),          /* user code */
[TSSSEG]  TSSSEGM(0,0),         /* tss segment */
};

#define PDX(va)           ((((ulong)(va))>>22) & 0x03FF)
#define PTX(va)           ((((ulong)(va))>>12) & 0x03FF)

static void
taskswitch(ulong pdb, ulong stack)
{
  Tss *tss;

  tss = m->tss;
  tss->ss0 = KDSEL;
  tss->esp0 = stack;
  tss->ss1 = KDSEL;
  tss->esp1 = stack;
  tss->ss2 = KDSEL;
  tss->esp2 = stack;
  tss->cr3 = pdb;
  putcr3(pdb);
}

void
mmuinit(void)
{
  ulong x, *p;
  ushort ptr[3];

  m->tss = malloc(sizeof(Tss));
  memset(m->tss, 0, sizeof(Tss));

  memmove(m->gdt, gdt, sizeof(m->gdt));
  x = (ulong)m->tss;
  m->gdt[TSSSEG].d0 = (x<<16)|sizeof(Tss);
  m->gdt[TSSSEG].d1 = (x&0xFF000000)|((x>>16)&0xFF)|SEGTSS|SEGPL(0)|SEGP;

  ptr[0] = sizeof(m->gdt);
  x = (ulong)m->gdt;
  ptr[1] = x & 0xFFFF;
  ptr[2] = (x>>16) & 0xFFFF;
  lgdt(ptr);

  ptr[0] = sizeof(Segdesc)*256;
  x = IDTADDR;
  ptr[1] = x & 0xFFFF;
  ptr[2] = (x>>16) & 0xFFFF;
  lidt(ptr);

  /* make kernel text unwritable */
  for(x = KTZERO; x < (ulong)etext; x += BY2PG){
          p = mmuwalk(m->pdb, x, 2, 0);
          if(p == nil)
                  panic("mmuinit");
          *p &= ~PTEWRITE;
  }

  taskswitch(PADDR(m->pdb),  (ulong)m + BY2PG);
  ltr(TSSSEL);
}

void
flushmmu(void)
{
  int s;

  s = splhi();
  up->newtlb = 1;
  mmuswitch(up);
  splx(s);
}

static void
mmuptefree(Proc* proc)
{
  ulong *pdb;
  Page **last, *page;

  if(proc->mmupdb && proc->mmuused){
          pdb = (ulong*)proc->mmupdb->va;
          last = &proc->mmuused;
          for(page = *last; page; page = page->next){
                  pdb[page->daddr] = 0;
                  last = &page->next;
          }
          *last = proc->mmufree;
          proc->mmufree = proc->mmuused;
          proc->mmuused = 0;
  }
}

void
mmuswitch(Proc* proc)
{
  ulong *pdb;

  if(proc->newtlb){
          mmuptefree(proc);
          proc->newtlb = 0;
  }

  if(proc->mmupdb){
          pdb = (ulong*)proc->mmupdb->va;
          pdb[PDX(MACHADDR)] = m->pdb[PDX(MACHADDR)];
          taskswitch(proc->mmupdb->pa, (ulong)(proc->kstack+KSTACK));
  }
  else
          taskswitch(PADDR(m->pdb), (ulong)(proc->kstack+KSTACK));
}

void
mmurelease(Proc* proc)
{
  Page *page, *next;

  /*
   * Release any pages allocated for a page directory base or page-tables
   * for this process:
```

```
137      *   switch to the prototype pdb for this processor (m->pdb);
138      *   call mmuptefree() to place all pages used for page-tables (proc->mmuused)
139      *   onto the process' free list (proc->mmufree). This has the side-effect of
140      *   cleaning any user entries in the pdb (proc->mmupdb);
141      *   if there's a pdb put it in the cache of pre-initialised pdb's
142      *   for this processor (m->pdbpool) or on the process' free list;
143      *   finally, place any pages freed back into the free pool (palloc).
144      * This routine is only called from sched() with palloc locked.
145      */
146     taskswitch(PADDR(m->pdb), (ulong)m + BY2PG);
147     mmuptefree(proc);
148
149     if(proc->mmupdb){
150             if(m->pdbcnt > 10){
151                     proc->mmupdb->next = proc->mmufree;
152                     proc->mmufree = proc->mmupdb;
153             }
154             else{
155                     proc->mmupdb->next = m->pdbpool;
156                     m->pdbpool = proc->mmupdb;
157                     m->pdbcnt++;
158             }
159             proc->mmupdb = 0;
160     }
161
162     for(page = proc->mmufree; page; page = next){
163             next = page->next;
164             if(--page->ref)
165                     panic("mmurelease: page->ref %d\n", page->ref);
166             pagechainhead(page);
167     }
168     if(proc->mmufree && palloc.r.p)
169             wakeup(&palloc.r);
170     proc->mmufree = 0;
171   }
172
173   static Page*
174   mmupdballoc(void)
175   {
176     int s;
177     Page *page;
178
179     s = splhi();
180     if(m->pdbpool == 0){
181             spllo();
182             page = newpage(0, 0, 0);
183             page->va = VA(kmap(page));
184             memmove((void*)page->va, m->pdb, BY2PG);
185     }
186     else{
187             page = m->pdbpool;
188             m->pdbpool = page->next;
189             m->pdbcnt--;
190     }
191     splx(s);
192     return page;
193   }
194
195   void
196   putmmu(ulong va, ulong pa, Page*)
197   {
198     int pdbx;
199     Page *page;
200     ulong *pdb, *pte;
201     int s;
202
203     if(up->mmupdb == 0)
204             up->mmupdb = mmupdballoc();
205     pdb = (ulong*)up->mmupdb->va;
206     pdbx = PDX(va);
207
208     if(PPN(pdb[pdbx]) == 0){
209             if(up->mmufree == 0){
210                     page = newpage(1, 0, 0);
211                     page->va = VA(kmap(page));
212             }
213             else {
214                     page = up->mmufree;
215                     up->mmufree = page->next;
216                     memset((void*)page->va, 0, BY2PG);
217             }
218             pdb[pdbx] = PPN(page->pa)|PTEUSER|PTEWRITE|PTEVALID;
219             page->daddr = pdbx;
220             page->next = up->mmuused;
221             up->mmuused = page;
222     }
223
224     pte = KADDR(PPN(pdb[pdbx]));
225     pte[PTX(va)] = pa|PTEUSER;
226
227     s = splhi();
228     pdb[PDX(MACHADDR)] = m->pdb[PDX(MACHADDR)];
229     mmuflushtlb(up->mmupdb->pa);
230     splx(s);
231   }
232
233   ulong*
234   mmuwalk(ulong* pdb, ulong va, int level, int create)
235   {
236     ulong pa, *table;
237
238     /*
239      * Walk the page-table pointed to by pdb and return a pointer
240      * to the entry for virtual address va at the requested level.
241      * If the entry is invalid and create isn't requested then bail
242      * out early. Otherwise, for the 2nd level walk, allocate a new
243      * page-table page and register it in the 1st level.
244      */
245     table = &pdb[PDX(va)];
246     if(!(*table & PTEVALID) && create == 0)
247             return 0;
248
249     switch(level){
250
251     default:
252             return 0;
253
254     case 1:
255             return table;
256
257     case 2:
258             if(*table & PTESIZE)
259                     panic("mmuwalk2: va %uX entry %uX\n", va, *table);
260             if(!(*table & PTEVALID)){
261                     pa = PADDR(xspanalloc(BY2PG, BY2PG, 0));
262                     *table = pa|PTEWRITE|PTEVALID;
263             }
264             table = KADDR(PPN(*table));
265
266             return &table[PTX(va)];
267     }
268   }
269
270   static Lock mmukmaplock;
271
272   int
```

```
mmukmapsync(ulong va)
{
  Mach *mach0;
  ulong entry, *pte;

  mach0 = MACHP(0);

  lock(&mmukmaplock);

  if((pte = mmuwalk(mach0->pdb, va, 1, 0)) == nil){
          unlock(&mmukmaplock);
          return 0;
  }
  if(!(*pte & PTESIZE) && mmuwalk(mach0->pdb, va, 2, 0) == nil){
          unlock(&mmukmaplock);
          return 0;
  }
  entry = *pte;

  if(!(m->pdb[PDX(va)] & PTEVALID))
          m->pdb[PDX(va)] = entry;

  if(up && up->mmupdb){
           ((ulong*)up->mmupdb->va)[PDX(va)] = entry;
          mmuflushtlb(up->mmupdb->pa);
  }
  else
          mmuflushtlb(PADDR(m->pdb));

  unlock(&mmukmaplock);

  return 1;
}

ulong
mmukmap(ulong pa, ulong va, int size)
{
  Mach *mach0;
  ulong ova, pae, *table, pgsz, *pte, x;
  int pse, sync;

  mach0 = MACHP(0);
  if((mach0->cpuiddx & 0x08) && (getcr4() & 0x10))
          pse = 1;
  else
          pse = 0;
  sync = 0;

  pa = PPN(pa);
  if(va == 0)
          va = (ulong)KADDR(pa);
  else
          va = PPN(va);
  ova = va;

  pae = pa + size;
  lock(&mmukmaplock);
  while(pa < pae){
          table = &mach0->pdb[PDX(va)];
          /*
           * Possibly already mapped.
           */
          if(*table & PTEVALID){
                  if(*table & PTESIZE){
                          /*
                           * Big page. Does it fit within?
                           * If it does, adjust pgsz so the correct end can be
                           * returned and get out.
                           * If not, adjust pgsz up to the next 4MB boundary
                           * and continue.
                           */
                          x = PPN(*table);
                          if(x != pa)
                                  panic("mmukmap1: pa %uX  entry %uX\n",
                                          pa, *table);
                          x += 4*MB;
                          if(pae <= x){
                                  pa = pae;
                                  break;
                          }
                          pgsz = x - pa;
                          pa += pgsz;
                          va += pgsz;

                          continue;
                  }
                  else{
                          /*
                           * Little page. Walk to the entry.
                           * If the entry is valid, set pgsz and continue.
                           * If not, make it so, set pgsz, sync and continue.
                           */
                          pte = mmuwalk(mach0->pdb, va, 2, 0);
                          if(pte && *pte & PTEVALID){
                                  x = PPN(*pte);
                                  if(x != pa)
                                          panic("mmukmap2: pa %uX entry %uX\n",
                                                  pa, *pte);
                                  pgsz = BY2PG;
                                  pa += pgsz;
                                  va += pgsz;
                                  sync++;

                                  continue;
                          }
                  }
          }

          /*
           * Not mapped. Check if it can be mapped using a big page -
           * starts on a 4MB boundary, size >= 4MB and processor can do it.
           * If not a big page, walk the walk, talk the talk.
           * Sync is set.
           */
          if(pse && (pa % (4*MB)) == 0 && (pae >= pa+4*MB)){
                  *table = pa|PTESIZE|PTEWRITE|PTEUNCACHED|PTEVALID;
                  pgsz = 4*MB;
          }
          else{
                  pte = mmuwalk(mach0->pdb, va, 2, 1);
                  *pte = pa|PTEWRITE|PTEUNCACHED|PTEVALID;
                  pgsz = BY2PG;
          }
          pa += pgsz;
          va += pgsz;
          sync++;
  }
  unlock(&mmukmaplock);

  /*
   * If something was added
   * then need to sync up.
   */
  if(sync)
          mmukmapsync(ova);

```

```
409     return pa;
410   }
```

```
/*
 *  PC specific code for the ns16552.  It includes support for the 2 built
 *  in uarts plus up to 5 MP-008 8 uart ISA cards.
 */
enum
{
  Maxcard= 5,             /* max serial cards */
  UartFREQ= 1843200,

  Serial= 0,
  Modem=  1,
};

#define uartwrreg(u,r,v)  outb((u)->port + r, (u)->sticky[r] | (v))
#define uartrdreg(u,r)            inb((u)->port + r)

void      ns16552setup(ulong, ulong, char*, int);
void      ns16552special(int, int, Queue**, Queue**, int (*)(Queue*, int));
void      uartclock(void);
/*
 *  definition of an optional serial card
 */
typedef struct Scard
{
  ISAConf;        /* card configuration */
  int     first;  /* number of first port */
} Scard;
static Scard *scard[Maxcard];     /* configs for the serial card */

/* power management currently only makes sense on the AT&T safari */
static void
uartpower(int dev, int onoff)
{
  switch(dev){
  case Modem:
          if((*arch->modempower)(onoff) < 0)
                  print("can't turn %s modem speaker\n", onoff?"on":"off");
          break;
  case Serial:
          if((*arch->serialpower)(onoff) < 0)
                  print("can't turn %s serial port power\n", onoff?"on":"off");
          break;
  }
}

/*
 *  handle an interrupt to a single uart
 */
static void
ns16552intrx(Ureg*, void* arg)
{
  ns16552intr((ulong)arg);
}

/*
 *  interrupts from the multiport card, MP-008.  A polling port
 *  tells which of 8 devices interrupted.
 */
static void
mp008intr(Ureg* ureg, void* arg)
{
  int i, loops;
  uchar n;
  Scard *mp;

  mp = arg;
  for(loops = 0; loops < 1024; loops++){
          n = ~inb(mp->mem);
          if(n == 0)
                  return;
          for(i = 0; n; i++){
                  if(n & 1)
                          ns16552intrx(ureg, (void*)(mp->first+i));
                  n >>= 1;
          }
    }
}

/*
 *  install the uarts (called by reset)
 */
void
ns16552install(void)
{
  int i, j, port, nscard;
  char *p, *q;
  Scard *sc;
  char name[NAMELEN];
  static int already;

  if(already)
          return;
  already = 1;

  /* first two ports are always there and always the normal frequency */
  if(ioalloc(0x3f8, 8, 0, "eia0") < 0)
          print("eia0: port %d in use\n", 0x3f8);
  ns16552setup(0x3F8, UartFREQ, "eia0", Ns550);
  intrenable(IrqUART0, ns16552intrx, (void*)0, BUSUNKNOWN, "eia0");
  if(ioalloc(0x2F8, 8, 0, "eia1") < 0)
          print("eia1: port %d in use\n", 0x2F8);
  ns16552setup(0x2F8, UartFREQ, "eia1", Ns550);
  intrenable(IrqUART1, ns16552intrx, (void*)1, BUSUNKNOWN, "eia1");
  addclock0link(uartclock);

  /* set up a serial console */
  if(p = getconf("console")){
          port = strtol(p, &q, 0);
          if(p != q && (port == 0 || port == 1))
                  ns16552special(port, 9600, &kbdq, &printq, kbdcr2nl);
  }

  /* the rest come out of plan9.ini */
  nscard = 0;
  for(i = 0; i < Maxcard; i++){
          sc = scard[nscard] = xalloc(sizeof(Scard));
          if(isaconfig("serial", i, sc) == 0){
                  xfree(sc);
                  scard[nscard] = 0;
                  continue;
          }

          snprint(name, sizeof name, "eia%d00", nscard);
          if(cistrcmp(sc->type, "MP008") == 0){
                  /*
                   *  port gives base port address for uarts
                   *  irq is interrupt
                   *  mem is the polling port
                   *  size is the number of serial ports on the same polling port
                   *  freq is the baud rate generator frequency
                   */
                  if(sc->size == 0)
                          sc->size = 8;
                  if(sc->freq == 0)
                          sc->freq = UartFREQ;
                  sc->first = nuart;
```

```
137                     if(ioalloc(sc->port, 8*sc->size, 0, name) < 0){
138                             print("mp008: port %lud in use\n", sc->port);
139                             xfree(sc);
140                             scard[nscard] = 0;
141                             continue;
142                     }
143                     intrenable(sc->irq, mp008intr, sc, BUSUNKNOWN, name);
144                     port = sc->port;
145                     for(j=0; j < sc->size; j++){
146                             sprint(name, "eia%d%2.2d", nscard, j);
147                             ns16552setup(port, sc->freq, name, Ns550);
148                             port += 8;
149                     }
150             } else if(cistrcmp(sc->type, "turbo650") == 0){
151                     /*
152                      *  port gives base port address for the uart
153                      *  irq is interrupt
154                      *  freq is the baud rate generator frequency
155                      */
156                     if(sc->freq == 0)
157                             sc->freq = UartFREQ*4;
158                     if(ioalloc(sc->port, 8, 0, name) < 0){
159                             print("%s: port %lud in use\n", name, sc->port);
160                             xfree(sc);
161                             scard[nscard] = 0;
162                             continue;
163                     }
164                     ns16552setup(sc->port, sc->freq, name, Ns650);
165
166                     /*
167                      *  multiply clock speed by 4
168                      */
169                     if(sc->mem == 0)
170                             outb(0x2c8, 0);
171                     else
172                             outb(sc->mem, 0);
173
174                     intrenable(sc->irq, ns16552intrx, (void*)(nuart-1), BUSUNKNOWN, name);
175             } else if(cistrcmp(sc->type, "com") == 0 && sc->port != 0x3F8 && sc->port != 0x2F8)
176                     /*
177                      *  port gives base port address for the uart
178                      *  irq is interrupt
179                      *  freq is the baud rate generator frequency
180                      */
181                     if(sc->freq == 0)
182                             sc->freq = UartFREQ;
183                     if(ioalloc(sc->port, 8, 0, name) < 0){
184                             print("%s: port %lud in use\n", name, sc->port);
185                             xfree(sc);
186                             scard[nscard] = 0;
187                             continue;
188                     }
189                     ns16552setup(sc->port, sc->freq, name, Ns550);
190                     intrenable(sc->irq, ns16552intrx, (void*)(nuart-1), BUSUNKNOWN, name);
191             }
192             nscard++;
193     }
194   }
```

```
1     /*
2      * Attach segment types
3      */
4     static Physseg physseg[10] = {
5       { SG_SHARED,    "shared",       0,      SEGMAXSIZE,     0,      0 },
6       { SG_BSS,       "memory",       0,      SEGMAXSIZE,     0,      0 },
7       { 0,            0,              0,      0,              0,      0 },
8     };
```

```
#include  "u.h"
#include  "../port/lib.h"
#include  "mem.h"
#include  "dat.h"
#include  "fns.h"
#include  "io.h"
#include  "ureg.h"
#include  "../port/error.h"

void      noted(Ureg*, ulong);

static void debugbpt(Ureg*, void*);
static void fault386(Ureg*, void*);

static Lock vctllock;
static Vctl *vctl[256];

void
intrenable(int irq, void (*f)(Ureg*, void*), void* a, int tbdf, char *name)
{
  int vno;
  Vctl *v, *p;

  v = xalloc(sizeof(Vctl));
  v->isintr = 1;
  v->irq = irq;
  v->tbdf = tbdf;
  v->f = f;
  v->a = a;
  strncpy(v->name, name, NAMELEN-1);
  v->name[NAMELEN-1] = 0;

  ilock(&vctllock);
  vno = arch->intrenable(v);
  if(vno == -1){
          iunlock(&vctllock);
          print("intrenable: couldn't enable irq %d, tbdf 0x%uX for %s\n",
                  irq, tbdf, v->name);
          if(p=vctl[vno]){
                  print("intrenable: irq %d is already used by", irq);
                  for(; p; p=p->next)
                          print(" %s", p->name);
                  print("\n");
          }
          xfree(v);
          return;
  }
  if(vctl[vno]){
          if(vctl[vno]->isr != v->isr || vctl[vno]->eoi != v->eoi)
                  panic("intrenable: handler: %s %s %luX %luX %luX %luX\n",
                          vctl[vno]->name, v->name,
                          vctl[vno]->isr, v->isr, vctl[vno]->eoi, v->eoi);
          v->next = vctl[vno];
  }
  vctl[vno] = v;
  iunlock(&vctllock);
}

int
irqallocread(char *buf, long n, vlong offset)
{
  int vno;
  Vctl *v;
  long oldn;
  char str[11+1+NAMELEN+1], *p;
  int m;

  if(n < 0 || offset < 0)
          error(Ebadarg);

  oldn = n;
  for(vno=0; vno<nelem(vctl); vno++){
          for(v=vctl[vno]; v; v=v->next){
                  m = snprint(str, sizeof str, "%11d %11d %.*s\n", vno, v->irq, NAMELEN, v->n
                  if(m <= offset) /* if do not want this, skip entry */
                          offset -= m;
                  else{
                          /* skip offset bytes */
                          m -= offset;
                          p = str+offset;
                          offset = 0;

                          /* write at most max(n,m) bytes */
                          if(m > n)
                                  m = n;
                          memmove(buf, p, m);
                          n -= m;
                          buf += m;

                          if(n == 0)
                                  return oldn;
                  }
          }
  }
  return oldn - n;
}

void
trapenable(int vno, void (*f)(Ureg*, void*), void* a, char *name)
{
  Vctl *v;

  if(vno < 0 || vno >= VectorPIC)
          panic("trapenable: vno %d\n", vno);
  v = xalloc(sizeof(Vctl));
  v->tbdf = BUSUNKNOWN;
  v->f = f;
  v->a = a;
  strncpy(v->name, name, NAMELEN);
  v->name[NAMELEN-1] = 0;

  lock(&vctllock);
  if(vctl[vno])
          v->next = vctl[vno]->next;
  vctl[vno] = v;
  unlock(&vctllock);
}

static void
nmienable(void)
{
  int x;

  /*
   * Hack: should be locked with NVRAM access.
   */
  outb(0x70, 0x80);               /* NMI latch clear */
  outb(0x70, 0);

  x = inb(0x61) & 0x07;           /* Enable NMI */
  outb(0x61, 0x08|x);
  outb(0x61, x);
}

void
trapinit(void)
```

```
{
  int d1, v;
  ulong vaddr;
  Segdesc *idt;

  idt = (Segdesc*)IDTADDR;
  vaddr = (ulong)vectortable;
  for(v = 0; v < 256; v++){
          d1 = (vaddr & 0xFFFF0000)|SEGP;
          switch(v){

          case VectorBPT:
                  d1 |= SEGPL(3)|SEGIG;
                  break;

          case VectorSYSCALL:
                  d1 |= SEGPL(3)|SEGIG;
                  break;

          default:
                  d1 |= SEGPL(0)|SEGIG;
                  break;
          }
          idt[v].d0 = (vaddr & 0xFFFF)|(KESEL<<16);
          idt[v].d1 = d1;
          vaddr += 6;
  }

  /*
   * Special traps.
   * Syscall() is called directly without going through trap().
   */
  trapenable(VectorBPT, debugbpt, 0, "debugpt");
  trapenable(VectorPF, fault386, 0, "fault386");

  nmienable();
}

static char* excname[32] = {
  "divide error",
  "debug exception",
  "nonmaskable interrupt",
  "breakpoint",
  "overflow",
  "bounds check",
  "invalid opcode",
  "coprocessor not available",
  "double fault",
  "coprocessor segment overrun",
  "invalid TSS",
  "segment not present",
  "stack exception",
  "general protection violation",
  "page fault",
  "15 (reserved)",
  "coprocessor error",
  "alignment check",
  "machine check",
  "19 (reserved)",
  "20 (reserved)",
  "21 (reserved)",
  "22 (reserved)",
  "23 (reserved)",
  "24 (reserved)",
  "25 (reserved)",
  "26 (reserved)",
  "27 (reserved)",
  "28 (reserved)",
  "29 (reserved)",
  "30 (reserved)",
  "31 (reserved)",
};

/*
 *  All traps come here.  It is slower to have all traps call trap()
 *  rather than directly vectoring the handler.  However, this avoids a
 *  lot of code duplication and possible bugs.  The only exception is
 *  VectorSYSCALL.
 *  Trap is called with interrupts disabled via interrupt-gates.
 */
void
trap(Ureg* ureg)
{
  int i, vno, user;
  char buf[ERRLEN];
  Vctl *ctl, *v;
  Mach *mach;

  m->intrts = fastticks(nil);
  user = 0;
  if((ureg->cs & 0xFFFF) == UESEL){
          user = 1;
          up->dbgreg = ureg;
  }

  vno = ureg->trap;
  if(ctl = vctl[vno]){
          if(ctl->isintr){
                  m->intr++;
                  if(vno >= VectorPIC && vno != VectorSYSCALL)
                          m->lastintr = ctl->irq;
          }

          if(ctl->isr)
                  ctl->isr(vno);
          for(v = ctl; v != nil; v = v->next){
                  if(v->f)
                          v->f(ureg, v->a);
          }
          if(ctl->eoi)
                  ctl->eoi(vno);

          /*
           *  preemptive scheduling.  to limit stack depth,
           *  make sure process has a chance to return from
           *  the current interrupt before being preempted a
           *  second time.
           */
          if(ctl->isintr && ctl->irq != IrqTIMER && ctl->irq != IrqCLOCK)
          if(up && up->state == Running)
          if(anyhigher())
          if(up->preempted == 0)
          if(!active.exiting){
                  up->preempted = 1;
                  sched();
                  splhi();
                  up->preempted = 0;
                  return;
          }
  }
  else if(vno <= nelem(excname) && user){
          spllo();
          sprint(buf, "sys: trap: %s", excname[vno]);
          postnote(up, 1, buf, NDebug);
  }
  else if(vno >= VectorPIC && vno != VectorSYSCALL){
```

```
273             /*
274              * An unknown interrupt.
275              * Check for a default IRQ7. This can happen when
276              * the IRQ input goes away before the acknowledge.
277              * In this case, a 'default IRQ7' is generated, but
278              * the corresponding bit in the ISR isn't set.
279              * In fact, just ignore all such interrupts.
280              */
281             print("cpu%d: spurious interrupt %d, last %d",
282                     m->machno, vno, m->lastintr);
283             for(i = 0; i < 32; i++){
284                     if(!(active.machs & (1<<i)))
285                             continue;
286                     mach = MACHP(i);
287                     if(m->machno == mach->machno)
288                             continue;
289                     print(": cpu%d: last %d", mach->machno, mach->lastintr);
290             }
291             print("\n");
292             m->spuriousintr++;
293             return;
294     }
295     else{
296             if(vno == VectorNMI){
297                     nmienable();
298                     if(m->machno != 0){
299                             print("cpu%d: PC %8.8luX\n",
300                                     m->machno, ureg->pc);
301                             for(;;);
302                     }
303             }
304             dumpregs(ureg);
305             if(vno < nelem(excname))
306                     panic("%s", excname[vno]);
307             panic("unknown trap/intr: %d\n", vno);
308     }
309
310     if(user && (up->procctl || up->nnote)){
311             splhi();
312             notify(ureg);
313     }
314   }
315
316   /*
317    *  dump registers
318    */
319   void
320   dumpregs2(Ureg* ureg)
321   {
322     if(up)
323             print("cpu%d: registers for %s %lud\n",
324                     m->machno, up->text, up->pid);
325     else
326             print("cpu%d: registers for kernel\n", m->machno);
327     print("FLAGS=%luX TRAP=%luX ECODE=%luX PC=%luX",
328             ureg->flags, ureg->trap, ureg->ecode, ureg->pc);
329     print(" SS=%4.4luX USP=%luX\n", ureg->ss & 0xFFFF, ureg->usp);
330     print("  AX %8.8luX  BX %8.8luX  CX %8.8luX  DX %8.8luX\n",
331             ureg->ax, ureg->bx, ureg->cx, ureg->dx);
332     print("  SI %8.8luX  DI %8.8luX  BP %8.8luX\n",
333             ureg->si, ureg->di, ureg->bp);
334     print("  CS %4.4luX  DS %4.4luX  ES %4.4luX  FS %4.4luX  GS %4.4luX\n",
335             ureg->cs & 0xFFFF, ureg->ds & 0xFFFF, ureg->es & 0xFFFF,
336             ureg->fs & 0xFFFF, ureg->gs & 0xFFFF);
337   }
338
339   void
340   dumpregs(Ureg* ureg)
341   {
342     extern ulong etext;
343     vlong mca, mct;
344
345     dumpregs2(ureg);
346
347     /*
348      * Processor control registers.
349      * If machine check exception, time stamp counter, page size extensions
350      * or enhanced virtual 8086 mode extensions are supported, there is a
351      * CR4. If there is a CR4 and machine check extensions, read the machine
352      * check address and machine check type registers if RDMSR supported.
353      */
354     print("  CR0 %8.8lux CR2 %8.8lux CR3 %8.8lux",
355             getcr0(), getcr2(), getcr3());
356     if(m->cpuiddx & 0x9A){
357             print(" CR4 %8.8lux", getcr4());
358             if((m->cpuiddx & 0xA0) == 0xA0){
359                     rdmsr(0x00, &mca);
360                     rdmsr(0x01, &mct);
361                     print("\n  MCA %8.8llux MCT %8.8llux", mca, mct);
362             }
363     }
364     print("\n  ur %lux up %lux\n", ureg, up);
365   }
366
367
368   /*
369    * Fill in enough of Ureg to get a stack trace, and call a function.
370    * Used by debugging interface rdb.
371    */
372   void
373   callwithureg(void (*fn)(Ureg*))
374   {
375     Ureg ureg;
376     ureg.pc = getcallerpc(&fn);
377     ureg.sp = (ulong)&fn;
378     fn(&ureg);
379   }
380
381   static void
382   _dumpstack(Ureg *ureg)
383   {
384     ulong l, v, i;
385     uchar *p;
386     extern ulong etext;
387
388     if(up == 0)
389             return;
390
391     print("ktrace /kernel/path %.8lux %.8lux\n", ureg->pc, ureg->sp);
392     i = 0;
393     for(l=(ulong)&l; l<(ulong)(up->kstack+KSTACK); l+=4){
394             v = *(ulong*)l;
395             if(KTZERO < v && v < (ulong)&etext){
396                     /*
397                      * Pick off general CALL (0xE8) and CALL indirect
398                      * through AX (0xFFD0).
399                      */
400                     p = (uchar*)v;
401                     if(*(p-5) == 0xE8
402                     || (*(p-2) == 0xFF && *(p-1) == 0xD0)){
403                             print("%.8lux=%.8lux ", l, v);
404                             i++;
405                     }
406             }
407             if(i == 4){
408                     i = 0;
```

```
                print("\n");
        }
    }
    if(i)
        print("\n");
}

void
dumpstack(void)
{
    callwithureg(_dumpstack);
}

static void
debugbpt(Ureg* ureg, void*)
{
    char buf[ERRLEN];

    if(up == 0)
        panic("kernel bpt");
    /* restore pc to instruction that caused the trap */
    ureg->pc--;
    sprint(buf, "sys: breakpoint");
    postnote(up, 1, buf, NDebug);
}

static void
fault386(Ureg* ureg, void*)
{
    ulong addr;
    int read, user, n, insyscall;
    char buf[ERRLEN];

    addr = getcr2();
    user = (ureg->cs & 0xFFFF) == UESEL;
    if(!user && mmukmapsync(addr))
        return;
    read = !(ureg->ecode & 2);
    insyscall = up->insyscall;
    up->insyscall = 1;
    n = fault(addr, read);
    if(n < 0){
        if(!user){
            dumpregs(ureg);
            panic("fault: 0x%lux\n", addr);
        }
        sprint(buf, "sys: trap: fault %s addr=0x%lux",
            read? "read" : "write", addr);
        postnote(up, 1, buf, NDebug);
    }
    up->insyscall = insyscall;
}

/*
 *  system calls
 */
#include "../port/systab.h"

/*
 *  Syscall is called directly from assembler without going through trap().
 */
void
syscall(Ureg* ureg)
{
    ulong   sp;
    long    ret;
    int     i, scallnr;

    if((ureg->cs & 0xFFFF) != UESEL)
        panic("syscall: cs 0x%4.4uX\n", ureg->cs);

    m->syscall++;
    up->insyscall = 1;
    up->pc = ureg->pc;
    up->dbgreg = ureg;

    scallnr = ureg->ax;
    up->scallnr = scallnr;
    if(scallnr == RFORK && up->fpstate == FPactive){
        fpsave(&up->fpsave);
        up->fpstate = FPinactive;
    }
    spllo();

    sp = ureg->usp;
    up->nerrlab = 0;
    ret = -1;
    if(!waserror()){
        if(scallnr >= nsyscall){
            pprint("bad sys call number %d pc %lux\n",
                scallnr, ureg->pc);
            postnote(up, 1, "sys: bad sys call", NDebug);
            error(Ebadarg);
        }

        if(sp<(USTKTOP-BY2PG) || sp>(USTKTOP-sizeof(Sargs)-BY2WD))
            validaddr(sp, sizeof(Sargs)+BY2WD, 0);

        up->s = *((Sargs*)(sp+BY2WD));
        up->psstate = sysctab[scallnr];

        ret = systab[scallnr](up->s.args);
        poperror();
    }
    if(up->nerrlab){
        print("bad errstack [%d]: %d extra\n", scallnr, up->nerrlab);
        for(i = 0; i < NERR; i++)
            print("sp=%lux pc=%lux\n",
                up->errlab[i].sp, up->errlab[i].pc);
        panic("error stack");
    }

    up->insyscall = 0;
    up->psstate = 0;

    /*
     *  Put return value in frame.  On the x86 the syscall is
     *  just another trap and the return value from syscall is
     *  ignored.  On other machines the return value is put into
     *  the results register by caller of syscall.
     */
    ureg->ax = ret;

    if(scallnr == NOTED)
        noted(ureg, *(ulong*)(sp+BY2WD));

    if(scallnr!=RFORK && (up->procctl || up->nnote)){
        splhi();
        notify(ureg);
    }
}

/*
 *  Call user, if necessary, with note.
 *  Pass user the Ureg struct and the note on his stack.
 */
```

```
545  int
546  notify(Ureg* ureg)
547  {
548    int l;
549    ulong s, sp;
550    Note *n;
551
552    if(up->procctl)
553            procctl(up);
554    if(up->nnote == 0)
555            return 0;
556
557    s = spllo();
558    qlock(&up->debug);
559    up->notepending = 0;
560    n = &up->note[0];
561    if(strncmp(n->msg, "sys:", 4) == 0){
562            l = strlen(n->msg);
563            if(l > ERRLEN-15)       /* " pc=0x12345678\0" */
564                    l = ERRLEN-15;
565            sprint(n->msg+l, " pc=0x%.8lux", ureg->pc);
566    }
567
568    if(n->flag!=NUser && (up->notified || up->notify==0)){
569            if(n->flag == NDebug)
570                    pprint("suicide: %s\n", n->msg);
571            qunlock(&up->debug);
572            pexit(n->msg, n->flag!=NDebug);
573    }
574
575    if(up->notified) {
576            qunlock(&up->debug);
577            splhi();
578            return 0;
579    }
580
581    if(!up->notify){
582            qunlock(&up->debug);
583            pexit(n->msg, n->flag!=NDebug);
584    }
585    sp = ureg->usp;
586    sp -= sizeof(Ureg);
587
588    if(!okaddr((ulong)up->notify, 1, 0)
589    || !okaddr(sp-ERRLEN-4*BY2WD, sizeof(Ureg)+ERRLEN+4*BY2WD, 1)){
590            pprint("suicide: bad address in notify\n");
591            qunlock(&up->debug);
592            pexit("Suicide", 0);
593    }
594
595    up->ureg = (void*)sp;
596    memmove((Ureg*)sp, ureg, sizeof(Ureg));
597    *(Ureg**)(sp-BY2WD) = up->ureg; /* word under Ureg is old up->ureg */
598    up->ureg = (void*)sp;
599    sp -= BY2WD+ERRLEN;
600    memmove((char*)sp, up->note[0].msg, ERRLEN);
601    sp -= 3*BY2WD;
602    *(ulong*)(sp+2*BY2WD) = sp+3*BY2WD;     /* arg 2 is string */
603    *(ulong*)(sp+1*BY2WD) = (ulong)up->ureg;        /* arg 1 is ureg* */
604    *(ulong*)(sp+0*BY2WD) = 0;              /* arg 0 is pc */
605    ureg->usp = sp;
606    ureg->pc = (ulong)up->notify;
607    up->notified = 1;
608    up->nnote--;
609    memmove(&up->lastnote, &up->note[0], sizeof(Note));
610    memmove(&up->note[0], &up->note[1], up->nnote*sizeof(Note));
611
612    qunlock(&up->debug);
613    splx(s);
614    return 1;
615  }
616
617  /*
618   *   Return user to state before notify()
619   */
620  void
621  noted(Ureg* ureg, ulong arg0)
622  {
623    Ureg *nureg;
624    ulong oureg, sp;
625
626    qlock(&up->debug);
627    if(arg0!=NRSTR && !up->notified) {
628            qunlock(&up->debug);
629            pprint("call to noted() when not notified\n");
630            pexit("Suicide", 0);
631    }
632    up->notified = 0;
633
634    nureg = up->ureg;       /* pointer to user returned Ureg struct */
635
636    /* sanity clause */
637    oureg = (ulong)nureg;
638    if(!okaddr((ulong)oureg-BY2WD, BY2WD+sizeof(Ureg), 0)){
639            pprint("bad ureg in noted or call to noted when not notified\n");
640            qunlock(&up->debug);
641            pexit("Suicide", 0);
642    }
643
644    /*
645     * Check the segment selectors are all valid, otherwise
646     * a fault will be taken on attempting to return to the
647     * user process.
648     * Take care with the comparisons as different processor
649     * generations push segment descriptors in different ways.
650     */
651    if((nureg->cs & 0xFFFF) != UESEL || (nureg->ss & 0xFFFF) != UDSEL
652      || (nureg->ds & 0xFFFF) != UDSEL || (nureg->es & 0xFFFF) != UDSEL
653      || (nureg->fs & 0xFFFF) != UDSEL || (nureg->gs & 0xFFFF) != UDSEL){
654            pprint("bad segment selector in noted\n");
655            qunlock(&up->debug);
656            pexit("Suicide", 0);
657    }
658
659    /* don't let user change system flags */
660    nureg->flags = (ureg->flags & ~0xCD5) | (nureg->flags & 0xCD5);
661
662    memmove(ureg, nureg, sizeof(Ureg));
663
664    switch(arg0){
665    case NCONT:
666    case NRSTR:
667            if(!okaddr(nureg->pc, 1, 0) || !okaddr(nureg->usp, BY2WD, 0)){
668                    qunlock(&up->debug);
669                    pprint("suicide: trap in noted\n");
670                    pexit("Suicide", 0);
671            }
672            up->ureg = (Ureg*)(*(ulong*)(oureg-BY2WD));
673            qunlock(&up->debug);
674            break;
675
676    case NSAVE:
677            if(!okaddr(nureg->pc, BY2WD, 0)
678            || !okaddr(nureg->usp, BY2WD, 0)){
679                    qunlock(&up->debug);
680                    pprint("suicide: trap in noted\n");
```

```
681                     pexit("Suicide", 0);
682             }
683             qunlock(&up->debug);
684             sp = oureg-4*BY2WD-ERRLEN;
685             splhi();
686             ureg->sp = sp;
687             ((ulong*)sp)[1] = oureg;        /* arg 1 0(FP) is ureg* */
688             ((ulong*)sp)[0] = 0;            /* arg 0 is pc */
689             break;
690
691     default:
692             pprint("unknown noted arg 0x%lux\n", arg0);
693             up->lastnote.flag = NDebug;
694             /* fall through */
695
696     case NDFLT:
697             if(up->lastnote.flag == NDebug){
698                     qunlock(&up->debug);
699                     pprint("suicide: %s\n", up->lastnote.msg);
700             } else
701                     qunlock(&up->debug);
702             pexit(up->lastnote.msg, up->lastnote.flag!=NDebug);
703     }
704   }
705
706   long
707   execregs(ulong entry, ulong ssize, ulong nargs)
708   {
709     ulong *sp;
710     Ureg *ureg;
711
712     sp = (ulong*)(USTKTOP - ssize);
713     *--sp = nargs;
714
715     ureg = up->dbgreg;
716     ureg->usp = (ulong)sp;
717     ureg->pc = entry;
718     return USTKTOP-BY2WD;           /* address of user-level clock */
719   }
720
721   ulong
722   userpc(void)
723   {
724     Ureg *ureg;
725
726     ureg = (Ureg*)up->dbgreg;
727     return ureg->pc;
728   }
729
730   /* This routine must save the values of registers the user is not permitted
731    * to write from devproc and then restore the saved values before returning.
732    */
733   void
734   setregisters(Ureg* ureg, char* pureg, char* uva, int n)
735   {
736     ulong flags;
737     ulong cs;
738     ulong ss;
739
740     flags = ureg->flags;
741     cs = ureg->cs;
742     ss = ureg->ss;
743     memmove(pureg, uva, n);
744     ureg->flags = (ureg->flags & 0x00FF) | (flags & 0xFF00);
745     ureg->cs = cs;
746     ureg->ss = ss;
747   }
748
749   static void
750   linkproc(void)
751   {
752     spllo();
753     up->kpfun(up->kparg);
754   }
755
756   void
757   kprocchild(Proc* p, void (*func)(void*), void* arg)
758   {
759     /*
760      * gotolabel() needs a word on the stack in
761      * which to place the return PC used to jump
762      * to linkproc().
763      */
764     p->sched.pc = (ulong)linkproc;
765     p->sched.sp = (ulong)p->kstack+KSTACK-BY2WD;
766
767     p->kpfun = func;
768     p->kparg = arg;
769   }
770
771   void
772   forkchild(Proc *p, Ureg *ureg)
773   {
774     Ureg *cureg;
775
776     /*
777      * Add 2*BY2WD to the stack to account for
778      *  - the return PC
779      *  - trap's argument (ur)
780      */
781     p->sched.sp = (ulong)p->kstack+KSTACK-(sizeof(Ureg)+2*BY2WD);
782     p->sched.pc = (ulong)forkret;
783
784     cureg = (Ureg*)(p->sched.sp+2*BY2WD);
785     memmove(cureg, ureg, sizeof(Ureg));
786     /* return value of syscall in child */
787     cureg->ax = 0;
788
789     /* Things from bottom of syscall which were never executed */
790     p->psstate = 0;
791     p->insyscall = 0;
792   }
793
794   /* Give enough context in the ureg to produce a kernel stack for
795    * a sleeping process
796    */
797   void
798   setkernur(Ureg* ureg, Proc* p)
799   {
800     ureg->pc = p->sched.pc;
801     ureg->sp = p->sched.sp+4;
802   }
803
804   ulong
805   dbgpc(Proc *p)
806   {
807     Ureg *ureg;
808
809     ureg = p->dbgreg;
810     if(ureg == 0)
811             return 0;
812
813     return ureg->pc;
814   }
```

```
#include "u.h"
#include "../port/lib.h"
#include "mem.h"
#include "dat.h"
#include "fns.h"
#include "io.h"
#include "../port/error.h"

#define   Image   IMAGE
#include <draw.h>
#include <memdraw.h>
#include <cursor.h>
#include "screen.h"

static int
s3pageset(VGAscr* scr, int page)
{
  uchar crt35, crt51;
  int opage;

  crt35 = vgaxi(Crtx, 0x35);
  if(scr->gscreen->depth >= 8){
          /*
           * The S3 registers need to be unlocked for this.
           * Let's hope they are already:
           *      vgaxo(Crtx, 0x38, 0x48);
           *      vgaxo(Crtx, 0x39, 0xA0);
           *
           * The page is 6 bits, the lower 4 bits in Crt35<3:0>,
           * the upper 2 in Crt51<3:2>.
           */
          vgaxo(Crtx, 0x35, page & 0x0F);
          crt51 = vgaxi(Crtx, 0x51);
          vgaxo(Crtx, 0x51, (crt51 & ~0x0C)|((page & 0x30)>>2));
          opage = ((crt51 & 0x0C)<<2)|(crt35 & 0x0F);
  }
  else{
          vgaxo(Crtx, 0x35, (page<<2) & 0x0C);
          opage = (crt35>>2) & 0x03;
  }

  return opage;
}

static void
s3page(VGAscr* scr, int page)
{
  int id;

  id = (vgaxi(Crtx, 0x30)<<8)|vgaxi(Crtx, 0x2E);
  switch(id){

  case 0xE110:                            /* ViRGE/GX2 */
          break;

  default:
          lock(&scr->devlock);
          s3pageset(scr, page);
          unlock(&scr->devlock);
          break;
  }
}

static ulong
s3linear(VGAscr* scr, int* size, int* align)
{
  ulong aperture, oaperture;
  int osize, oapsize, wasupamem;
  Pcidev *p;
  Physseg seg;

  osize = *size;
  oaperture = scr->aperture;
  oapsize = scr->apsize;
  wasupamem = scr->isupamem;
  if(wasupamem)
          upafree(oaperture, oapsize);
  scr->isupamem = 0;

  if(p = pcimatch(nil, 0x5333, 0)){
          aperture = p->mem[0].bar & ~0x0F;
          *size = p->mem[0].size;
  }
  else
          aperture = 0;

  aperture = upamalloc(aperture, *size, *align);
  if(aperture == 0){
          if(wasupamem && upamalloc(oaperture, oapsize, 0))
                  scr->isupamem = 1;
  }
  else
          scr->isupamem = 1;

  if(oaperture)
          print("warning (BUG): redefinition of aperture does not change s3screen segment\n")
  memset(&seg, 0, sizeof(seg));
  seg.attr = SG_PHYSICAL;
  seg.name = smalloc(NAMELEN);
  snprint(seg.name, NAMELEN, "s3screen");
  seg.pa = aperture;
  seg.size = osize;
  addphysseg(&seg);

  return aperture;
}

static void
s3vsyncactive(void)
{
  /*
   * Hardware cursor information is fetched from display memory
   * during the horizontal blank active time. The 80x chips may hang
   * if the cursor is turned on or off during this period.
   */
  while((vgai(Status1) & 0x08) == 0)
          ;
}

static void
s3disable(VGAscr*)
{
  uchar crt45;

  /*
   * Turn cursor off.
   */
  crt45 = vgaxi(Crtx, 0x45) & 0xFE;
  s3vsyncactive();
  vgaxo(Crtx, 0x45, crt45);
}

static void
s3enable(VGAscr* scr)
{
  int i;
```

```
137     ulong storage;
138
139     s3disable(scr);
140
141     /*
142      * Cursor colours. Set both the CR0[EF] and the colour
143      * stack in case we are using a 16-bit RAMDAC.
144      */
145     vgaxo(Crtx, 0x0E, Pwhite);
146     vgaxo(Crtx, 0x0F, Pblack);
147     vgaxi(Crtx, 0x45);
148
149     for(i = 0; i < 3; i++)
150             vgaxo(Crtx, 0x4A, Pblack);
151     vgaxi(Crtx, 0x45);
152     for(i = 0; i < 3; i++)
153             vgaxo(Crtx, 0x4B, Pwhite);
154
155     /*
156      * Find a place for the cursor data in display memory.
157      * Must be on a 1024-byte boundary.
158      */
159     storage = (scr->gscreen->width*BY2WD*scr->gscreen->r.max.y+1023)/1024;
160     vgaxo(Crtx, 0x4C, (storage>>8) & 0x0F);
161     vgaxo(Crtx, 0x4D, storage & 0xFF);
162     storage *= 1024;
163     scr->storage = storage;
164
165     /*
166      * Enable the cursor in Microsoft Windows format.
167      */
168     vgaxo(Crtx, 0x55, vgaxi(Crtx, 0x55) & ~0x10);
169     s3vsyncactive();
170     vgaxo(Crtx, 0x45, 0x01);
171   }
172
173   static void
174   s3load(VGAscr* scr, Cursor* curs)
175   {
176     uchar *p;
177     int id, opage, x, y;
178
179     /*
180      * Disable the cursor and
181      * set the pointer to the two planes.
182      */
183     s3disable(scr);
184
185     opage = 0;
186     p = KADDR(scr->aperture);
187     id = (vgaxi(Crtx, 0x30)<<8)|vgaxi(Crtx, 0x2E);
188     switch(id){
189
190     case 0xE131:                            /* ViRGE */
191     case 0xE18A:                            /* ViRGE/[DG]X */
192     case 0xE110:                            /* ViRGE/GX2 */
193     case 0xE13D:                            /* ViRGE/VX */
194             p += scr->storage;
195             break;
196
197     default:
198             lock(&scr->devlock);
199             opage = s3pageset(scr, scr->storage>>16);
200             p += (scr->storage & 0xFFFF);
201             break;
202     }
203
204     /*
205      * The cursor is set in Microsoft Windows format (the ViRGE/GX2 no
206      * longer supports the X11 format) which gives the following truth table:
207      *      and xor colour
208      *       0   0  background colour
209      *       0   1  foreground colour
210      *       1   0  current screen pixel
211      *       1   1  NOT current screen pixel
212      * Put the cursor into the top-left of the 64x64 array.
213      *
214      * The cursor pattern in memory is interleaved words of
215      * AND and XOR patterns.
216      */
217     for(y = 0; y < 64; y++){
218             for(x = 0; x < 64/8; x += 2){
219                     if(x < 16/8 && y < 16){
220                             *p++ = ~(curs->clr[2*y + x]|curs->set[2*y + x]);
221                             *p++ = ~(curs->clr[2*y + x+1]|curs->set[2*y + x+1]);
222                             *p++ = curs->set[2*y + x];
223                             *p++ = curs->set[2*y + x+1];
224                     }
225                     else {
226                             *p++ = 0xFF;
227                             *p++ = 0xFF;
228                             *p++ = 0x00;
229                             *p++ = 0x00;
230                     }
231             }
232     }
233
234     switch(id){
235
236     case 0xE131:                            /* ViRGE */
237     case 0xE18A:                            /* ViRGE/[DG]X */
238     case 0xE110:                            /* ViRGE/GX2 */
239     case 0xE13D:                            /* ViRGE/VX */
240             break;
241
242     default:
243             s3pageset(scr, opage);
244             unlock(&scr->devlock);
245             break;
246     }
247
248     /*
249      * Save the cursor hotpoint and enable the cursor.
250      */
251     scr->offset = curs->offset;
252     s3vsyncactive();
253     vgaxo(Crtx, 0x45, 0x01);
254   }
255
256   static int
257   s3move(VGAscr* scr, Point p)
258   {
259     int x, xo, y, yo;
260
261     /*
262      * Mustn't position the cursor offscreen even partially,
263      * or it disappears. Therefore, if x or y is -ve, adjust the
264      * cursor offset instead.
265      * There seems to be a bug in that if the offset is 1, the
266      * cursor doesn't disappear off the left edge properly, so
267      * round it up to be even.
268      */
269     if((x = p.x+scr->offset.x) < 0){
270             xo = -x;
271             xo = ((xo+1)/2)*2;
272             x = 0;
```

```
    }
    else
            xo = 0;
    if((y = p.y+scr->offset.y) < 0){
            yo = -y;
            y = 0;
    }
    else
            yo = 0;

    vgaxo(Crtx, 0x46, (x>>8) & 0x07);
    vgaxo(Crtx, 0x47, x & 0xFF);
    vgaxo(Crtx, 0x49, y & 0xFF);
    vgaxo(Crtx, 0x4E, xo);
    vgaxo(Crtx, 0x4F, yo);
    vgaxo(Crtx, 0x48, (y>>8) & 0x07);

    return 0;
  }

  /*
   * The manual gives byte offsets, but we want ulong offsets, hence /4.
   */
  enum {
    SrcBase = 0xA4D4/4,
    DstBase = 0xA4D8/4,
    Stride = 0xA4E4/4,
    FgrdData = 0xA4F4/4,
    WidthHeight = 0xA504/4,
    SrcXY = 0xA508/4,
    DestXY = 0xA50C/4,
    Command = 0xA500/4,
    SubStat = 0x8504/4,
    FifoStat = 0x850C/4,
  };

  /*
   * Wait for writes to VGA memory via linear aperture to flush.
   */
  enum {Maxloop = 1<<24};
  struct {
    ulong linear;
    ulong fifo;
    ulong idle;
  } waitcount;

  static void
  waitforlinearfifo(VGAscr *scr)
  {
    ulong *mmio;
    long x;
    static ulong nwaitforlinearfifo;
    ulong mask, val;

    switch(scr->id){
    default:
            panic("unknown scr->id in s3 waitforlinearfifo");
    case 0xE131:    /* ViRGE */
    case 0xE13D:    /* ViRGE/VX */
            mask = 0x0F<<6;
            val = 0x08<<6;
            break;
    case 0xE110:    /* ViRGE/GX2 */
            mask = 0x1F<<6;
            val = 0x10<<6;
            break;
    }
    mmio = scr->mmio;
    x = 0;
    while((mmio[FifoStat]&mask) != val && x++ < Maxloop)
            waitcount.linear++;
  }

  static void
  waitforfifo(VGAscr *scr, int entries)
  {
    ulong *mmio;
    long x;
    static ulong nwaitforfifo;

    mmio = scr->mmio;
    x = 0;
    while((mmio[SubStat]&0x1F00) < ((entries+2)<<8) && x++ < Maxloop)
            waitcount.fifo++;
  }

  static void
  waitforidle(VGAscr *scr)
  {
    ulong *mmio;
    long x;

    mmio = scr->mmio;
    x = 0;
    while((mmio[SubStat]&0x3F00) != 0x3000 && x++ < Maxloop)
            waitcount.idle++;
  }

  static int
  hwscroll(VGAscr *scr, Rectangle r, Rectangle sr)
  {
    enum { Bitbltop = 0xCC };       /* copy source */
    ulong *mmio;
    ulong cmd, stride;
    Point dp, sp;
    int did, d;

    d = scr->gscreen->depth;
    did = (d-8)/8;
    cmd = 0x00000020|(Bitbltop<<17)|(did<<2);
    stride = Dx(scr->gscreen->r)*d/8;

    if(r.min.x <= sr.min.x){
            cmd |= 1<<25;
            dp.x = r.min.x;
            sp.x = sr.min.x;
    }else{
            dp.x = r.max.x-1;
            sp.x = sr.max.x-1;
    }

    if(r.min.y <= sr.min.y){
            cmd |= 1<<26;
            dp.y = r.min.y;
            sp.y = sr.min.y;
    }else{
            dp.y = r.max.y-1;
            sp.y = sr.max.y-1;
    }

    mmio = scr->mmio;
    waitforlinearfifo(scr);
    waitforfifo(scr, 7);
    mmio[SrcBase] = scr->aperture;
    mmio[DstBase] = scr->aperture;
    mmio[Stride] = (stride<<16)|stride;
```

```
409     mmio[WidthHeight] = ((Dx(r)-1)<<16)|Dy(r);
410     mmio[SrcXY] = (sp.x<<16)|sp.y;
411     mmio[DestXY] = (dp.x<<16)|dp.y;
412     mmio[Command] = cmd;
413     waitforidle(scr);
414     return 1;
415   }
416
417   static int
418   hwfill(VGAscr *scr, Rectangle r, ulong sval)
419   {
420     enum { Bitbltop = 0xCC };       /* copy source */
421     ulong *mmio;
422     ulong cmd, stride;
423     int did, d;
424
425     d = scr->gscreen->depth;
426     did = (d-8)/8;
427     cmd = 0x16000120|(Bitbltop<<17)|(did<<2);
428     stride = Dx(scr->gscreen->r)*d/8;
429     mmio = scr->mmio;
430     waitforlinearfifo(scr);
431     waitforfifo(scr, 8);
432     mmio[SrcBase] = scr->aperture;
433     mmio[DstBase] = scr->aperture;
434     mmio[DstBase] = scr->aperture;
435     mmio[Stride] = (stride<<16)|stride;
436     mmio[FgrdData] = sval;
437     mmio[WidthHeight] = ((Dx(r)-1)<<16)|Dy(r);
438     mmio[DestXY] = (r.min.x<<16)|r.min.y;
439     mmio[Command] = cmd;
440     waitforidle(scr);
441     return 1;
442   }
443
444   enum {
445     CursorSyncCtl = 0x0D,   /* in Seqx */
446     VsyncHi = 0x80,
447     VsyncLo = 0x40,
448     HsyncHi = 0x20,
449     HsyncLo = 0x10,
450   };
451
452   static void
453   s3blank(int blank)
454   {
455     uchar x;
456
457     x = vgaxi(Seqx, CursorSyncCtl);
458     x &= ~0xF0;
459     if(blank)
460             x |= VsyncLo | HsyncLo;
461     vgaxo(Seqx, CursorSyncCtl, x);
462   }
463
464   static void
465   s3drawinit(VGAscr *scr)
466   {
467     ulong id;
468
469     id = (vgaxi(Crtx, 0x30)<<8)|vgaxi(Crtx, 0x2E);
470     scr->id = id;
471
472     /*
473      * It's highly likely that other ViRGEs will work without
474      * change to the driver, with the exception of the size of
475      * the linear aperture memory write FIFO.  Since we don't
476      * know that size, I'm not turning them on.  See waitforlinearfifo
477      * above.
478      */
479     switch(id){
480     case 0xE131:                            /* ViRGE */
481     case 0xE13D:                            /* ViRGE/VX */
482     case 0xE110:                            /* ViRGE/GX2 */
483             scr->mmio = (ulong*)(scr->aperture+0x1000000);
484             scr->fill = hwfill;
485             scr->scroll = hwscroll;
486             /* scr->blank = hwblank; */
487     }
488   }
489
490   VGAdev vgas3dev = {
491     "s3",
492
493     0,
494     0,
495     s3page,
496     s3linear,
497     s3drawinit,
498   };
499
500   VGAcur vgas3cur = {
501     "s3hwgc",
502
503     s3enable,
504     s3disable,
505     s3load,
506     s3move,
507   };
```

```
#include  "u.h"
#include  "../port/lib.h"
#include  "mem.h"
#include  "dat.h"
#include  "fns.h"

Alarms    alarms;
Rendez    alarmr;
Talarm    talarm;

void
alarmkproc(void*)
{
  Proc *rp;
  ulong now;

  for(;;){
          now = MACHP(0)->ticks;
          qlock(&alarms);
          while((rp = alarms.head) && rp->alarm <= now){
                  if(rp->alarm != 0L){
                          if(canqlock(&rp->debug)){
                                  if(!waserror()){
                                          postnote(rp, 0, "alarm", NUser);
                                          poperror();
                                  }
                                  qunlock(&rp->debug);
                                  rp->alarm = 0L;
                          }else
                                  break;
                  }
                  alarms.head = rp->palarm;
          }
          qunlock(&alarms);

          sleep(&alarmr, return0, 0);
  }
}

/*
 *  called every clock tick
 */
void
checkalarms(void)
{
  Proc *p;
  ulong now;

  p = alarms.head;
  now = MACHP(0)->ticks;

  if(p && p->alarm <= now)
          wakeup(&alarmr);

  if(talarm.list == 0 || !canlock(&talarm))
          return;

  for(;;) {
          p = talarm.list;
          if(p == 0)
                  break;

          if(p->twhen == 0) {
                  talarm.list = p->tlink;
                  p->trend = 0;
                  continue;
          }
          if(now < p->twhen)
                  break;
          wakeup(p->trend);
          talarm.list = p->tlink;
          p->trend = 0;
  }

  unlock(&talarm);
}

ulong
procalarm(ulong time)
{
  Proc **l, *f;
  ulong when, old;

  if(up->alarm)
          old = TK2MS(up->alarm - MACHP(0)->ticks);
  else
          old = 0;
  if(time == 0) {
          up->alarm = 0;
          return old;
  }
  when = MS2TK(time)+MACHP(0)->ticks;

  qlock(&alarms);
  l = &alarms.head;
  for(f = *l; f; f = f->palarm) {
          if(up == f){
                  *l = f->palarm;
                  break;
          }
          l = &f->palarm;
  }

  up->palarm = 0;
  if(alarms.head) {
          l = &alarms.head;
          for(f = *l; f; f = f->palarm) {
                  if(f->alarm > when) {
                          up->palarm = f;
                          *l = up;
                          goto done;
                  }
                  l = &f->palarm;
          }
          *l = up;
  }
  else
          alarms.head = up;
done:
  up->alarm = when;
  qunlock(&alarms);

  return old;
}
```

```
#include  "u.h"
#include  "../port/lib.h"
#include  "mem.h"
#include  "dat.h"
#include  "fns.h"
#include  "error.h"
#include  <pool.h>

static void poolprint(Pool*, char*, ...);
static void ppanic(Pool*, char*, ...);
static void plock(Pool*);
static void punlock(Pool*);

typedef struct Private    Private;
struct Private {
  Lock            lk;
  char            msg[256];       /* a rock for messages to be printed at unlock */
};

static Private pmainpriv;
static Pool pmainmem = {
  .name=  "Main",
  .maxsize=       4*1024*1024,
  .minarena=      128*1024,
  .quantum=       32,
  .alloc= xalloc,
  .merge= xmerge,
  .flags= POOL_TOLERANCE,

  .lock=  plock,
  .unlock=        punlock,
  .print= poolprint,
  .panic= ppanic,

  .private=       &pmainpriv,
};

static Private pimagpriv;
static Pool pimagmem = {
  .name=  "Image",
  .maxsize=       16*1024*1024,
  .minarena=      2*1024*1024,
  .quantum=       32,
  .alloc= xalloc,
  .merge= xmerge,
  .flags= 0,

  .lock=  plock,
  .unlock=        punlock,
  .print= poolprint,
  .panic= ppanic,

  .private=       &pimagpriv,
};

Pool*     mainmem = &pmainmem;
Pool*     imagmem = &pimagmem;

/*
 * because we can't print while we're holding the locks,
 * we have the save the message and print it once we let go.
 */
static void
poolprint(Pool *p, char *fmt, ...)
{
  va_list v;
  Private *pv;

  pv = p->private;
  va_start(v, fmt);
  doprint(pv->msg+strlen(pv->msg), pv->msg+sizeof pv->msg, fmt, v);
  va_end(v);
}

static void
ppanic(Pool *p, char *fmt, ...)
{
  va_list v;
  Private *pv;
  char msg[sizeof pv->msg];

  pv = p->private;
  va_start(v, fmt);
  doprint(pv->msg+strlen(pv->msg), pv->msg+sizeof pv->msg, fmt, v);
  va_end(v);
  memmove(msg, pv->msg, sizeof msg);
  iunlock(&pv->lk);
  panic("%s", msg);
}

static void
plock(Pool *p)
{
  Private *pv;

  pv = p->private;
  ilock(&pv->lk);
  pv->msg[0] = 0;
}

static void
punlock(Pool *p)
{
  Private *pv;
  char msg[sizeof pv->msg];

  pv = p->private;
  if(pv->msg[0] == 0){
          iunlock(&pv->lk);
          return;
  }

  memmove(msg, pv->msg, sizeof msg);
  iunlock(&pv->lk);
  print("%.*s", sizeof pv->msg, msg);
}

void
poolsummary(Pool *p)
{
  print("%s max %lud cur %lud free %lud alloc %lud\n", p->name,
          p->maxsize, p->cursize, p->curfree, p->curalloc);
}

void
mallocsummary(void)
{
  poolsummary(mainmem);
  poolsummary(imagmem);
}

/* everything from here down should be the same in libc, libdebugmalloc, and the kernel */
/* - except the code for malloc(), which alternately doesn't clear or does. */
/* - except the code for smalloc(), which lives only in the kernel. */

/*
```

```
137   * Npadlong is the number of 32-bit longs to leave at the beginning of
138   * each allocated buffer for our own bookkeeping.  We return to the callers
139   * a pointer that points immediately after our bookkeeping area.  Incoming pointers
140   * must be decremented by that much, and outgoing pointers incremented.
141   * The malloc tag is stored at MallocOffset from the beginning of the block,
142   * and the realloc tag at ReallocOffset.  The offsets are from the true beginning
143   * of the block, not the beginning the caller sees.
144   *
145   * The extra if(Npadlong != 0) in various places is a hint for the compiler to
146   * compile out function calls that would otherwise be no-ops.
147   */
148
149  /*        non tracing
150   *
151  enum {
152    Npadlong        = 0,
153    MallocOffset = 0,
154    ReallocOffset = 0,
155  };
156   *
157   */
158
159  /* tracing */
160  enum {
161    Npadlong        = 2,
162    MallocOffset = 0,
163    ReallocOffset = 1
164  };
165
166
167  void*
168  smalloc(ulong size)
169  {
170    void *v;
171
172    for(;;) {
173            v = poolalloc(mainmem, size+Npadlong*sizeof(ulong));
174            if(v != nil)
175                    break;
176            tsleep(&up->sleep, return0, 0, 100);
177    }
178    if(Npadlong){
179            v = (ulong*)v+Npadlong;
180            setmalloctag(v, getcallerpc(&size));
181    }
182    memset(v, 0, size);
183    return v;
184  }
185
186  void*
187  malloc(ulong size)
188  {
189    void *v;
190
191    v = poolalloc(mainmem, size+Npadlong*sizeof(ulong));
192    if(Npadlong && v != nil) {
193            v = (ulong*)v+Npadlong;
194            setmalloctag(v, getcallerpc(&size));
195            setrealloctag(v, 0);
196            memset(v, 0, size);
197    }
198    return v;
199  }
200
201  void*
202  mallocz(ulong size, int clr)
203  {
204    void *v;
205
206    v = poolalloc(mainmem, size+Npadlong*sizeof(ulong));
207    if(Npadlong && v != nil){
208            v = (ulong*)v+Npadlong;
209            setmalloctag(v, getcallerpc(&size));
210            setrealloctag(v, 0);
211    }
212    if(clr && v != nil)
213            memset(v, 0, size);
214    return v;
215  }
216
217  void
218  free(void *v)
219  {
220    if(v != nil)
221            poolfree(mainmem, (ulong*)v-Npadlong);
222  }
223
224  void*
225  realloc(void *v, ulong size)
226  {
227    void *nv;
228
229    if(v != nil)
230            v = (ulong*)v-Npadlong;
231    if(Npadlong !=0 && size != 0)
232            size += Npadlong*sizeof(ulong);
233
234    if(nv = poolrealloc(mainmem, v, size)){
235            nv = (ulong*)nv+Npadlong;
236            setrealloctag(nv, getcallerpc(&v));
237            if(v == nil)
238                    setmalloctag(nv, getcallerpc(&v));
239    }
240    return nv;
241  }
242
243  ulong
244  msize(void *v)
245  {
246    return poolmsize(mainmem, (ulong*)v-Npadlong)-Npadlong*sizeof(ulong);
247  }
248
249  void*
250  calloc(ulong n, ulong szelem)
251  {
252    void *v;
253    if(v = mallocz(n*szelem, 1))
254            setmalloctag(v, getcallerpc(&n));
255    return v;
256  }
257
258  void
259  setmalloctag(void *v, ulong pc)
260  {
261    ulong *u;
262    USED(v, pc);
263    if(Npadlong <= MallocOffset || v == nil)
264            return;
265    u = v;
266    u[-Npadlong+MallocOffset] = pc;
267  }
268
269  void
270  setrealloctag(void *v, ulong pc)
271  {
272    ulong *u;
```

```
273     USED(v, pc);
274     if(Npadlong <= ReallocOffset || v == nil)
275             return;
276     u = v;
277     u[-Npadlong+ReallocOffset] = pc;
278   }
279
280   ulong
281   getmalloctag(void *v)
282   {
283     USED(v);
284     if(Npadlong <= MallocOffset)
285             return ~0;
286     return ((ulong*)v)[-Npadlong+MallocOffset];
287   }
288
289   ulong
290   getrealloctag(void *v)
291   {
292     USED(v);
293     if(Npadlong <= ReallocOffset)
294             return ((ulong*)v)[-Npadlong+ReallocOffset];
295     return ~0;
296   }
```

```
#include  "u.h"
#include  "../port/lib.h"
#include  "mem.h"
#include  "dat.h"
#include  "fns.h"
#include  "error.h"
#include  "pool.h"

enum
{
  Hdrspc          = 64,           /* leave room for high-level headers */
  Bdead           = 0x51494F42,   /* "QIOB" */
};

struct
{
  Lock;
  ulong   bytes;
} ialloc;

/*
 *  allocate blocks, round the data base up to a multiple of BLOCKALIGN.
 */
Block*
allocb(int size)
{
  Block *b;
  ulong addr;
  int n;

  n = sizeof(Block) + size;
  b = malloc(n+Hdrspc);
  if(b == 0)
          exhausted("Blocks");
  memset(b, 0, sizeof(Block));

  /* align start of data portion by rounding up */
  addr = (ulong)b;
  addr = ROUND(addr + sizeof(Block), BLOCKALIGN);
  b->base = (uchar*)addr;

  /* align end of data portion by rounding down */
  b->lim = ((uchar*)b) + msize(b);
  addr = (ulong)(b->lim);
  addr = addr & ~(BLOCKALIGN-1);
  b->lim = (uchar*)addr;

  /* leave sluff at beginning for added headers */
  b->rp = b->lim - ROUND(size, BLOCKALIGN);
  if(b->rp < b->base)
          panic("allocb");
  b->wp = b->rp;
  setmalloctag(b, getcallerpc(&size));

  return b;
}

/*
 *  interrupt time allocation
 */
Block*
iallocb(int size)
{
  Block *b;
  ulong addr;
  int n;

  if(ialloc.bytes > conf.ialloc){
          print("iallocb: limited %lud/%lud\n",
                  ialloc.bytes, conf.ialloc);
          return 0;
  }

  n = sizeof(Block) + size;
  b = malloc(n+Hdrspc);
  if(b == 0){
          print("iallocb: no memory %lud/%lud\n",
                  ialloc.bytes, conf.ialloc);
          return nil;
  }
  memset(b, 0, sizeof(Block));

  /* align start of data portion by rounding up */
  addr = (ulong)b;
  addr = ROUND(addr + sizeof(Block), BLOCKALIGN);
  b->base = (uchar*)addr;

  /* align end of data portion by rounding down */
  b->lim = ((uchar*)b) + msize(b);
  addr = (ulong)(b->lim);
  addr = addr & ~(BLOCKALIGN-1);
  b->lim = (uchar*)addr;

  /* leave sluff at beginning for added headers */
  b->rp = b->lim - ROUND(size, BLOCKALIGN);
  if(b->rp < b->base)
          panic("allocb");
  b->wp = b->rp;

  b->flag = BINTR;

  ilock(&ialloc);
  ialloc.bytes += b->lim - b->base;
  iunlock(&ialloc);
  setmalloctag(b, getcallerpc(&size));

  return b;
}

void
freeb(Block *b)
{
  void *dead = (void*)Bdead;

  if(b == nil)
          return;

  /*
   * drivers which perform non cache coherent DMA manage their own buffer
   * pool of uncached buffers and provide their own free routine.
   */
  if(b->free) {
          b->free(b);
          return;
  }
  if(b->flag & BINTR) {
          ilock(&ialloc);
          ialloc.bytes -= b->lim - b->base;
          iunlock(&ialloc);
  }

  /* poison the block in case someone is still holding onto it */
  b->next = dead;
  b->rp = dead;
  b->wp = dead;
  b->lim = dead;
```

```
137     b->base = dead;
138
139     free(b);
140   }
141
142   void
143   checkb(Block *b, char *msg)
144   {
145     void *dead = (void*)Bdead;
146
147     if(b == dead)
148             panic("checkb b %s %lux", msg, b);
149     if(b->base == dead || b->lim == dead || b->next == dead
150       || b->rp == dead || b->wp == dead){
151             print("checkb: base 0x%8.8luX lim 0x%8.8luX next 0x%8.8luX\n",
152                     b->base, b->lim, b->next);
153             print("checkb: rp 0x%8.8luX wp 0x%8.8luX\n", b->rp, b->wp);
154             panic("checkb dead: %s\n", msg);
155     }
156
157     if(b->base > b->lim)
158             panic("checkb 0 %s %lux %lux", msg, b->base, b->lim);
159     if(b->rp < b->base)
160             panic("checkb 1 %s %lux %lux", msg, b->base, b->rp);
161     if(b->wp < b->base)
162             panic("checkb 2 %s %lux %lux", msg, b->base, b->wp);
163     if(b->rp > b->lim)
164             panic("checkb 3 %s %lux %lux", msg, b->rp, b->lim);
165     if(b->wp > b->lim)
166             panic("checkb 4 %s %lux %lux", msg, b->wp, b->lim);
167
168   }
169
170   void
171   iallocsummary(void)
172   {
173     print("ialloc %lud/%lud\n", ialloc.bytes, conf.ialloc);
174   }
```

```
#include  "u.h"
#include  "../port/lib.h"
#include  "mem.h"
#include  "dat.h"
#include  "fns.h"
#include  "../port/error.h"

typedef struct Crypt      Crypt;
struct Crypt
{
  Crypt           *next;
  Ticket          t;
  Authenticator   a;
  char            tbuf[TICKETLEN];        /* remote ticket */
};

typedef struct Session    Session;
struct Session
{
  Lock;
  Lock    send;
  Crypt   *cache;                 /* cache of tickets */
  char    cchal[CHALLEN];         /* client challenge */
  char    schal[CHALLEN];         /* server challenge */
  char    authid[NAMELEN];        /* server encryption uid */
  char    authdom[DOMLEN];        /* server encryption domain */
  ulong   cid;                    /* challenge id */
  int     valid;
};

struct
{
  Lock;
  Crypt           *free;
} cryptalloc;

char      eve[NAMELEN];
char      evekey[DESKEYLEN];
char      hostdomain[DOMLEN];

/*
 *  return true if current user is eve
 */
int
iseve(void)
{
  return strcmp(eve, up->user) == 0;
}

/*
 * crypt entries are allocated from a pool rather than allocated using malloc so
 * the memory can be protected from reading by devproc. The base and top of the
 * crypt arena is stored in palloc for devproc.
 */
Crypt*
newcrypt(void)
{
  Crypt *c;

  lock(&cryptalloc);
  if(cryptalloc.free) {
          c = cryptalloc.free;
          cryptalloc.free = c->next;
          unlock(&cryptalloc);
          memset(c, 0, sizeof(Crypt));
          return c;
  }

  cryptalloc.free = xalloc(sizeof(Crypt)*conf.nproc);
  if(cryptalloc.free == 0)
          panic("newcrypt");

  for(c = cryptalloc.free; c < cryptalloc.free+conf.nproc-1; c++)
          c->next = c+1;

  palloc.cmembase = (ulong)cryptalloc.free;
  palloc.cmemtop = palloc.cmembase+(sizeof(Crypt)*conf.nproc);
  unlock(&cryptalloc);
  return newcrypt();
}

void
freecrypt(Crypt *c)
{
  lock(&cryptalloc);
  c->next = cryptalloc.free;
  cryptalloc.free = c;
  unlock(&cryptalloc);
}

/*
 *  return the info received in the session message on this channel.
 *  if no session message has been exchanged, do it.
 */
long
sysfsession(ulong *arg)
{
  int i, n;
  Chan *c;
  Crypt *cp;
  Session *s;
  Ticketreq tr;
  Fcall f;
  char buf[MAXMSG];

  validaddr(arg[1], TICKREQLEN, 1);
  c = fdtochan(arg[0], OWRITE, 0, 1);
  if(waserror()){
          cclose(c);
          nexterror();
  }

  /* add a session structure to the channel if it has none */
  lock(c);
  s = c->session;
  if(s == 0){
          s = malloc(sizeof(Session));
          if(s == 0){
                  unlock(c);
                  error(Enomem);
          }
          c->session = s;
  }
  unlock(c);

  /* back off if someone else is doing an fsession */
  while(!canlock(&s->send))
          sched();

  if(s->valid == 0 && (c->flag & CMSG) == 0){

          /*
           *  Exchange a session message with the server.
           *  If an error occurs reading or writing,
           *  assume this is a mount of a mount and turn off
           *  authentication.
```

```
137             */
138            if(!waserror()){
139                    for(i = 0; i < CHALLEN; i++)
140                            s->cchal[i] = nrand(256);
141                    f.tag = NOTAG;
142                    f.type = Tsession;
143                    memmove(f.chal, s->cchal, CHALLEN);
144                    n = convS2M(&f, buf);
145                    if(devtab[c->type]->write(c, buf, n, 0) != n)
146                            error(Emountrpc);
147                    n = devtab[c->type]->read(c, buf, sizeof buf, 0);
148                    if(n == 2 && buf[0] == 'O' && buf[1] == 'K')
149                            n = devtab[c->type]->read(c, buf, sizeof buf, 0);
150                    poperror();
151                    if(convM2S(buf, &f, n) == 0){
152                            unlock(&s->send);
153                            error(Emountrpc);
154                    }
155                    switch(f.type){
156                    case Rsession:
157                            memmove(s->schal, f.chal, CHALLEN);
158                            memmove(s->authid, f.authid, NAMELEN);
159                            memmove(s->authdom, f.authdom, DOMLEN);
160                            break;
161                    case Rerror:
162                            unlock(&s->send);
163                            error(f.ename);
164                    default:
165                            unlock(&s->send);
166                            error(Emountrpc);
167                    }
168            }
169            s->valid = 1;
170    }
171    unlock(&s->send);
172
173    /*
174     *  If server requires no ticket, or user is "none", or a ticket
175     *  is already cached, zero the request type
176     */
177    tr.type = AuthTreq;
178    if(strcmp(up->user, "none") == 0 || s->authid[0] == 0)
179            tr.type = 0;
180    else{
181            lock(s);
182            for(cp = s->cache; cp; cp = cp->next)
183                    if(strcmp(cp->t.cuid, up->user) == 0){
184                            tr.type = 0;
185                            break;
186                    }
187            unlock(s);
188    }
189
190    /*  create ticket request */
191    memmove(tr.chal, s->schal, CHALLEN);
192    memmove(tr.authid, s->authid, NAMELEN);
193    memmove(tr.authdom, s->authdom, DOMLEN);
194    memmove(tr.uid, up->user, NAMELEN);
195    memmove(tr.hostid, eve, NAMELEN);
196    convTR2M(&tr, (char*)arg[1]);
197
198    cclose(c);
199    poperror();
200    return 0;
201  }
202
203  /*
204   *  attach tickets to a session
205   */
206  long
207  sysfauth(ulong *arg)
208  {
209    Chan *c;
210    char *ta;
211    Session *s;
212    Crypt *cp, *ncp, **l;
213    char tbuf[2*TICKETLEN];
214
215    validaddr(arg[1], 2*TICKETLEN, 0);
216    c = fdtochan(arg[0], OWRITE, 0, 1);
217    s = c->session;
218    if(s == 0){
219            cclose(c);
220            error("fauth must follow fsession");
221    }
222    cp = newcrypt();
223    if(waserror()){
224            cclose(c);
225            freecrypt(cp);
226            nexterror();
227    }
228
229    /*  ticket supplied, use it */
230    ta = (char*)arg[1];
231    memmove(tbuf, ta, 2*TICKETLEN);
232    convM2T(tbuf, &cp->t, evekey);
233    if(cp->t.num != AuthTc)
234            error("bad AuthTc in ticket");
235    if(strncmp(up->user, cp->t.cuid, NAMELEN) != 0)
236            error("bad uid in ticket");
237    if(memcmp(cp->t.chal, s->schal, CHALLEN) != 0)
238            error("bad chal in ticket");
239    memmove(cp->tbuf, tbuf+TICKETLEN, TICKETLEN);
240
241    /* string onto local list, replace old version */
242    lock(s);
243    l = &s->cache;
244    for(ncp = s->cache; ncp; ncp = *l){
245            if(strcmp(ncp->t.cuid, up->user) == 0){
246                    *l = ncp->next;
247                    freecrypt(ncp);
248                    break;
249            }
250            l = &ncp->next;
251    }
252    cp->next = s->cache;
253    s->cache = cp;
254    unlock(s);
255    cclose(c);
256    poperror();
257    return 0;
258  }
259
260  /*
261   *  free a session created by fsession
262   */
263  void
264  freesession(Session *s)
265  {
266    Crypt *cp, *next;
267
268    for(cp = s->cache; cp; cp = next) {
269            next = cp->next;
270            freecrypt(cp);
271    }
272    free(s);
```

```
}

/*
 *  called by mattach() to fill in the Tattach message
 */
ulong
authrequest(Session *s, Fcall *f)
{
  Crypt *cp;
  ulong id, dofree;

  /* no authentication if user is "none" or if no ticket required by remote */
  if(s == 0 || s->authid[0] == 0 || strcmp(up->user, "none") == 0){
          memset(f->ticket, 0, TICKETLEN);
          memset(f->auth, 0, AUTHENTLEN);
          return 0;
  }

  /* look for ticket in cache */
  dofree = 0;
  lock(s);
  for(cp = s->cache; cp; cp = cp->next)
          if(strcmp(cp->t.cuid, up->user) == 0)
                  break;

  id = s->cid++;
  unlock(s);

  if(cp == 0){
          /*
           *  create a ticket using hostkey, this solves the
           *  chicken and egg problem
           */
          cp = newcrypt();
          cp->t.num = AuthTs;
          memmove(cp->t.chal, s->schal, CHALLEN);
          memmove(cp->t.cuid, up->user, NAMELEN);
          memmove(cp->t.suid, up->user, NAMELEN);
          memmove(cp->t.key, evekey, DESKEYLEN);
          convT2M(&cp->t, f->ticket, evekey);
          dofree = 1;
  } else
          memmove(f->ticket, cp->tbuf, TICKETLEN);

  /* create an authenticator */
  memmove(cp->a.chal, s->schal, CHALLEN);
  cp->a.num = AuthAc;
  cp->a.id = id;
  convA2M(&cp->a, f->auth, cp->t.key);
  if(dofree)
          freecrypt(cp);
  return id;
}

/*
 *  called by mattach() to check the Rattach message
 */
void
authreply(Session *s, ulong id, Fcall *f)
{
  Crypt *cp;

  if(s == 0)
          return;

  lock(s);
  for(cp = s->cache; cp; cp = cp->next)
          if(strcmp(cp->t.cuid, up->user) == 0)
                  break;
  unlock(s);

  /* we're getting around authentication */
  if(s == 0 || cp == 0 || s->authid[0] == 0 || strcmp(up->user, "none") == 0)
          return;

  convM2A(f->rauth, &cp->a, cp->t.key);
  if(cp->a.num != AuthAs){
          print("bad encryption type\n");
          error("server lies");
  }
  if(memcmp(cp->a.chal, s->cchal, sizeof(cp->a.chal))){
          print("bad returned challenge\n");
          error("server lies");
  }
  if(cp->a.id != id){
          print("bad returned id\n");
          error("server lies");
  }
}

/*
 *  called by devcons() for #c/authenticate
 *
 *  The protocol is
 *        1) read ticket request from #c/authenticate
 *        2) write ticket+authenticator to #c/authenticate. if it matches
 *          the challenge the user is changed to the suid field of the ticket
 *        3) read authenticator (to confirm this is the server advertised)
 */
long
authread(Chan *c, char *a, int n)
{
  Crypt *cp;
  int i;
  Ticketreq tr;

  if(c->aux == 0){
          /*
           *  first read returns a ticket request
           */
          if(n != TICKREQLEN)
                  error(Ebadarg);
          c->aux = newcrypt();
          cp = c->aux;

          memset(&tr, 0, sizeof(tr));
          tr.type = AuthTreq;
          strcpy(tr.hostid, eve);
          strcpy(tr.authid, eve);
          strcpy(tr.authdom, hostdomain);
          strcpy(tr.uid, up->user);
          for(i = 0; i < CHALLEN; i++)
                  tr.chal[i] = nrand(256);
          memmove(cp->a.chal, tr.chal, CHALLEN);
          convTR2M(&tr, a);
  } else {
          /*
           *  subsequent read returns an authenticator
           */
          if(n < AUTHENTLEN)
                  error(Ebadarg);
          cp = c->aux;

          cp->a.num = AuthAs;
          memmove(cp->a.chal, cp->t.chal, CHALLEN);
          cp->a.id = 0;
```

```
409             convA2M(&cp->a, cp->tbuf, cp->t.key);
410             memmove(a, cp->tbuf, AUTHENTLEN);
411
412             if(n >= AUTHENTLEN + TICKETLEN)
413                     convT2M(&cp->t, a+AUTHENTLEN, nil);
414
415             freecrypt(cp);
416             c->aux = 0;
417     }
418     return n;
419   }
420
421   long
422   authwrite(Chan *c, char *a, int n)
423   {
424     Crypt *cp;
425
426     if(n != TICKETLEN+AUTHENTLEN)
427             error(Ebadarg);
428     if(c->aux == 0)
429             error(Ebadarg);
430     cp = c->aux;
431
432     memmove(cp->tbuf, a, TICKETLEN);
433     convM2T(cp->tbuf, &cp->t, evekey);
434     if(cp->t.num != AuthTs || memcmp(cp->a.chal, cp->t.chal, CHALLEN))
435             error(Eperm);
436
437     memmove(cp->tbuf, a+TICKETLEN, AUTHENTLEN);
438     convM2A(cp->tbuf, &cp->a, cp->t.key);
439     if(cp->a.num != AuthAc || memcmp(cp->a.chal, cp->t.chal, CHALLEN))
440             error(Eperm);
441
442     memmove(up->user, cp->t.suid, NAMELEN);
443     return n;
444   }
445
446   /*
447    *  called by devcons() for #c/authcheck
448    *
449    *  a write of a ticket+authenticator [+challenge+id] succeeds if they match
450    */
451   long
452   authcheck(Chan *c, char *a, int n)
453   {
454     Crypt *cp;
455     char *chal;
456     ulong id;
457
458     if(n != TICKETLEN+AUTHENTLEN && n != TICKETLEN+AUTHENTLEN+CHALLEN+4)
459             error(Ebadarg);
460     if(c->aux == 0)
461             c->aux = newcrypt();
462     cp = c->aux;
463
464     memmove(cp->tbuf, a, TICKETLEN);
465     convM2T(cp->tbuf, &cp->t, evekey);
466     if(cp->t.num != AuthTc)
467             error(Ebadarg);
468     if(strcmp(up->user, cp->t.cuid))
469             error(cp->t.cuid);
470
471     memmove(cp->tbuf, a+TICKETLEN, AUTHENTLEN);
472     convM2A(cp->tbuf, &cp->a, cp->t.key);
473     if(n == TICKETLEN+AUTHENTLEN+CHALLEN+4){
474             uchar *p = (uchar *)&a[TICKETLEN+AUTHENTLEN+CHALLEN];
475             id = p[0] | (p[1]<<8) | (p[2]<<16) | (p[3]<<24);
476             chal = &a[TICKETLEN+AUTHENTLEN];
477     }else{
478             id = 0;
479             chal = cp->t.chal;
480     }
481     if(cp->a.num != AuthAs || memcmp(chal, cp->a.chal, CHALLEN) || cp->a.id != id)
482             error(Eperm);
483
484     return n;
485   }
486
487   /*
488    *  reading authcheck after writing into it yields the
489    *  unencrypted ticket
490    */
491   long
492   authcheckread(Chan *c, char *a, int n)
493   {
494     Crypt *cp;
495
496     cp = c->aux;
497     if(cp == nil)
498             error(Ebadarg);
499     if(n < TICKETLEN)
500             error(Ebadarg);
501     convT2M(&cp->t, a, nil);
502     return sizeof(cp->t);
503   }
504
505   /*
506    *  called by devcons() for #c/authenticator
507    *
508    *  a read after a write of a ticket (or ticket+id) returns an authenticator
509    *  for that ticket.
510    */
511   long
512   authentwrite(Chan *c, char *a, int n)
513   {
514     Crypt *cp;
515
516     if(n != TICKETLEN && n != TICKETLEN+4)
517             error(Ebadarg);
518     if(c->aux == 0)
519             c->aux = newcrypt();
520     cp = c->aux;
521
522     memmove(cp->tbuf, a, TICKETLEN);
523     convM2T(cp->tbuf, &cp->t, evekey);
524     if(cp->t.num != AuthTc || strcmp(cp->t.cuid, up->user)){
525             freecrypt(cp);
526             c->aux = 0;
527             error(Ebadarg);
528     }
529     if(n == TICKETLEN+4){
530             uchar *p = (uchar *)&a[TICKETLEN];
531             cp->a.id = p[0] | (p[1]<<8) | (p[2]<<16) | (p[3]<<24);
532     }else
533             cp->a.id = 0;
534
535     return n;
536   }
537
538   /*
539    *  create an authenticator and return it and optionally the
540    *  unencrypted ticket
541    */
542   long
543   authentread(Chan *c, char *a, int n)
544   {
```

```
545     Crypt *cp;
546
547     cp = c->aux;
548     if(cp == 0)
549             error("authenticator read must follow a write");
550
551     cp->a.num = AuthAc;
552     memmove(cp->a.chal, cp->t.chal, CHALLEN);
553     convA2M(&cp->a, cp->tbuf, cp->t.key);
554
555     if(n >= AUTHENTLEN)
556             memmove(a, cp->tbuf, AUTHENTLEN);
557     return n;
558   }
559
560   void
561   authclose(Chan *c)
562   {
563     if(c->aux)
564             freecrypt(c->aux);
565     c->aux = 0;
566   }
567
568   /*
569    *  called by devcons() for key device
570    */
571   long
572   keyread(char *a, int n, long offset)
573   {
574     if(n<DESKEYLEN || offset != 0)
575             error(Ebadarg);
576     if(!cpuserver || !iseve())
577             error(Eperm);
578     memmove(a, evekey, DESKEYLEN);
579     return DESKEYLEN;
580   }
581
582   long
583   keywrite(char *a, int n)
584   {
585     if(n != DESKEYLEN)
586             error(Ebadarg);
587     if(!iseve())
588             error(Eperm);
589     memmove(evekey, a, DESKEYLEN);
590     return DESKEYLEN;
591   }
592
593   /*
594    *  called by devcons() for user device
595    *
596    *  anyone can become none
597    */
598   long
599   userwrite(char *a, int n)
600   {
601     if(n >= NAMELEN)
602             error(Ebadarg);
603     if(strcmp(a, "none") != 0)
604             error(Eperm);
605     memset(up->user, 0, NAMELEN);
606     strcpy(up->user, "none");
607     up->basepri = PriNormal;
608     return n;
609   }
610
611   /*
612    *  called by devcons() for host owner/domain
613    *
614    *  writing hostowner also sets user
615    */
616   long
617   hostownerwrite(char *a, int n)
618   {
619     char buf[NAMELEN];
620
621     if(!iseve())
622             error(Eperm);
623     if(n >= NAMELEN)
624             error(Ebadarg);
625     memset(buf, 0, NAMELEN);
626     strncpy(buf, a, n);
627     if(buf[0] == 0)
628             error(Ebadarg);
629     renameuser(eve, buf);
630     memmove(eve, buf, NAMELEN);
631     memmove(up->user, buf, NAMELEN);
632     up->basepri = PriNormal;
633     return n;
634   }
635
636   long
637   hostdomainwrite(char *a, int n)
638   {
639     char buf[DOMLEN];
640
641     if(!iseve())
642             error(Eperm);
643     if(n >= DOMLEN)
644             error(Ebadarg);
645     memset(buf, 0, DOMLEN);
646     strncpy(buf, a, n);
647     if(buf[0] == 0)
648             error(Ebadarg);
649     memmove(hostdomain, buf, DOMLEN);
650     return n;
651   }
```

```
#include  "u.h"
#include  "../port/lib.h"
#include  "mem.h"
#include  "dat.h"
#include  "fns.h"
#include  "../port/error.h"

enum
{
  NHASH           = 128,
  MAXCACHE        = 1024*1024,
  NFILE           = 4096,
  NEXTENT         = 200,          /* extent allocation size */
};

typedef struct Extent Extent;
struct Extent
{
  int     bid;
  ulong   start;
  int     len;
  Page    *cache;
  Extent  *next;
};

typedef struct Mntcache Mntcache;
struct Mntcache
{
  Qid;
  int     dev;
  int     type;
  QLock;
  Extent   *list;
  Mntcache *hash;
  Mntcache *prev;
  Mntcache *next;
};

typedef struct Cache Cache;
struct Cache
{
  Lock;
  int             pgno;
  Mntcache        *head;
  Mntcache        *tail;
  Mntcache        *hash[NHASH];
};

typedef struct Ecache Ecache;
struct Ecache
{
  Lock;
  int     total;
  int     free;
  Extent* head;
};

static Image fscache;
static Cache cache;
static Ecache ecache;
static int maxcache = MAXCACHE;

static void
extentfree(Extent* e)
{
  lock(&ecache);
  e->next = ecache.head;
  ecache.head = e;
  ecache.free++;
  unlock(&ecache);
}

static Extent*
extentalloc(void)
{
  Extent *e;
  int i;

  lock(&ecache);
  if(ecache.head == nil){
          e = xalloc(NEXTENT*sizeof(Extent));
          if(e == nil){
                  unlock(&ecache);
                  return nil;
          }
          for(i = 0; i < NEXTENT; i++){
                  e->next = ecache.head;
                  ecache.head = e;
                  e++;
          }
          ecache.free += NEXTENT;
          ecache.total += NEXTENT;
  }

  e = ecache.head;
  ecache.head = e->next;
  memset(e, 0, sizeof(Extent));
  ecache.free--;
  unlock(&ecache);

  return e;
}

void
cinit(void)
{
  int i;
  Mntcache *m;

  cache.head = xalloc(sizeof(Mntcache)*NFILE);
  m = cache.head;

  if(conf.npage*BY2PG > 200*MB)
          maxcache = 10*MAXCACHE;

  for(i = 0; i < NFILE-1; i++) {
          m->next = m+1;
          m->prev = m-1;
          m++;
  }

  cache.tail = m;
  cache.tail->next = 0;
  cache.head->prev = 0;

  fscache.notext = 1;
}

void
cprint(Chan *c, Mntcache *m, char *s)
{
  ulong o;
  int nb, ct;
  Extent *e;

  nb = 0;
```

```
137     ct = 1;
138     o = 0;
139     if(m->list)
140             o = m->list->start;
141     for(e = m->list; e; e = e->next) {
142             nb += e->len;
143             if(o != e->start)
144                     ct = 0;
145             o = e->start+e->len;
146     }
147     pprint("%s: 0x%lux.0x%lux %d %d %s (%d %c)\n",
148     s, m->path, m->vers, m->type, m->dev, c->name, nb, ct ? 'C' : 'N');
149
150     for(e = m->list; e; e = e->next) {
151             pprint("\t%4d %5d %4d %lux\n",
152                     e->bid, e->start, e->len, e->cache);
153     }
154   }
155
156   Page*
157   cpage(Extent *e)
158   {
159     /* Easy consistency check */
160     if(e->cache->daddr != e->bid)
161             return 0;
162
163     return lookpage(&fscache, e->bid);
164   }
165
166   void
167   cnodata(Mntcache *m)
168   {
169     Extent *e, *n;
170
171     /*
172      * Invalidate all extent data
173      * Image lru will waste the pages
174      */
175     for(e = m->list; e; e = n) {
176             n = e->next;
177             extentfree(e);
178     }
179     m->list = 0;
180   }
181
182   void
183   ctail(Mntcache *m)
184   {
185     /* Unlink and send to the tail */
186     if(m->prev)
187             m->prev->next = m->next;
188     else
189             cache.head = m->next;
190     if(m->next)
191             m->next->prev = m->prev;
192     else
193             cache.tail = m->prev;
194
195     if(cache.tail) {
196             m->prev = cache.tail;
197             cache.tail->next = m;
198             m->next = 0;
199             cache.tail = m;
200     }
201     else {
202             cache.head = m;
203             cache.tail = m;
204             m->prev = 0;
205             m->next = 0;
206     }
207   }
208
209   void
210   copen(Chan *c)
211   {
212     int h;
213     Extent *e, *next;
214     Mntcache *m, *f, **l;
215
216     if(c->qid.path&CHDIR)
217             return;
218
219     h = c->qid.path%NHASH;
220     lock(&cache);
221     for(m = cache.hash[h]; m; m = m->hash) {
222             if(m->path == c->qid.path)
223             if(m->dev == c->dev && m->type == c->type) {
224                     c->mcp = m;
225                     ctail(m);
226                     unlock(&cache);
227
228                     /* File was updated, invalidate cache */
229                     if(m->vers != c->qid.vers) {
230                             m->vers = c->qid.vers;
231                             qlock(m);
232                             cnodata(m);
233                             qunlock(m);
234                     }
235                     return;
236             }
237     }
238
239     /* LRU the cache headers */
240     m = cache.head;
241     l = &cache.hash[m->path%NHASH];
242     for(f = *l; f; f = f->hash) {
243             if(f == m) {
244                     *l = m->hash;
245                     break;
246             }
247             l = &f->hash;
248     }
249
250     m->Qid = c->qid;
251     m->dev = c->dev;
252     m->type = c->type;
253
254     l = &cache.hash[h];
255     m->hash = *l;
256     *l = m;
257     ctail(m);
258
259     qlock(m);
260     c->mcp = m;
261     e = m->list;
262     m->list = 0;
263     unlock(&cache);
264
265     while(e) {
266             next = e->next;
267             extentfree(e);
268             e = next;
269     }
270     qunlock(m);
271   }
272
```

```
static int
cdev(Mntcache *m, Chan *c)
{
  if(m->path != c->qid.path)
          return 0;
  if(m->dev != c->dev)
          return 0;
  if(m->type != c->type)
          return 0;
  if(m->vers != c->qid.vers)
          return 0;
  return 1;
}

int
cread(Chan *c, uchar *buf, int len, vlong off)
{
  KMap *k;
  Page *p;
  Mntcache *m;
  Extent *e, **t;
  int o, l, total;
  ulong offset;

  if(off+len > maxcache)
          return 0;

  m = c->mcp;
  if(m == 0)
          return 0;

  qlock(m);
  if(cdev(m, c) == 0) {
          qunlock(m);
          return 0;
  }

  offset = off;
  t = &m->list;
  for(e = *t; e; e = e->next) {
          if(offset >= e->start && offset < e->start+e->len)
                  break;
          t = &e->next;
  }

  if(e == 0) {
          qunlock(m);
          return 0;
  }

  total = 0;
  while(len) {
          p = cpage(e);
          if(p == 0) {
                  *t = e->next;
                  extentfree(e);
                  qunlock(m);
                  return total;
          }

          o = offset - e->start;
          l = len;
          if(l > e->len-o)
                  l = e->len-o;

          k = kmap(p);
          if(waserror()) {
                  kunmap(k);
                  putpage(p);
                  qunlock(m);
                  nexterror();
          }

          memmove(buf, (uchar*)VA(k) + o, l);

          poperror();
          kunmap(k);

          putpage(p);

          buf += l;
          len -= l;
          offset += l;
          total += l;
          t = &e->next;
          e = e->next;
          if(e == 0 || e->start != offset)
                  break;
  }

  qunlock(m);
  return total;
}

Extent*
cchain(uchar *buf, ulong offset, int len, Extent **tail)
{
  int l;
  Page *p;
  KMap *k;
  Extent *e, *start, **t;

  start = 0;
  *tail = 0;
  t = &start;
  while(len) {
          e = extentalloc();
          if(e == 0)
                  break;

          p = auxpage();
          if(p == 0) {
                  extentfree(e);
                  break;
          }
          l = len;
          if(l > BY2PG)
                  l = BY2PG;

          e->cache = p;
          e->start = offset;
          e->len = l;

          lock(&cache);
          e->bid = cache.pgno;
          cache.pgno += BY2PG;
          /* wrap the counter; low bits are unused by pghash but checked by lookpage */
          if((cache.pgno & ~(BY2PG-1)) == 0){
                  if(cache.pgno == BY2PG-1){
                          print("cache wrapped\n");
                          cache.pgno = 0;
                  }else
                          cache.pgno++;
          }
          unlock(&cache);

```

```
409             p->daddr = e->bid;
410             k = kmap(p);
411             if(waserror()) {                /* buf may be virtual */
412                     kunmap(k);
413                     nexterror();
414             }
415             memmove((void*)VA(k), buf, l);
416             poperror();
417             kunmap(k);
418
419             cachepage(p, &fscache);
420             putpage(p);
421
422             buf += l;
423             offset += l;
424             len -= l;
425
426             *t = e;
427             *tail = e;
428             t = &e->next;
429     }
430
431     return start;
432   }
433
434   int
435   cpgmove(Extent *e, uchar *buf, int boff, int len)
436   {
437     Page *p;
438     KMap *k;
439
440     p = cpage(e);
441     if(p == 0)
442             return 0;
443
444     k = kmap(p);
445     if(waserror()) {                /* Since buf may be virtual */
446             kunmap(k);
447             nexterror();
448     }
449
450     memmove((uchar*)VA(k)+boff, buf, len);
451
452     poperror();
453     kunmap(k);
454     putpage(p);
455
456     return 1;
457   }
458
459   void
460   cupdate(Chan *c, uchar *buf, int len, vlong off)
461   {
462     Mntcache *m;
463     Extent *tail;
464     Extent *e, *f, *p;
465     int o, ee, eblock;
466     ulong offset;
467
468     if(off > maxcache || len == 0)
469             return;
470
471     m = c->mcp;
472     if(m == 0)
473             return;
474     qlock(m);
475     if(cdev(m, c) == 0) {
476             qunlock(m);
477             return;
478     }
479
480     /*
481      * Find the insertion point
482      */
483     offset = off;
484     p = 0;
485     for(f = m->list; f; f = f->next) {
486             if(f->start > offset)
487                     break;
488             p = f;
489     }
490
491     /* trim if there is a successor */
492     eblock = offset+len;
493     if(f != 0 && eblock > f->start) {
494             len -= (eblock - f->start);
495             if(len <= 0) {
496                     qunlock(m);
497                     return;
498             }
499     }
500
501     if(p == 0) {            /* at the head */
502             e = cchain(buf, offset, len, &tail);
503             if(e != 0) {
504                     m->list = e;
505                     tail->next = f;
506             }
507             qunlock(m);
508             return;
509     }
510
511     /* trim to the predecessor */
512     ee = p->start+p->len;
513     if(offset < ee) {
514             o = ee - offset;
515             len -= o;
516             if(len <= 0) {
517                     qunlock(m);
518                     return;
519             }
520             buf += o;
521             offset += o;
522     }
523
524     /* try and pack data into the predecessor */
525     if(offset == ee && p->len < BY2PG) {
526             o = len;
527             if(o > BY2PG - p->len)
528                     o = BY2PG - p->len;
529             if(cpgmove(p, buf, p->len, o)) {
530                     p->len += o;
531                     buf += o;
532                     len -= o;
533                     offset += o;
534                     if(len <= 0) {
535   if (f && p->start + p->len > f->start) print("CACHE: p->start=%uld p->len=%d f->start=%uld\n"
536   art, p->len, f->start);
537                             qunlock(m);
538                             return;
539                     }
540             }
541     }
542
543     e = cchain(buf, offset, len, &tail);
544     if(e != 0) {
```

```
545             p->next = e;
546             tail->next = f;
547     }
548     qunlock(m);
549   }
550
551   void
552   cwrite(Chan* c, uchar *buf, int len, vlong off)
553   {
554     int o, eo;
555     Mntcache *m;
556     ulong eblock, ee;
557     Extent *p, *f, *e, *tail;
558     ulong offset;
559
560     if(off > maxcache || len == 0)
561             return;
562
563     m = c->mcp;
564     if(m == 0)
565             return;
566
567     qlock(m);
568     if(cdev(m, c) == 0) {
569             qunlock(m);
570             return;
571     }
572
573     offset = off;
574     m->vers++;
575     c->qid.vers++;
576
577     p = 0;
578     for(f = m->list; f; f = f->next) {
579             if(f->start >= offset)
580                     break;
581             p = f;
582     }
583
584     if(p != 0) {
585             ee = p->start+p->len;
586             eo = offset - p->start;
587             /* pack in predecessor if there is space */
588             if(offset <= ee && eo < BY2PG) {
589                     o = len;
590                     if(o > BY2PG - eo)
591                             o = BY2PG - eo;
592                     if(cpgmove(p, buf, eo, o)) {
593                             if(eo+o > p->len)
594                                     p->len = eo+o;
595                             buf += o;
596                             len -= o;
597                             offset += o;
598                     }
599             }
600     }
601
602     /* free the overlap -- it's a rare case */
603     eblock = offset+len;
604     while(f && f->start < eblock) {
605             e = f->next;
606             extentfree(f);
607             f = e;
608     }
609
610     /* link the block (if any) into the middle */
611     e = cchain(buf, offset, len, &tail);
612     if(e != 0) {
613             tail->next = f;
614             f = e;
615     }
616
617     if(p == 0)
618             m->list = f;
619     else
620             p->next = f;
621     qunlock(m);
622   }
```

```
#include  "u.h"
#include  "../port/lib.h"
#include  "mem.h"
#include  "dat.h"
#include  "fns.h"
#include  "../port/error.h"

enum
{
  CNAMESLOP       = 20
};

struct
{
  Lock;
  int     fid;
  Chan    *free;
  Chan    *list;
}chanalloc;

#define SEP(c) ((c) == 0 || (c) == '/')
void cleancname(Cname*);

int
incref(Ref *r)
{
  int x;

  lock(r);
  x = ++r->ref;
  unlock(r);
  return x;
}

int
decref(Ref *r)
{
  int x;

  lock(r);
  x = --r->ref;
  unlock(r);
  if(x < 0)
          panic("decref");

  return x;
}

void
chandevreset(void)
{
  int i;

  for(i=0; devtab[i] != nil; i++)
          devtab[i]->reset();
}

void
chandevinit(void)
{
  int i;

  for(i=0; devtab[i] != nil; i++)
          devtab[i]->init();
}

Chan*
newchan(void)
{
  Chan *c;

  lock(&chanalloc);
  c = chanalloc.free;
  if(c != 0)
          chanalloc.free = c->next;
  unlock(&chanalloc);

  if(c == 0) {
          c = smalloc(sizeof(Chan));
          lock(&chanalloc);
          c->fid = ++chanalloc.fid;
          c->link = chanalloc.list;
          chanalloc.list = c;
          unlock(&chanalloc);
  }

  /* if you get an error before associating with a dev,
     close calls rootclose, a nop */
  c->type = 0;
  c->flag = 0;
  c->ref = 1;
  c->dev = 0;
  c->offset = 0;
  c->mh = 0;
  c->xmh = 0;
  c->uri = 0;
  c->aux = 0;
  c->mchan = 0;
  c->mcp = 0;
  c->mqid = (Qid){0, 0};
  c->name = 0;
  return c;
}

static Ref ncname;

Cname*
newcname(char *s)
{
  Cname *n;
  int i;

  n = smalloc(sizeof(Cname));
  i = strlen(s);
  n->len = i;
  n->alen = i+CNAMESLOP;
  n->s = smalloc(n->alen);
  memmove(n->s, s, i+1);
  n->ref = 1;
  incref(&ncname);
  return n;
}

void
cnameclose(Cname *n)
{
  if(n == 0)
          return;
  if(decref(n))
          return;
  decref(&ncname);
  free(n->s);
  free(n);
}

Cname*
```

```
addelem(Cname *n, char *s)
{
  int i, a;
  char *t;
  Cname *new;

  if(n->ref > 1){
          /* copy on write */
          new = newcname(n->s);
          cnameclose(n);
          n = new;
  }

  i = strlen(s);
  if(n->len+1+i+1 > n->alen){
          a = n->len+1+i+1 + CNAMESLOP;
          t = smalloc(a);
          memmove(t, n->s, n->len+1);
          free(n->s);
          n->s = t;
          n->alen = a;
  }
  if(n->len>0 && n->s[n->len-1]!='/' && s[0]!='/')        /* don't insert extra slash if one
          n->s[n->len++] = '/';
  memmove(n->s+n->len, s, i+1);
  n->len += i;
  return n;
}

void
chanfree(Chan *c)
{
  c->flag = CFREE;

  if(c->session){
          freesession(c->session);
          c->session = 0;
  }

  if(c->mh != nil){
          putmhead(c->mh);
          c->mh = nil;
  }

  cnameclose(c->name);

  lock(&chanalloc);
  c->next = chanalloc.free;
  chanalloc.free = c;
  unlock(&chanalloc);
}

void
cclose(Chan *c)
{
  if(c->flag&CFREE)
          panic("cclose %lux", getcallerpc(&c));
  if(decref(c))
          return;

  if(!waserror()) {
          devtab[c->type]->close(c);
          poperror();
  }

  chanfree(c);
}

int
eqqid(Qid a, Qid b)
{
  return a.path==b.path && a.vers==b.vers;
}

int
eqchan(Chan *a, Chan *b, int pathonly)
{
  if(a->qid.path != b->qid.path)
          return 0;
  if(!pathonly && a->qid.vers!=b->qid.vers)
          return 0;
  if(a->type != b->type)
          return 0;
  if(a->dev != b->dev)
          return 0;
  return 1;
}

int
cmount(Chan *new, Chan *old, int flag, char *spec)
{
  Pgrp *pg;
  int order, flg;
  Mhead *m, **l;
  Mount *nm, *f, *um, **h;

  if(CHDIR & (old->qid.path^new->qid.path))
          error(Emount);

  order = flag&MORDER;

  if((old->qid.path&CHDIR)==0 && order != MREPL)
          error(Emount);

  pg = up->pgrp;
  wlock(&pg->ns);

  l = &MOUNTH(pg, old);
  for(m = *l; m; m = m->hash) {
          if(eqchan(m->from, old, 1))
                  break;
          l = &m->hash;
  }

  if(m == 0) {
          /*
           *  nothing mounted here yet.  create a mount
           *  head and add to the hash table.
           */
          m = smalloc(sizeof(Mhead));
          m->ref = 1;
          m->from = old;
          incref(old);
          m->hash = *l;
          *l = m;

          /*
           *  if this is a union mount, add the old
           *  node to the mount chain.
           */
          if(order != MREPL)
                  m->mount = newmount(m, old, 0, 0);
  }
  wlock(&m->lock);
  if(waserror()){
          wunlock(&m->lock);
```

```
273             nexterror();
274     }
275     wunlock(&pg->ns);
276
277     nm = newmount(m, new, flag, spec);
278     if(new->mh != nil && new->mh->mount != nil) {
279             /*
280              *  copy a union when binding it onto a directory
281              */
282             flg = order;
283             if(order == MREPL)
284                     flg = MAFTER;
285             h = &nm->next;
286             um = new->mh->mount;
287             for(um = um->next; um; um = um->next) {
288                     f = newmount(m, um->to, flg, um->spec);
289                     *h = f;
290                     h = &f->next;
291             }
292     }
293
294     if(m->mount && order == MREPL) {
295             mountfree(m->mount);
296             m->mount = 0;
297     }
298
299     if(flag & MCREATE)
300             new->flag |= CCREATE;
301
302     if(m->mount && order == MAFTER) {
303             for(f = m->mount; f->next; f = f->next)
304                     ;
305             f->next = nm;
306     }
307     else {
308             for(f = nm; f->next; f = f->next)
309                     ;
310             f->next = m->mount;
311             m->mount = nm;
312     }
313
314     wunlock(&m->lock);
315     poperror();
316     return nm->mountid;
317   }
318
319   void
320   cunmount(Chan *mnt, Chan *mounted)
321   {
322     Pgrp *pg;
323     Mhead *m, **l;
324     Mount *f, **p;
325
326     pg = up->pgrp;
327     wlock(&pg->ns);
328
329     l = &MOUNTH(pg, mnt);
330     for(m = *l; m; m = m->hash) {
331             if(eqchan(m->from, mnt, 1))
332                     break;
333             l = &m->hash;
334     }
335
336     if(m == 0) {
337             wunlock(&pg->ns);
338             error(Eunmount);
339     }
340
341     wlock(&m->lock);
342     if(mounted == 0) {
343             *l = m->hash;
344             wunlock(&pg->ns);
345             mountfree(m->mount);
346             m->mount = nil;
347             cclose(m->from);
348             wunlock(&m->lock);
349             putmhead(m);
350             return;
351     }
352     wunlock(&pg->ns);
353
354     p = &m->mount;
355     for(f = *p; f; f = f->next) {
356             /* BUG: Needs to be 2 pass */
357             if(eqchan(f->to, mounted, 1) ||
358               (f->to->mchan && eqchan(f->to->mchan, mounted, 1))) {
359                     *p = f->next;
360                     f->next = 0;
361                     mountfree(f);
362                     if(m->mount == nil) {
363                             *l = m->hash;
364                             wunlock(&pg->ns);
365                             cclose(m->from);
366                             wunlock(&m->lock);
367                             putmhead(m);
368                             return;
369                     }
370                     wunlock(&m->lock);
371                     return;
372             }
373             p = &f->next;
374     }
375     wunlock(&m->lock);
376     error(Eunion);
377   }
378
379   Chan*
380   cclone(Chan *c, Chan *nc)
381   {
382     nc = devtab[c->type]->clone(c, nc);
383     if(nc != nil){
384             nc->name = c->name;
385             if(c->name)
386                     incref(c->name);
387     }
388     return nc;
389   }
390
391   Chan*
392   domount(Chan *c)
393   {
394     Pgrp *pg;
395     Chan *nc;
396     Mhead *m;
397
398     pg = up->pgrp;
399     rlock(&pg->ns);
400     if(c->mh){
401             putmhead(c->mh);
402             c->mh = 0;
403     }
404
405     for(m = MOUNTH(pg, c); m; m = m->hash){
406             rlock(&m->lock);
407             if(eqchan(m->from, c, 1)) {
408                     if(waserror()) {
```

```
409                             runlock(&m->lock);
410                             nexterror();
411                     }
412                     runlock(&pg->ns);
413                     nc = cclone(m->mount->to, 0);
414                     if(nc->mh != nil)
415                             putmhead(nc->mh);
416                     nc->mh = m;
417                     nc->xmh = m;
418                     incref(m);
419                     cnameclose(nc->name);
420                     nc->name = c->name;
421                     incref(c->name);
422                     cclose(c);
423                     c = nc;
424                     poperror();
425                     runlock(&m->lock);
426                     return c;
427             }
428             runlock(&m->lock);
429     }
430
431     runlock(&pg->ns);
432     return c;
433   }
434
435   Chan*
436   undomount(Chan *c)
437   {
438     Chan *nc;
439     Pgrp *pg;
440     Mount *t;
441     Mhead **h, **he, *f;
442
443     pg = up->pgrp;
444     rlock(&pg->ns);
445     if(waserror()) {
446             runlock(&pg->ns);
447             nexterror();
448     }
449
450     he = &pg->mnthash[MNTHASH];
451     for(h = pg->mnthash; h < he; h++) {
452             for(f = *h; f; f = f->hash) {
453                     for(t = f->mount; t; t = t->next) {
454                             if(eqchan(c, t->to, 1)) {
455                                     /*
456                                      * We want to come out on the left hand side of the mount
457                                      * point using the element of the union that we entered on.
458                                      * To do this, find the element that has a from name of
459                                      * c->name->s.
460                                      */
461                                     if(strcmp(t->head->from->name->s, c->name->s) != 0)
462                                             continue;
463                                     nc = cclone(t->head->from, 0);
464                                     /* don't need to update nc->name because c->name is same! *
465                                     cclose(c);
466                                     c = nc;
467                                     break;
468                             }
469                     }
470             }
471     }
472     poperror();
473     runlock(&pg->ns);
474     return c;
475   }
476
```

```
477   Chan *
478   updatecname(Chan *c, char *name, int dotdot)
479   {
480     if(c->name == nil)
481             c->name = newcname(name);
482     else
483             c->name = addelem(c->name, name);
484
485     if(dotdot){
486             cleancname(c->name);    /* could be cheaper */
487             c = undomount(c);
488     }
489     return c;
490   }
491
492   int
493   walk(Chan **cp, char *name, int domnt)
494   {
495     Chan *c, *ac;
496     Mount *f;
497     int dotdot;
498
499     ac = *cp;
500
501     if(name[0] == '\0')
502             return 0;
503
504     dotdot = 0;
505     if(name[0] == '.' && name[1] == '.' && name[2] == '\0') {
506             *cp = ac = undomount(ac);
507             dotdot = 1;
508     }
509
510     ac->flag &= ~CCREATE;   /* not inherited through a walk */
511     if(devtab[ac->type]->walk(ac, name) != 0) {
512             /* walk succeeded: update name associated with *cp (ac) */
513             *cp = updatecname(*cp, name, dotdot);
514             if(domnt)
515                     *cp = domount(*cp);
516             return 0;
517     }
518
519     if(ac->mh == nil)
520             return -1;
521
522     c = nil;
523
524     rlock(&ac->mh->lock);
525     if(waserror()) {
526             runlock(&ac->mh->lock);
527             if(c)
528                     cclose(c);
529             nexterror();
530     }
531     for(f = ac->mh->mount; f; f = f->next) {
532             c = cclone(f->to, 0);
533             c->flag &= ~CCREATE;    /* not inherited through a walk */
534             if(devtab[c->type]->walk(c, name) != 0)
535                     break;
536             cclose(c);
537             c = nil;
538     }
539     poperror();
540     runlock(&ac->mh->lock);
541
542     if(c == nil)
543             return -1;
544
```

```
545     if(c->mh){
546             putmhead(c->mh);
547             c->mh = nil;
548     }
549
550     /* replace c->name by ac->name */
551     cnameclose(c->name);
552     c->name = ac->name;
553     if(ac->name)
554             incref(ac->name);
555     c = updatecname(c, name, dotdot);
556     cclose(ac);
557     *cp = c;
558
559     if(domnt)
560             *cp = domount(c);
561     return 0;
562   }
563
564   /*
565    * c is a mounted non-creatable directory.  find a creatable one.
566    */
567   Chan*
568   createdir(Chan *c)
569   {
570     Chan *nc;
571     Mount *f;
572
573     rlock(&c->mh->lock);
574     if(waserror()) {
575             runlock(&c->mh->lock);
576             nexterror();
577     }
578     for(f = c->mh->mount; f; f = f->next) {
579             if(f->to->flag&CCREATE) {
580                     nc = cclone(f->to, 0);
581                     if(nc->mh != nil)
582                             putmhead(nc->mh);
583                     nc->mh = c->mh;
584                     incref(c->mh);
585                     runlock(&c->mh->lock);
586                     poperror();
587                     cclose(c);
588                     return nc;
589             }
590     }
591     error(Enocreate);
592     return 0;
593   }
594
595   void
596   saveregisters(void)
597   {
598   }
599
600   /*
601    * In place, rewrite name to compress multiple /, eliminate ., and process ..
602    */
603   void
604   cleancname(Cname *n)
605   {
606     char *p;
607
608     if(n->s[0] == '#'){
609             p = strchr(n->s, '/');
610             if(p == nil)
611                     return;
612             cleanname(p);
613
614             /*
615              * The correct name is #i rather than #i/,
616              * but the correct name of #/ is #/.
617              */
618             if(strcmp(p, "/")==0 && n->s[1] != '/')
619                     *p = '\0';
620     }else
621             cleanname(n->s);
622     n->len = strlen(n->s);
623   }
624
625   /*
626    * Turn a name into a channel.
627    * &name[0] is known to be a valid address.  It may be a kernel address.
628    */
629   Chan*
630   namec(char *name, int amode, int omode, ulong perm)
631   {
632     Rune r;
633     char *p;
634     char *elem;
635     Cname *cname;
636     int t, n, newname;
637     int mntok, isdot;
638     Chan *c, *cc;
639     char createerr[ERRLEN];
640
641     if(name[0] == 0)
642             error(Enonexist);
643
644     if(!((ulong)name & KZERO)) {
645             p = name;
646             t = BY2PG-((ulong)p&(BY2PG-1));
647             while(vmemchr(p, 0, t) == 0) {
648                     p += t;
649                     t = BY2PG;
650             }
651     }
652
653     newname = 1;
654     cname = nil;
655     if(waserror()){
656             cnameclose(cname);
657             nexterror();
658     }
659
660     elem = up->elem;
661     mntok = 1;
662     isdot = 0;
663     switch(name[0]) {
664     case '/':
665             cname = newcname(name); /* save this before advancing */
666             name = skipslash(name);
667             c = cclone(up->slash, 0);
668             break;
669     case '#':
670             cname = newcname(name); /* save this before advancing */
671             mntok = 0;
672             elem[0] = 0;
673             n = 0;
674             while(*name && (*name != '/' || n < 2)){
675                     if(n >= NAMELEN-1)
676                             error(Efilename);
677                     elem[n++] = *name++;
678             }
679             elem[n] = '\0';
680             /*
```

```
681             *  noattach is sandboxing.
682             *
683             *  the OK exceptions are:
684             *      |  it only gives access to pipes you create
685             *      d  this process's file descriptors
686             *      e  this process's environment
687             *  the iffy exceptions are:
688             *      c  time and pid, but also cons and consctl
689             *      p  control of your own processes (and unfortunately
690             *         any others left unprotected)
691             */
692            n = chartorune(&r, elem+1)+1;
693            /* actually / is caught by parsing earlier */
694            if(utfrune("M", r))
695                    error(Enoattach);
696            if(up->pgrp->noattach && utfrune("|decp", r))
697                    error(Enoattach);
698            t = devno(r, 1);
699            if(t == -1)
700                    error(Ebadsharp);
701
702            c = devtab[t]->attach(elem+n);
703            name = skipslash(name);
704            break;
705    default:
706            cname = newcname(up->dot->name->s);
707            cname = addelem(cname, name);
708            c = cclone(up->dot, 0);
709            name = skipslash(name);
710            if(*name == 0)
711                    isdot = 1;
712    }
713
714    if(waserror()){
715            cclose(c);
716            nexterror();
717    }
718
719    name = nextelem(name, elem);
720
721    /*
722     *  If mounting, don't follow the mount entry for root or the
723     *  current directory.
724     */
725    if(mntok && !isdot && !(amode==Amount && elem[0]==0))
726            c = domount(c);                 /* see case Atodir below */
727
728    while(*name) {
729            if(walk(&c, elem, mntok) < 0)
730                    error(Enonexist);
731            name = nextelem(name, elem);
732    }
733
734    switch(amode) {
735    case Aaccess:
736            if(isdot) {
737                    c = domount(c);
738                    break;
739            }
740            if(walk(&c, elem, mntok) < 0)
741                    error(Enonexist);
742            break;
743
744    case Atodir:
745            /*
746             * Directories (e.g. for cd) are left before the mount point,
747             * so one may mount on / or . and see the effect.
748             */
749            if(walk(&c, elem, 0) < 0)
750                    error(Enonexist);
751            if(!(c->qid.path & CHDIR))
752                    error(Enotdir);
753            break;
754
755    case Aopen:
756            if(isdot)
757                    c = domount(c);
758            else {
759                    if(walk(&c, elem, mntok) < 0)
760                            error(Enonexist);
761            }
762    Open:
763            /* else error() in open has wrong value of c saved */
764            saveregisters();
765
766            if(omode == OEXEC)
767                    c->flag &= ~CCACHE;
768
769            cc = c;
770            c = devtab[c->type]->open(c, omode&~OCEXEC);
771            if(cc != c)
772                    newname = 0;
773
774            if(omode & OCEXEC)
775                    c->flag |= CCEXEC;
776            if(omode & ORCLOSE)
777                    c->flag |= CRCLOSE;
778            break;
779
780    case Amount:
781            /*
782             * When mounting on an already mounted upon directory,
783             * one wants subsequent mounts to be attached to the
784             * original directory, not the replacement.
785             */
786            if(walk(&c, elem, 0) < 0)
787                    error(Enonexist);
788            break;
789
790    case Acreate:
791            if(isdot)
792                    error(Eisdir);
793
794            /*
795             *  Walk the element before trying to create it
796             *  to see if it exists.  We clone the channel
797             *  first, just in case someone is trying to
798             *  use clwalk outside the kernel.
799             */
800            cc = cclone(c, 0);
801            if(waserror()){
802                    cclose(cc);
803                    nexterror();
804            }
805            nameok(elem, 0);
806            if(walk(&cc, elem, 1) == 0){
807                    poperror();
808                    cclose(c);
809                    c = cc;
810                    omode |= OTRUNC;
811                    goto Open;
812            }
813            cclose(cc);
814            poperror();
815
816            /*
```

```
817             *  the file didn't exist, try the create
818             */
819            if(c->mh != nil && !(c->flag&CCREATE))
820                    c = createdir(c);
821
822            /*
823             * protect against the open/create race.
824             * This is not a complete fix. It just reduces the window.
825             */
826            if(waserror()) {
827                    strcpy(createerr, up->error);
828                    if(walk(&c, elem, 1) < 0)
829                            error(createerr);
830                    omode |= OTRUNC;
831                    goto Open;
832            }
833            devtab[c->type]->create(c, elem, omode&~OCEXEC, perm);
834            if(omode & OCEXEC)
835                    c->flag |= CCEXEC;
836            if(omode & ORCLOSE)
837                    c->flag |= CRCLOSE;
838            poperror();
839            break;
840
841    default:
842            panic("unknown namec access %d\n", amode);
843    }
844
845    poperror();
846
847    if(newname){
848            cleancname(cname);
849            cnameclose(c->name);
850            c->name = cname;
851    }else
852            cnameclose(cname);
853
854    poperror();
855    return c;
856  }
857
858  /*
859   * name[0] is addressable.
860   */
861  char*
862  skipslash(char *name)
863  {
864      Again:
865    while(*name == '/')
866            name++;
867    if(*name=='.' && (name[1]==0 || name[1]=='/')){
868            name++;
869            goto Again;
870    }
871    return name;
872  }
873
874  char isfrog[256]={
875    /*NUL*/ 1, 1, 1, 1, 1, 1, 1, 1,
876    /*BKS*/ 1, 1, 1, 1, 1, 1, 1, 1,
877    /*DLE*/ 1, 1, 1, 1, 1, 1, 1, 1,
878    /*CAN*/ 1, 1, 1, 1, 1, 1, 1, 1,
879  /*        [' ']   1,      rob - let's try this out */
880    ['/']   1,
881    [0x7f]  1,
882  };
883
884  void
885  nameok(char *elem, int slashok)
886  {
887    char *eelem;
888
889    eelem = elem+NAMELEN;
890    while(*elem) {
891            if(isfrog[*(uchar*)elem])
892                    if(!slashok || *elem!='/')
893                            error(Ebadchar);
894            elem++;
895            if(elem >= eelem)
896                    error(Efilename);
897    }
898  }
899
900  /*
901   * name[0] should not be a slash.
902   */
903  char*
904  nextelem(char *name, char *elem)
905  {
906    int w;
907    char *end;
908    Rune r;
909
910    if(*name == '/')
911            error(Efilename);
912    end = utfrune(name, '/');
913    if(end == 0)
914            end = strchr(name, 0);
915    w = end-name;
916    if(w >= NAMELEN)
917            error(Efilename);
918    memmove(elem, name, w);
919    elem[w] = 0;
920    while(name < end){
921            name += chartorune(&r, name);
922            if(r<sizeof(isfrog) && isfrog[r])
923                    error(Ebadchar);
924    }
925    return skipslash(name);
926  }
927
928  void
929  isdir(Chan *c)
930  {
931    if(c->qid.path & CHDIR)
932            return;
933    error(Enotdir);
934  }
935
936  void
937  putmhead(Mhead *m)
938  {
939    if(decref(m) == 0)
940            free(m);
941  }
```

```
#include  "u.h"
#include  "../port/lib.h"
#include  "mem.h"
#include  "dat.h"
#include  "fns.h"
#include  "../port/error.h"

extern ulong      kerndate;

int
devno(int c, int user)
{
  int i;

  for(i = 0; devtab[i] != nil; i++) {
          if(devtab[i]->dc == c)
                  return i;
  }
  if(user == 0)
          panic("devno %C 0x%ux", c, c);

  return -1;
}

void
devdir(Chan *c, Qid qid, char *n, vlong length, char *user, long perm, Dir *db)
{
  strcpy(db->name, n);
  db->qid = qid;
  db->type = devtab[c->type]->dc;
  db->dev = c->dev;
  if(qid.path & CHDIR)
          db->mode = CHDIR|perm;
  else
          db->mode = perm;
  if(c->flag&CMSG)
          db->mode |= CHMOUNT;
  db->atime = seconds();
  db->mtime = kerndate;
  db->length = length;
  memmove(db->uid, user, NAMELEN);
  memmove(db->gid, eve, NAMELEN);
}

//
// the zeroth element of the table MUST be the directory itself for ..
//
int
devgen(Chan *c, Dirtab *tab, int ntab, int i, Dir *dp)
{
  if(tab==0)
          return -1;
  if(i!=DEVDOTDOT){
          if(i>=ntab)
                  return -1;
          tab += i;
  }
  devdir(c, tab->qid, tab->name, tab->length, eve, tab->perm, dp);
  return 1;
}

void
devreset(void)
{
}

void
devinit(void)
{
}

Chan*
devattach(int tc, char *spec)
{
  Chan *c;
  char buf[NAMELEN+4];

  c = newchan();
  c->qid = (Qid){CHDIR, 0};
  c->type = devno(tc, 0);
  sprint(buf, "#%C%s", tc, spec==nil? "" : spec);
  c->name = newcname(buf);
  return c;
}

Chan*
devclone(Chan *c, Chan *nc)
{
  if(c->flag & COPEN)
          panic("clone of open file type %C\n", devtab[c->type]->dc);

  if(nc == 0)
          nc = newchan();

  nc->type = c->type;
  nc->dev = c->dev;
  nc->mode = c->mode;
  nc->qid = c->qid;
  nc->offset = c->offset;
  nc->flag = c->flag;
  nc->mh = c->mh;
  if(c->mh != nil)
          incref(c->mh);
  nc->mountid = c->mountid;
  nc->aux = c->aux;
  nc->mchan = c->mchan;
  nc->mqid = c->mqid;
  nc->mcp = c->mcp;
  return nc;
}

int
devwalk(Chan *c, char *name, Dirtab *tab, int ntab, Devgen *gen)
{
  long i;
  Dir dir;

  isdir(c);
  if(name[0]=='.' && name[1]==0)
          return 1;
  if(name[0]=='.' && name[1]=='.' && name[2]==0){
          (*gen)(c, tab, ntab, DEVDOTDOT, &dir);
          c->qid = dir.qid;
          return 1;
  }
  for(i=0;; i++) {
          switch((*gen)(c, tab, ntab, i, &dir)){
          case -1:
                  strncpy(up->error, Enonexist, NAMELEN);
                  return 0;
          case 0:
                  continue;
          case 1:
                  if(strcmp(name, dir.name) == 0){
                          c->qid = dir.qid;
                          return 1;
```

```
                }
                continue;
        }
  }
  return 0;
}

void
devstat(Chan *c, char *db, Dirtab *tab, int ntab, Devgen *gen)
{
  int i;
  Dir dir;
  char *p, *elem;

  for(i=0;; i++)
        switch((*gen)(c, tab, ntab, i, &dir)){
        case -1:
                if(c->qid.path & CHDIR){
                        if(c->name == nil)
                                elem = "???";
                        else if(strcmp(c->name->s, "/") == 0)
                                elem = "/";
                        else
                                for(elem=p=c->name->s; *p; p++)
                                        if(*p == '/')
                                                elem = p+1;
                        devdir(c, c->qid, elem, i*DIRLEN, eve, CHDIR|0555, &dir);
                        convD2M(&dir, db);
                        return;
                }
                print("%s %s: devstat %C %lux\n",
                        up->text, up->user,
                        devtab[c->type]->dc, c->qid.path);

                error(Enonexist);
        case 0:
                break;
        case 1:
                if(c->qid.path == dir.qid.path) {
                        if(c->flag&CMSG)
                                dir.mode |= CHMOUNT;
                        convD2M(&dir, db);
                        return;
                }
                break;
        }
}

long
devdirread(Chan *c, char *d, long n, Dirtab *tab, int ntab, Devgen *gen)
{
  long k, m;
  Dir dir;

  k = c->offset/DIRLEN;
  for(m=0; m<n; k++) {
        switch((*gen)(c, tab, ntab, k, &dir)){
        case -1:
                return m;

        case 0:
                c->offset += DIRLEN;
                break;

        case 1:
                convD2M(&dir, d);
                m += DIRLEN;
                d += DIRLEN;
                break;
        }
  }

  return m;
}

Chan*
devopen(Chan *c, int omode, Dirtab *tab, int ntab, Devgen *gen)
{
  int i;
  Dir dir;
  ulong t, mode;
  static int access[] = { 0400, 0200, 0600, 0100 };

  for(i=0;; i++) {
        switch((*gen)(c, tab, ntab, i, &dir)){
        case -1:
                goto Return;
        case 0:
                break;
        case 1:
                if(c->qid.path == dir.qid.path) {
                        if(strcmp(up->user, dir.uid) == 0)
                                mode = dir.mode;
                        else
                        if(strcmp(up->user, eve) == 0)
                                mode = dir.mode<<3;
                        else
                                mode = dir.mode<<6;

                        t = access[omode&3];
                        if((t & mode) == t)
                                goto Return;
                        error(Eperm);
                }
                break;
        }
  }
Return:
  c->offset = 0;
  if((c->qid.path&CHDIR) && omode!=OREAD)
        error(Eperm);
  c->mode = openmode(omode);
  c->flag |= COPEN;
  return c;
}

void
devcreate(Chan*, char*, int, ulong)
{
  error(Eperm);
}

Block*
devbread(Chan *c, long n, ulong offset)
{
  Block *bp;

  bp = allocb(n);
  if(bp == 0)
        error(Enomem);
  if(waserror()) {
        freeb(bp);
        nexterror();
  }
  bp->wp += devtab[c->type]->read(c, bp->wp, n, offset);
  poperror();
```

```
273     return bp;
274   }
275
276   long
277   devbwrite(Chan *c, Block *bp, ulong offset)
278   {
279     long n;
280
281     n = devtab[c->type]->write(c, bp->rp, BLEN(bp), offset);
282     freeb(bp);
283
284     return n;
285   }
286
287   void
288   devremove(Chan*)
289   {
290     error(Eperm);
291   }
292
293   void
294   devwstat(Chan*, char*)
295   {
296     error(Eperm);
297   }
```

```
#include  "u.h"
#include  "../port/lib.h"
#include  "mem.h"
#include  "dat.h"
#include  "fns.h"
#include  "../port/error.h"

void      (*consdebug)(void) = nil;

Queue*    kbdq;                   /* unprocessed console input */
Queue*    lineq;                  /* processed console input */
Queue*    printq;                 /* console output */

static struct
{
  QLock;

  int     raw;            /* true if we shouldn't process input */
  int     ctl;            /* number of opens to the control file */
  int     x;              /* index into line */
  char    line[1024];     /* current input line */

  Rune    c;
  int     count;
  int     repeat;
  int     ctlpoff;
} kbd;

char      sysname[NAMELEN];
vlong     fasthz;

static ulong      randomread(void*, ulong);
static void       randominit(void);
static void       seedrand(void);
static int        readtime(ulong, char*, int);
static int        readbintime(char*, int);
static int        writetime(char*, int);
static int        writebintime(char*, int);

void
printinit(void)
{
  lineq = qopen(2*1024, 0, 0, 0);
  if(lineq == nil)
          panic("printinit");
  qnoblock(lineq, 1);
}

int
consactive(void)
{
  if(printq)
          return qlen(printq) > 0;
  return 0;
}

void
prflush(void)
{
  while(consactive())
          ;
}

/*
 *   Print a string on the console.  Convert \n to \r\n for serial
 *   line consoles.  Locking of the queues is left up to the screen
 *   or uart code.  Multi-line messages to serial consoles may get
 *   interspersed with other messages.
 */
static void
putstrn0(char *str, int n, int usewrite)
{
  int m;
  char *t;
  char buf[PRINTSIZE+2];

  /*
   *  if there's an attached bit mapped display,
   *  put the message there.  screenputs is defined
   *  as a null macro for systems that have no such
   *  display.
   */
  screenputs(str, n);

  /*
   *  if there's a serial line being used as a console,
   *  put the message there.
   */
  if(printq == 0)
          return;

  while(n > 0) {
          t = memchr(str, '\n', n);
          if(t) {
                  m = t - str;
                  memmove(buf, str, m);
                  buf[m] = '\r';
                  buf[m+1] = '\n';
                  if(usewrite)
                          qwrite(printq, buf, m+2);
                  else
                          qiwrite(printq, buf, m+2);
                  str = t + 1;
                  n -= m + 1;
          } else {
                  if(usewrite)
                          qwrite(printq, str, n);
                  else
                          qiwrite(printq, str, n);
                  break;
          }
  }
}

void
putstrn(char *str, int n)
{
  putstrn0(str, n, 0);
}

int
snprint(char *s, int n, char *fmt, ...)
{
  va_list arg;

  va_start(arg, fmt);
  n = doprint(s, s+n, fmt, arg) - s;
  va_end(arg);
  return n;
}

int
sprint(char *s, char *fmt, ...)
{
  int n;
  va_list arg;
```

```
137
138     va_start(arg, fmt);
139     n = doprint(s, s+PRINTSIZE, fmt, arg) - s;
140     va_end(arg);
141     return n;
142   }
143
144   int noprint;
145
146   int
147   print(char *fmt, ...)
148   {
149     int n;
150     va_list arg;
151     char buf[PRINTSIZE];
152
153     if(noprint)
154             return -1;
155
156     va_start(arg, fmt);
157     n = doprint(buf, buf+sizeof(buf), fmt, arg) - buf;
158     va_end(arg);
159     putstrn(buf, n);
160
161     return n;
162   }
163
164   int
165   iprint(char *fmt, ...)
166   {
167     int n, s;
168     va_list arg;
169     char buf[PRINTSIZE];
170
171     s = splhi();
172     va_start(arg, fmt);
173     n = doprint(buf, buf+sizeof(buf), fmt, arg) - buf;
174     va_end(arg);
175     serialputs(buf, n);
176   //        screenputs(buf, n);
177     splx(s);
178
179     return n;
180   }
181
182   void
183   panic(char *fmt, ...)
184   {
185     int n;
186     va_list arg;
187     char buf[PRINTSIZE];
188
189     splhi();
190     strcpy(buf, "panic: ");
191     va_start(arg, fmt);
192     n = doprint(buf+strlen(buf), buf+sizeof(buf), fmt, arg) - buf;
193     va_end(arg);
194     buf[n] = '\n';
195     serialputs(buf, n+1);
196     if(consdebug)
197             consdebug();
198     putstrn(buf, n+1);
199     spllo();
200     prflush();
201     dumpstack();
202
203     exit(1);
204   }
205
206   void
207   _assert(char *fmt)
208   {
209     panic("assert failed: %s", fmt);
210   }
211
212   int
213   pprint(char *fmt, ...)
214   {
215     int n;
216     Chan *c;
217     va_list arg;
218     char buf[2*PRINTSIZE];
219
220     if(up == nil || up->fgrp == nil)
221             return 0;
222
223     c = up->fgrp->fd[2];
224     if(c==0 || (c->mode!=OWRITE && c->mode!=ORDWR))
225             return 0;
226     n = sprint(buf, "%s %lud: ", up->text, up->pid);
227     va_start(arg, fmt);
228     n = doprint(buf+n, buf+sizeof(buf), fmt, arg) - buf;
229     va_end(arg);
230
231     if(waserror())
232             return 0;
233     devtab[c->type]->write(c, buf, n, c->offset);
234     poperror();
235
236     lock(c);
237     c->offset += n;
238     unlock(c);
239
240     return n;
241   }
242
243   void
244   echo(Rune r, char *buf, int n)
245   {
246     static int ctrlt, pid;
247     extern ulong etext;
248     int x;
249
250     /*
251      * ^p hack
252      */
253     if(r==0x10 && cpuserver && !kbd.ctlpoff){
254             lock(&active);
255             active.exiting = 1;
256             unlock(&active);
257     }
258
259     /*
260      * ^t hack BUG
261      */
262     if(ctrlt == 2){
263             ctrlt = 0;
264             switch(r){
265             case 0x14:
266                     break;  /* pass it on */
267             case 's':
268                     dumpstack();
269                     break;
270             case 'x':
271                     xsummary();
272                     ixsummary();
```

```
273                     mallocsummary();
274                     pagersummary();
275                     break;
276             case 'd':
277                     if(consdebug == nil)
278                             consdebug = rdb;
279                     else
280                             consdebug = nil;
281                     print("consdebug now 0x%p\n", consdebug);
282                     return;
283             case 'D':
284                     if(consdebug == nil)
285                             consdebug = rdb;
286                     consdebug();
287                     return;
288             case 'p':
289                     x = spllo();
290                     procdump();
291                     splx(x);
292                     return;
293             case 'q':
294                     scheddump();
295                     break;
296             case 'k':
297                     if(!cpuserver)
298                             killbig();
299                     break;
300             case 'r':
301                     exit(0);
302                     break;
303             }
304     }
305     else if(r == 0x14){
306             ctrlt++;
307             return;
308     }
309     ctrlt = 0;
310     if(kbd.raw)
311             return;
312
313     /*
314      *  finally, the actual echoing
315      */
316     if(r == '\n'){
317             if(printq)
318                     qiwrite(printq, "\r", 1);
319     } else if(r == 0x15){
320             buf = "^U\n";
321             n = 3;
322     }
323     screenputs(buf, n);
324     if(printq)
325             qiwrite(printq, buf, n);
326   }
327
328   /*
329    *  Called by a uart interrupt for console input.
330    *
331    *  turn '\r' into '\n' before putting it into the queue.
332    */
333   int
334   kbdcr2nl(Queue *q, int ch)
335   {
336     if(ch == '\r')
337             ch = '\n';
338     return kbdputc(q, ch);
339   }
340
```

```
341   /*
342    *  Put character, possibly a rune, into read queue at interrupt time.
343    *  Called at interrupt time to process a character.
344    */
345   int
346   kbdputc(Queue*, int ch)
347   {
348     int n;
349     char buf[3];
350     Rune r;
351
352     r = ch;
353     n = runetochar(buf, &r);
354     if(n == 0)
355             return 0;
356     echo(r, buf, n);
357     kbd.c = r;
358     qproduce(kbdq, buf, n);
359     return 0;
360   }
361
362   void
363   kbdrepeat(int rep)
364   {
365     kbd.repeat = rep;
366     kbd.count = 0;
367   }
368
369   void
370   kbdclock(void)
371   {
372     if(kbd.repeat == 0)
373             return;
374     if(kbd.repeat==1 && ++kbd.count>HZ){
375             kbd.repeat = 2;
376             kbd.count = 0;
377             return;
378     }
379     if(++kbd.count&1)
380             kbdputc(kbdq, kbd.c);
381   }
382
383   enum{
384     Qdir,
385     Qauth,
386     Qauthcheck,
387     Qauthent,
388     Qbintime,
389     Qcons,
390     Qconsctl,
391     Qcputime,
392     Qdrivers,
393     Qkey,
394     Qhostdomain,
395     Qhostowner,
396     Qnull,
397     Qpgrpid,
398     Qpid,
399     Qppid,
400     Qrandom,
401     Qreboot,
402     Qswap,
403     Qsysname,
404     Qsysstat,
405     Qtime,
406     Quser,
407     Qzero,
408   };
```

```
409
410  enum
411  {
412    VLNUMSIZE=      22,
413  };
414
415  static Dirtab consdir[]={
416    "authenticate", {Qauth},        0,              0666,
417    "authcheck",    {Qauthcheck},   0,              0666,
418    "authenticator", {Qauthent},    0,              0666,
419    "bintime",      {Qbintime},     24,             0664,
420    "cons",         {Qcons},        0,              0660,
421    "consctl",      {Qconsctl},     0,              0220,
422    "cputime",      {Qcputime},     6*NUMSIZE,      0444,
423    "drivers",      {Qdrivers},     0,              0644,
424    "hostdomain",   {Qhostdomain},  DOMLEN,         0664,
425    "hostowner",    {Qhostowner},   NAMELEN,        0664,
426    "key",          {Qkey},         DESKEYLEN,      0622,
427    "null",         {Qnull},        0,              0666,
428    "pgrpid",       {Qpgrpid},      NUMSIZE,        0444,
429    "pid",          {Qpid},         NUMSIZE,        0444,
430    "ppid",         {Qppid},        NUMSIZE,        0444,
431    "random",       {Qrandom},      0,              0664,
432    "reboot",       {Qreboot},      0,              0664,
433    "swap",         {Qswap},        0,              0664,
434    "sysname",      {Qsysname},     0,              0664,
435    "sysstat",      {Qsysstat},     0,              0666,
436    "time",         {Qtime},        NUMSIZE+3*VLNUMSIZE,    0664,
437    "user",         {Quser},        NAMELEN,        0666,
438    "zero",         {Qzero},        0,              0444,
439  };
440
441  int
442  readnum(ulong off, char *buf, ulong n, ulong val, int size)
443  {
444    char tmp[64];
445
446    snprint(tmp, sizeof(tmp), "%*.0lud", size-1, val);
447    tmp[size-1] = ' ';
448    if(off >= size)
449            return 0;
450    if(off+n > size)
451            n = size-off;
452    memmove(buf, tmp+off, n);
453    return n;
454  }
455
456  int
457  readstr(ulong off, char *buf, ulong n, char *str)
458  {
459    int size;
460
461    size = strlen(str);
462    if(off >= size)
463            return 0;
464    if(off+n > size)
465            n = size-off;
466    memmove(buf, str+off, n);
467    return n;
468  }
469
470  static void
471  consinit(void)
472  {
473    todinit();
474    randominit();
475  }
476
477  static Chan*
478  consattach(char *spec)
479  {
480    return devattach('c', spec);
481  }
482
483  static int
484  conswalk(Chan *c, char *name)
485  {
486    return devwalk(c, name, consdir, nelem(consdir), devgen);
487  }
488
489  static void
490  consstat(Chan *c, char *dp)
491  {
492    devstat(c, dp, consdir, nelem(consdir), devgen);
493  }
494
495  static Chan*
496  consopen(Chan *c, int omode)
497  {
498    c->aux = 0;
499    switch(c->qid.path){
500    case Qconsctl:
501            if(!iseve())
502                    error(Eperm);
503            qlock(&kbd);
504            kbd.ctl++;
505            qunlock(&kbd);
506            break;
507    }
508    return devopen(c, omode, consdir, nelem(consdir), devgen);
509  }
510
511  static void
512  consclose(Chan *c)
513  {
514    /* last close of control file turns off raw */
515    switch(c->qid.path){
516    case Qconsctl:
517            if(c->flag&COPEN){
518                    qlock(&kbd);
519                    if(--kbd.ctl == 0)
520                            kbd.raw = 0;
521                    qunlock(&kbd);
522            }
523            break;
524    case Qauth:
525    case Qauthcheck:
526    case Qauthent:
527            authclose(c);
528    }
529  }
530
531  static long
532  consread(Chan *c, void *buf, long n, vlong off)
533  {
534    ulong l;
535    Mach *mp;
536    char *b, *bp;
537    char tmp[128];          /* must be >= 6*NUMSIZE */
538    char *cbuf = buf;
539    int ch, i, k, id, eol;
540    vlong offset = off;
541
542    if(n <= 0)
543            return n;
544    switch(c->qid.path & ~CHDIR){
```

```
545     case Qdir:
546             return devdirread(c, buf, n, consdir, nelem(consdir), devgen);
547
548     case Qcons:
549             qlock(&kbd);
550             if(waserror()) {
551                     qunlock(&kbd);
552                     nexterror();
553             }
554             if(kbd.raw) {
555                     if(qcanread(lineq))
556                             n = qread(lineq, buf, n);
557                     else {
558                             /* read as much as possible */
559                             do {
560                                     i = qread(kbdq, cbuf, n);
561                                     cbuf += i;
562                                     n -= i;
563                             } while (n>0 && qcanread(kbdq));
564                             n = cbuf - (char*)buf;
565                     }
566             } else {
567                     while(!qcanread(lineq)) {
568                             qread(kbdq, &kbd.line[kbd.x], 1);
569                             ch = kbd.line[kbd.x];
570                             if(kbd.raw){
571                                     qiwrite(lineq, kbd.line, kbd.x+1);
572                                     kbd.x = 0;
573                                     continue;
574                             }
575                             eol = 0;
576                             switch(ch){
577                             case '\b':
578                                     if(kbd.x)
579                                             kbd.x--;
580                                     break;
581                             case 0x15:
582                                     kbd.x = 0;
583                                     break;
584                             case '\n':
585                             case 0x04:
586                                     eol = 1;
587                             default:
588                                     kbd.line[kbd.x++] = ch;
589                                     break;
590                             }
591                             if(kbd.x == sizeof(kbd.line) || eol){
592                                     if(ch == 0x04)
593                                             kbd.x--;
594                                     qwrite(lineq, kbd.line, kbd.x);
595                                     kbd.x = 0;
596                             }
597                     }
598                     n = qread(lineq, buf, n);
599             }
600             qunlock(&kbd);
601             poperror();
602             return n;
603
604     case Qcputime:
605             k = offset;
606             if(k >= 6*NUMSIZE)
607                     return 0;
608             if(k+n > 6*NUMSIZE)
609                     n = 6*NUMSIZE - k;
610             /* easiest to format in a separate buffer and copy out */
611             for(i=0; i<6 && NUMSIZE*i<k+n; i++){
612                     l = up->time[i];
613                     if(i == TReal)
614                             l = MACHP(0)->ticks - l;
615                     l = TK2MS(l);
616                     readnum(0, tmp+NUMSIZE*i, NUMSIZE, l, NUMSIZE);
617             }
618             memmove(buf, tmp+k, n);
619             return n;
620
621     case Qpgrpid:
622             return readnum((ulong)offset, buf, n, up->pgrp->pgrpid, NUMSIZE);
623
624     case Qpid:
625             return readnum((ulong)offset, buf, n, up->pid, NUMSIZE);
626
627     case Qppid:
628             return readnum((ulong)offset, buf, n, up->parentpid, NUMSIZE);
629
630     case Qtime:
631             return readtime((ulong)offset, buf, n);
632
633     case Qbintime:
634             return readbintime(buf, n);
635
636     case Qkey:
637             return keyread(buf, n, offset);
638
639     case Qauth:
640             return authread(c, cbuf, n);
641
642     case Qauthcheck:
643             return authcheckread(c, cbuf, n);
644
645     case Qauthent:
646             return authentread(c, cbuf, n);
647
648     case Qhostowner:
649             return readstr((ulong)offset, buf, n, eve);
650
651     case Qhostdomain:
652             return readstr((ulong)offset, buf, n, hostdomain);
653
654     case Quser:
655             return readstr((ulong)offset, buf, n, up->user);
656
657     case Qnull:
658             return 0;
659
660     case Qsysstat:
661             b = smalloc(conf.nmach*(NUMSIZE*8+1) + 1);      /* +1 for NUL */
662             bp = b;
663             for(id = 0; id < 32; id++) {
664                     if(active.machs & (1<<id)) {
665                             mp = MACHP(id);
666                             readnum(0, bp, NUMSIZE, id, NUMSIZE);
667                             bp += NUMSIZE;
668                             readnum(0, bp, NUMSIZE, mp->cs, NUMSIZE);
669                             bp += NUMSIZE;
670                             readnum(0, bp, NUMSIZE, mp->intr, NUMSIZE);
671                             bp += NUMSIZE;
672                             readnum(0, bp, NUMSIZE, mp->syscall, NUMSIZE);
673                             bp += NUMSIZE;
674                             readnum(0, bp, NUMSIZE, mp->pfault, NUMSIZE);
675                             bp += NUMSIZE;
676                             readnum(0, bp, NUMSIZE, mp->tlbfault, NUMSIZE);
677                             bp += NUMSIZE;
678                             readnum(0, bp, NUMSIZE, mp->tlbpurge, NUMSIZE);
679                             bp += NUMSIZE;
680                             readnum(0, bp, NUMSIZE, mp->load, NUMSIZE);
```

```
681                             bp += NUMSIZE;
682                             *bp++ = '\n';
683                     }
684             }
685             n = readstr((ulong)offset, buf, n, b);
686             free(b);
687             return n;
688
689     case Qswap:
690             sprint(tmp, "%lud/%lud memory %lud/%lud swap\n",
691                     palloc.user-palloc.freecount,
692                     palloc.user, conf.nswap-swapalloc.free, conf.nswap);
693
694             return readstr((ulong)offset, buf, n, tmp);
695
696     case Qsysname:
697             return readstr((ulong)offset, buf, n, sysname);
698
699     case Qrandom:
700             return randomread(buf, n);
701
702     case Qdrivers:
703             b = malloc(READSTR);
704             if(b == nil)
705                     error(Enomem);
706             n = 0;
707             for(i = 0; devtab[i] != nil; i++)
708                     n += snprint(b+n, READSTR-n, "#%C %s\n", devtab[i]->dc,  devtab[i]->name);
709             n = readstr((ulong)offset, buf, n, b);
710             free(b);
711             return n;
712
713     case Qzero:
714             memset(buf, 0, n);
715             return n;
716
717     default:
718             print("consread %lux\n", c->qid.path);
719             error(Egreg);
720     }
721     return -1;              /* never reached */
722   }
723
724   static long
725   conswrite(Chan *c, void *va, long n, vlong off)
726   {
727     char buf[256];
728     long l, bp;
729     char *a = va;
730     Mach *mp;
731     int id, fd;
732     Chan *swc;
733     ulong offset = off;
734
735     switch(c->qid.path){
736     case Qcons:
737             /*
738              * Can't page fault in putstrn, so copy the data locally.
739              */
740             l = n;
741             while(l > 0){
742                     bp = l;
743                     if(bp > sizeof buf)
744                             bp = sizeof buf;
745                     memmove(buf, a, bp);
746                     putstrn0(a, bp, 1);
747                     a += bp;
748                     l -= bp;
749             }
750             break;
751
752     case Qconsctl:
753             if(n >= sizeof(buf))
754                     n = sizeof(buf)-1;
755             strncpy(buf, a, n);
756             buf[n] = 0;
757             for(a = buf; a;){
758                     if(strncmp(a, "rawon", 5) == 0){
759                             qlock(&kbd);
760                             if(kbd.x){
761                                     qwrite(kbdq, kbd.line, kbd.x);
762                                     kbd.x = 0;
763                             }
764                             kbd.raw = 1;
765                             qunlock(&kbd);
766                     } else if(strncmp(a, "rawoff", 6) == 0){
767                             kbd.raw = 0;
768                             kbd.x = 0;
769                     } else if(strncmp(a, "ctlpon", 6) == 0){
770                             kbd.ctlpoff = 0;
771                     } else if(strncmp(a, "ctlpoff", 7) == 0){
772                             kbd.ctlpoff = 1;
773                     }
774                     if(a = strchr(a, ' '))
775                             a++;
776             }
777             break;
778
779     case Qtime:
780             if(!iseve())
781                     error(Eperm);
782             return writetime(a, n);
783
784     case Qbintime:
785             if(!iseve())
786                     error(Eperm);
787             return writebintime(a, n);
788
789     case Qkey:
790             return keywrite(a, n);
791
792     case Qhostowner:
793             return hostownerwrite(a, n);
794
795     case Qhostdomain:
796             return hostdomainwrite(a, n);
797
798     case Quser:
799             return userwrite(a, n);
800
801     case Qauth:
802             return authwrite(c, a, n);
803
804     case Qauthcheck:
805             return authcheck(c, a, n);
806
807     case Qauthent:
808             return authentwrite(c, a, n);
809
810     case Qnull:
811             break;
812
813     case Qreboot:
814             if(!iseve())
815                     error(Eperm);
816             if(strncmp(a, "reboot", 6) == 0){
```

```
817                     print("conswrite: reboot\n");
818                     exit(0);
819             }
820             if(strncmp(a, "malloc", 6) == 0){       /* rsc bug */
821                     a = malloc(2);
822                     strcpy(a, "hi");
823                     free(a);
824                     a = malloc(2);
825                     strcpy(a, "helo");
826                     free(a);
827                     panic("not reached conswrite");
828             }
829             if(strncmp(a, "panic", 5) == 0)
830                     panic("/dev/reboot");
831             break;
832
833     case Qsysstat:
834             for(id = 0; id < 32; id++) {
835                     if(active.machs & (1<<id)) {
836                             mp = MACHP(id);
837                             mp->cs = 0;
838                             mp->intr = 0;
839                             mp->syscall = 0;
840                             mp->pfault = 0;
841                             mp->tlbfault = 0;
842                             mp->tlbpurge = 0;
843                     }
844             }
845             break;
846
847     case Qswap:
848             if(n >= sizeof buf)
849                     error(Egreg);
850             memmove(buf, va, n);    /* so we can NUL-terminate */
851             buf[n] = 0;
852             /* start a pager if not already started */
853             if(strncmp(buf, "start", 5) == 0){
854                     kickpager();
855                     break;
856             }
857             if(cpuserver && !iseve())
858                     error(Eperm);
859             if(buf[0]<'0' || '9'<buf[0])
860                     error(Ebadarg);
861             fd = strtoul(buf, 0, 0);
862             swc = fdtochan(fd, -1, 1, 1);
863             setswapchan(swc);
864             break;
865
866     case Qsysname:
867             if(offset != 0)
868                     error(Ebadarg);
869             if(n <= 0 || n >= NAMELEN)
870                     error(Ebadarg);
871             strncpy(sysname, a, n);
872             sysname[n] = 0;
873             if(sysname[n-1] == '\n')
874                     sysname[n-1] = 0;
875             break;
876
877     default:
878             print("conswrite: %lud\n", c->qid.path);
879             error(Egreg);
880     }
881     return n;
882   }
883
884   void
885   setterm(char *f)
886   {
887     char buf[2*NAMELEN];
888
889     sprint(buf, f, conffile);
890     ksetenv("terminal", buf);
891   }
892
893   Dev consdevtab = {
894     'c',
895     "cons",
896
897     devreset,
898     consinit,
899     consattach,
900     devclone,
901     conswalk,
902     consstat,
903     consopen,
904     devcreate,
905     consclose,
906     consread,
907     devbread,
908     conswrite,
909     devbwrite,
910     devremove,
911     devwstat,
912   };
913
914   struct Rb
915   {
916     QLock;
917     Rendez  producer;
918     Rendez  consumer;
919     ulong   randomcount;
920     uchar   buf[1024];
921     uchar   *ep;
922     uchar   *rp;
923     uchar   *wp;
924     uchar   next;
925     uchar   wakeme;
926     ushort  bits;
927     ulong   randn;
928   } rb;
929
930   static void
931   seedrand(void)
932   {
933     randomread((void*)&rb.randn, sizeof(rb.randn));
934   }
935
936   int
937   nrand(int n)
938   {
939     if(rb.randn == 0)
940             seedrand();
941     rb.randn = rb.randn*1103515245 + 12345 + MACHP(0)->ticks;
942     return (rb.randn>>16) % n;
943   }
944
945   int
946   rand(void)
947   {
948     nrand(1);
949     return rb.randn;
950   }
951
952
```

```
static int
rbnotfull(void*)
{
  int i;

  i = rb.rp - rb.wp;
  return i != 1 && i != (1 - sizeof(rb.buf));
}

static int
rbnotempty(void*)
{
  return rb.wp != rb.rp;
}

void
genrandom(void*)
{
  up->basepri = PriNormal;
  up->priority = up->basepri;

  for(;;){
          for(;;)
                  if(++rb.randomcount > 100000)
                          break;
          if(anyhigher())
                  sched();
          if(!rbnotfull(0))
                  sleep(&rb.producer, rbnotfull, 0);
  }
}

/*
 *  produce random bits in a circular buffer
 */
static void
randomclock(void)
{
  if(rb.randomcount == 0 || !rbnotfull(0))
          return;

  rb.bits = (rb.bits<<2) ^ rb.randomcount;
  rb.randomcount = 0;

  rb.next++;
  if(rb.next != 8/2)
          return;
  rb.next = 0;

  *rb.wp ^= rb.bits;
  if(rb.wp+1 == rb.ep)
          rb.wp = rb.buf;
  else
          rb.wp = rb.wp+1;

  if(rb.wakeme)
          wakeup(&rb.consumer);
}

static void
randominit(void)
{
  addclock0link(randomclock);
  rb.ep = rb.buf + sizeof(rb.buf);
  rb.rp = rb.wp = rb.buf;
  kproc("genrandom", genrandom, 0);
}

/*
 *  consume random bytes from a circular buffer
 */
static ulong
randomread(void *xp, ulong n)
{
  uchar *e, *p;
  ulong x;

  p = xp;

  if(waserror()){
          qunlock(&rb);
          nexterror();
  }

  qlock(&rb);
  for(e = p + n; p < e; ){
          if(rb.wp == rb.rp){
                  rb.wakeme = 1;
                  wakeup(&rb.producer);
                  sleep(&rb.consumer, rbnotempty, 0);
                  rb.wakeme = 0;
                  continue;
          }

          /*
           *  beating clocks will be precictable if
           *  they are synchronized.  Use a cheap pseudo
           *  random number generator to obscure any cycles.
           */
          x = rb.randn*1103515245 ^ *rb.rp;
          *p++ = rb.randn = x;

          if(rb.rp+1 == rb.ep)
                  rb.rp = rb.buf;
          else
                  rb.rp = rb.rp+1;
  }
  qunlock(&rb);
  poperror();

  wakeup(&rb.producer);

  return n;
}

static uvlong uvorder = 0x0001020304050607ULL;

static uchar*
le2vlong(vlong *to, uchar *f)
{
  uchar *t, *o;
  int i;

  t = (uchar*)to;
  o = (uchar*)&uvorder;
  for(i = 0; i < sizeof(vlong); i++)
          t[o[i]] = f[i];
  return f+sizeof(vlong);
}

static uchar*
vlong2le(uchar *t, vlong from)
{
  uchar *f, *o;
  int i;
```

```
1089    f = (uchar*)&from;
1090    o = (uchar*)&uvorder;
1091    for(i = 0; i < sizeof(vlong); i++)
1092            t[i] = f[o[i]];
1093    return t+sizeof(vlong);
1094  }
1095
1096  static long order = 0x00010203;
1097
1098  static uchar*
1099  le2long(long *to, uchar *f)
1100  {
1101    uchar *t, *o;
1102    int i;
1103
1104    t = (uchar*)to;
1105    o = (uchar*)&order;
1106    for(i = 0; i < sizeof(long); i++)
1107            t[o[i]] = f[i];
1108    return f+sizeof(long);
1109  }
1110
1111  static uchar*
1112  long2le(uchar *t, long from)
1113  {
1114    uchar *f, *o;
1115    int i;
1116
1117    f = (uchar*)&from;
1118    o = (uchar*)&order;
1119    for(i = 0; i < sizeof(long); i++)
1120            t[i] = f[o[i]];
1121    return t+sizeof(long);
1122  }
1123
1124  char *Ebadtimectl = "bad time control";
1125
1126  /*
1127   *  like the old #c/time but with added info.  Return
1128   *
1129   *        secs    nanosecs        fastticks       fasthz
1130   */
1131  static int
1132  readtime(ulong off, char *buf, int n)
1133  {
1134    vlong   nsec, ticks;
1135    long sec;
1136    char str[7*NUMSIZE+4];  // extra 4 bytes are null plus doprint
1137                            // reserving space for a frigging UTF
1138                            // char
1139
1140    nsec = todget(&ticks);
1141    if(fasthz == 0LL)
1142            fastticks((uvlong*)&fasthz);
1143    sec = nsec/1000000000ULL;
1144    snprint(str, sizeof(str), "%*.0lud %*.0llud %*.0llud %*.0llud ",
1145            NUMSIZE-1, sec,
1146            VLNUMSIZE-1, nsec,
1147            VLNUMSIZE-1, ticks,
1148            VLNUMSIZE-1, fasthz);
1149    return readstr(off, buf, n, str);
1150  }
1151
1152  /*
1153   *  set the time in seconds
1154   */
1155  static int
1156  writetime(char *buf, int n)
1157  {
1158    char b[13];
1159    long i;
1160    vlong now;
1161
1162    if(n >= sizeof(b))
1163            error(Ebadtimectl);
1164    strncpy(b, buf, n);
1165    b[n] = 0;
1166    i = strtol(b, 0, 0);
1167    if(i <= 0)
1168            error(Ebadtimectl);
1169    now = i*1000000000LL;
1170    todset(now, 0, 0);
1171    return n;
1172  }
1173
1174  /*
1175   *  read binary time info.  all numbers are little endian.
1176   *  ticks and nsec are syncronized.
1177   */
1178  static int
1179  readbintime(char *buf, int n)
1180  {
1181    int i;
1182    vlong nsec, ticks;
1183    uchar *b = (uchar*)buf;
1184
1185    i = 0;
1186    if(fasthz == 0LL)
1187            fastticks((uvlong*)&fasthz);
1188    nsec = todget(&ticks);
1189    if(n >= 3*sizeof(uvlong)){
1190            vlong2le(b+2*sizeof(uvlong), fasthz);
1191            i += sizeof(uvlong);
1192    }
1193    if(n >= 2*sizeof(uvlong)){
1194            vlong2le(b+sizeof(uvlong), ticks);
1195            i += sizeof(uvlong);
1196    }
1197    if(n >= 8){
1198            vlong2le(b, nsec);
1199            i += sizeof(vlong);
1200    }
1201    return i;
1202  }
1203
1204  /*
1205   *  set any of the following
1206   *        - time in nsec
1207   *        - nsec trim applied over some seconds
1208   *        - clock frequency
1209   */
1210  static int
1211  writebintime(char *buf, int n)
1212  {
1213    uchar *p;
1214    vlong delta;
1215    long period;
1216
1217    n--;
1218    p = (uchar*)buf + 1;
1219    switch(*buf){
1220    case 'n':
1221            if(n < sizeof(vlong))
1222                    error(Ebadtimectl);
1223            le2vlong(&delta, p);
1224            todset(delta, 0, 0);
```

```
1225            break;
1226    case 'd':
1227            if(n < sizeof(vlong)+sizeof(long))
1228                    error(Ebadtimectl);
1229            p = le2vlong(&delta, p);
1230            le2long(&period, p);
1231            todset(-1, delta, period);
1232            break;
1233    case 'f':
1234            if(n < sizeof(uvlong))
1235                    error(Ebadtimectl);
1236            le2vlong(&fasthz, p);
1237            todsetfreq(fasthz);
1238            break;
1239    }
1240    return n;
1241  }
```

```
#include  "u.h"
#include  "../port/lib.h"
#include  "mem.h"
#include  "dat.h"
#include  "fns.h"
#include  "../port/error.h"


#define   LRES    3               /* log of PC resolution */
#define   SZ      4               /* sizeof of count cell; well known as 4 */

struct
{
  int     minpc;
  int     maxpc;
  int     nbuf;
  int     time;
  ulong   *buf;
}kprof;

enum{
  Kprofdirqid,
  Kprofdataqid,
  Kprofctlqid,
};
Dirtab kproftab[]={
  "kpdata",       {Kprofdataqid},         0,      0600,
  "kpctl",        {Kprofctlqid},          0,      0600,
};

static void
_kproftimer(ulong pc)
{
  extern void spldone(void);

  if(kprof.time == 0)
          return;
  /*
   *  if the pc is coming out of spllo or splx,
   *  use the pc saved when we went splhi.
   */
  if(pc>=(ulong)spllo && pc<=(ulong)spldone)
          pc = m->splpc;

  kprof.buf[0] += TK2MS(1);
  if(kprof.minpc<=pc && pc<kprof.maxpc){
          pc -= kprof.minpc;
          pc >>= LRES;
          kprof.buf[pc] += TK2MS(1);
  }else
          kprof.buf[1] += TK2MS(1);
}

static void
kprofinit(void)
{
  if(SZ != sizeof kprof.buf[0])
          panic("kprof size");
  kproftimer = _kproftimer;
}

static Chan*
kprofattach(char *spec)
{
  ulong n;

  /* allocate when first used */
  kprof.minpc = KTZERO;
  kprof.maxpc = (ulong)etext;
  kprof.nbuf = (kprof.maxpc-kprof.minpc) >> LRES;
  n = kprof.nbuf*SZ;
  if(kprof.buf == 0) {
          kprof.buf = xalloc(n);
          if(kprof.buf == 0)
                  error(Enomem);
  }
  kproftab[0].length = n;
  return devattach('T', spec);
}

static int
kprofwalk(Chan *c, char *name)
{
  return devwalk(c, name, kproftab, nelem(kproftab), devgen);
}

static void
kprofstat(Chan *c, char *db)
{
  devstat(c, db, kproftab, nelem(kproftab), devgen);
}

static Chan*
kprofopen(Chan *c, int omode)
{
  if(c->qid.path == CHDIR){
          if(omode != OREAD)
                  error(Eperm);
  }
  c->mode = openmode(omode);
  c->flag |= COPEN;
  c->offset = 0;
  return c;
}

static void
kprofclose(Chan*)
{
}

static long
kprofread(Chan *c, void *va, long n, vlong off)
{
  ulong end;
  ulong w, *bp;
  uchar *a, *ea;
  ulong offset = off;

  switch(c->qid.path & ~CHDIR){
  case Kprofdirqid:
          return devdirread(c, va, n, kproftab, nelem(kproftab), devgen);

  case Kprofdataqid:
          end = kprof.nbuf*SZ;
          if(offset & (SZ-1))
                  error(Ebadarg);
          if(offset >= end){
                  n = 0;
                  break;
          }
          if(offset+n > end)
                  n = end-offset;
          n &= ~(SZ-1);
          a = va;
          ea = a + n;
          bp = kprof.buf + offset/SZ;
```

```
137             while(a < ea){
138                     w = *bp++;
139                     *a++ = w>>24;
140                     *a++ = w>>16;
141                     *a++ = w>>8;
142                     *a++ = w>>0;
143             }
144             break;
145
146     default:
147             n = 0;
148             break;
149     }
150     return n;
151   }
152
153   static long
154   kprofwrite(Chan *c, void *a, long n, vlong)
155   {
156     switch((int)(c->qid.path&~CHDIR)){
157     case Kprofctlqid:
158             if(strncmp(a, "startclr", 8) == 0){
159                     memset((char *)kprof.buf, 0, kprof.nbuf*SZ);
160                     kprof.time = 1;
161             }else if(strncmp(a, "start", 5) == 0)
162                     kprof.time = 1;
163             else if(strncmp(a, "stop", 4) == 0)
164                     kprof.time = 0;
165             break;
166     default:
167             error(Ebadusefd);
168     }
169     return n;
170   }
171
172   Dev kprofdevtab = {
173     'T',
174     "kprof",
175
176     devreset,
177     kprofinit,
178     kprofattach,
179     devclone,
180     kprofwalk,
181     kprofstat,
182     kprofopen,
183     devcreate,
184     kprofclose,
185     kprofread,
186     devbread,
187     kprofwrite,
188     devbwrite,
189     devremove,
190     devwstat,
191   };
```

```
#include  "u.h"
#include  "../port/lib.h"
#include  "mem.h"
#include  "dat.h"
#include  "fns.h"
#include  "../port/error.h"

struct Mntrpc
{
  Chan*   c;              /* Channel for whom we are working */
  Mntrpc* list;           /* Free/pending list */
  Fcall   request;        /* Outgoing file system protocol message */
  Fcall   reply;          /* Incoming reply */
  Mnt*    m;              /* Mount device during rpc */
  Rendez  r;              /* Place to hang out */
  char*   rpc;            /* I/O Data buffer */
  char    done;           /* Rpc completed */
  uvlong  stime;          /* start time for mnt statistics */
  ulong   reqlen;         /* request length for mnt statistics */
  ulong   replen;         /* reply length for mnt statistics */
  Mntrpc* flushed;        /* message this one flushes */
};

struct Mntalloc
{
  Lock;
  Mnt*    list;           /* Mount devices in use */
  Mnt*    mntfree;        /* Free list */
  Mntrpc* rpcfree;
  int     nrpcfree;
  int     nrpcused;
  ulong   id;
  int     rpctag;
}mntalloc;

void      mattach(Mnt*, Chan*, char*);
void      mntauth(Mnt*, Mntrpc*, char*, ushort);
Mnt*      mntchk(Chan*);
void      mntdirfix(uchar*, Chan*);
Mntrpc*   mntflushalloc(Mntrpc*);
void      mntflushfree(Mnt*, Mntrpc*);
void      mntfree(Mntrpc*);
void      mntgate(Mnt*);
void      mntpntfree(Mnt*);
void      mntqrm(Mnt*, Mntrpc*);
Mntrpc*   mntralloc(Chan*);
long      mntrdwr(int, Chan*, void*, long, vlong);
long      mnt9prdwr(int, Chan*, void*, long, vlong);
void      mntrpcread(Mnt*, Mntrpc*);
void      mountio(Mnt*, Mntrpc*);
void      mountmux(Mnt*, Mntrpc*);
void      mountrpc(Mnt*, Mntrpc*);
int       rpcattn(void*);
void      mclose(Mnt*, Chan*);
Chan*     mntchan(void);

void (*mntstats)(int, Chan*, uvlong, ulong);

enum
{
  Tagspace        = 1,
};

static void
mntreset(void)
{
  mntalloc.id = 1;
  mntalloc.rpctag = Tagspace;

  cinit();
}

static Chan*
mntattach(char *muxattach)
{
  Mnt *m;
  Chan *c, *mc;
  char buf[NAMELEN];
  struct bogus{
          Chan    *chan;
          char    *spec;
          int     flags;
  }bogus;

  bogus = *((struct bogus *)muxattach);
  c = bogus.chan;

  lock(&mntalloc);
  for(m = mntalloc.list; m; m = m->list) {
          if(m->c == c && m->id) {
                  lock(m);
                  if(m->id && m->ref > 0 && m->c == c) {
                          m->ref++;
                          unlock(m);
                          unlock(&mntalloc);
                          c = mntchan();
                          if(waserror()) {
                                  chanfree(c);
                                  nexterror();
                          }
                          mattach(m, c, bogus.spec);
                          poperror();
                          if(bogus.flags&MCACHE)
                                  c->flag |= CCACHE;
                          return c;
                  }
                  unlock(m);
          }
  }

  m = mntalloc.mntfree;
  if(m != 0)
          mntalloc.mntfree = m->list;
  else {
          m = malloc(sizeof(Mnt));
          if(m == 0) {
                  unlock(&mntalloc);
                  exhausted("mount devices");
          }
  }
  m->list = mntalloc.list;
  mntalloc.list = m;
  m->id = mntalloc.id++;
  unlock(&mntalloc);

  lock(m);
  m->ref = 1;
  m->queue = 0;
  m->rip = 0;
  m->c = c;
  m->c->flag |= CMSG;
  if(strncmp(bogus.spec, "mntblk=", 7) == 0) {
          m->blocksize = strtoul(bogus.spec+7, 0, 0);
          if(m->blocksize > MAXFDATA)
                  m->blocksize = MAXFDATA;
          print("mount blk %d\n", m->blocksize);
```

```
137             bogus.spec = "";
138     }
139     else
140             m->blocksize = MAXFDATA;
141     m->flags = bogus.flags & ~MCACHE;
142
143     incref(m->c);
144
145     sprint(buf, "#M%lud", m->id);
146
147     unlock(m);
148
149     c = mntchan();
150     if(waserror()) {
151             mclose(m, c);
152             /* Close must not be called since it will
153              * call mnt recursively
154              */
155             chanfree(c);
156             nexterror();
157     }
158
159     mattach(m, c, bogus.spec);
160     poperror();
161
162     /*
163      * Detect a recursive mount for a mount point served by exportfs.
164      * If CHDIR is clear in the returned qid, the foreign server is
165      * requesting the mount point be folded into the connection
166      * to the exportfs. In this case the remote mount driver does
167      * the multiplexing.
168      */
169     mc = m->c;
170     if(mc->type == devno('M', 0) && (c->qid.path&CHDIR) == 0) {
171             mclose(m, c);
172             c->qid.path |= CHDIR;
173             c->mntptr = mc->mntptr;
174             c->mchan = c->mntptr->c;
175             c->mqid = c->qid;
176             incref(c->mntptr);
177     }
178
179     if(bogus.flags & MCACHE)
180             c->flag |= CCACHE;
181
182     return c;
183   }
184
185   Chan*
186   mntchan(void)
187   {
188     Chan *c;
189
190     c = devattach('M', 0);
191     lock(&mntalloc);
192     c->dev = mntalloc.id++;
193     unlock(&mntalloc);
194
195     return c;
196   }
197
198   void
199   mattach(Mnt *m, Chan *c, char *spec)
200   {
201     ulong id;
202     Mntrpc *r;
203
204     r = mntralloc(0);
205     c->mntptr = m;
206
207     if(waserror()) {
208             mntfree(r);
209             nexterror();
210     }
211
212     r->request.type = Tattach;
213     r->request.fid = c->fid;
214     memmove(r->request.uname, up->user, NAMELEN);
215     strncpy(r->request.aname, spec, NAMELEN);
216     id = authrequest(m->c->session, &r->request);
217     mountrpc(m, r);
218     authreply(m->c->session, id, &r->reply);
219
220     c->qid = r->reply.qid;
221     c->mchan = m->c;
222     c->mqid = c->qid;
223
224     poperror();
225     mntfree(r);
226   }
227
228   static Chan*
229   mntclone(Chan *c, Chan *nc)
230   {
231     Mnt *m;
232     Mntrpc *r;
233     int alloc = 0;
234
235     m = mntchk(c);
236     r = mntralloc(c);
237     if(nc == 0) {
238             nc = newchan();
239             alloc = 1;
240     }
241     if(waserror()) {
242             mntfree(r);
243             if(alloc)
244                     cclose(nc);
245             nexterror();
246     }
247
248     r->request.type = Tclone;
249     r->request.fid = c->fid;
250     r->request.newfid = nc->fid;
251     mountrpc(m, r);
252
253     devclone(c, nc);
254     nc->mqid = c->qid;
255     incref(m);
256
257     USED(alloc);
258     poperror();
259     mntfree(r);
260     return nc;
261   }
262
263   static int
264   mntwalk(Chan *c, char *name)
265   {
266     Mnt *m;
267     Mntrpc *r;
268
269     m = mntchk(c);
270     r = mntralloc(c);
271     if(waserror()) {
272             mntfree(r);
```

```
273             return 0;
274     }
275     r->request.type = Twalk;
276     r->request.fid = c->fid;
277     strncpy(r->request.name, name, NAMELEN);
278     mountrpc(m, r);
279
280     c->qid = r->reply.qid;
281
282     poperror();
283     mntfree(r);
284     return 1;
285   }
286
287   static void
288   mntstat(Chan *c, char *dp)
289   {
290     Mnt *m;
291     Mntrpc *r;
292
293     m = mntchk(c);
294     r = mntralloc(c);
295     if(waserror()) {
296             mntfree(r);
297             nexterror();
298     }
299     r->request.type = Tstat;
300     r->request.fid = c->fid;
301     mountrpc(m, r);
302
303     memmove(dp, r->reply.stat, DIRLEN);
304     mntdirfix((uchar*)dp, c);
305     poperror();
306     mntfree(r);
307   }
308
309   static Chan*
310   mntopen(Chan *c, int omode)
311   {
312     Mnt *m;
313     Mntrpc *r;
314
315     m = mntchk(c);
316     r = mntralloc(c);
317     if(waserror()) {
318             mntfree(r);
319             nexterror();
320     }
321     r->request.type = Topen;
322     r->request.fid = c->fid;
323     r->request.mode = omode;
324     mountrpc(m, r);
325
326     c->qid = r->reply.qid;
327     c->offset = 0;
328     c->mode = openmode(omode);
329     c->flag |= COPEN;
330     poperror();
331     mntfree(r);
332
333     if(c->flag & CCACHE)
334             copen(c);
335
336     return c;
337   }
338
339   static void
340   mntcreate(Chan *c, char *name, int omode, ulong perm)
341   {
342     Mnt *m;
343     Mntrpc *r;
344
345     m = mntchk(c);
346     r = mntralloc(c);
347     if(waserror()) {
348             mntfree(r);
349             nexterror();
350     }
351     r->request.type = Tcreate;
352     r->request.fid = c->fid;
353     r->request.mode = omode;
354     r->request.perm = perm;
355     strncpy(r->request.name, name, NAMELEN);
356     mountrpc(m, r);
357
358     c->qid = r->reply.qid;
359     c->flag |= COPEN;
360     c->mode = openmode(omode);
361     poperror();
362     mntfree(r);
363
364     if(c->flag & CCACHE)
365             copen(c);
366   }
367
368   static void
369   mntclunk(Chan *c, int t)
370   {
371     Mnt *m;
372     Mntrpc *r;
373
374     m = mntchk(c);
375     r = mntralloc(c);
376     if(waserror()){
377             mntfree(r);
378             mclose(m, c);
379             nexterror();
380     }
381
382     r->request.type = t;
383     r->request.fid = c->fid;
384     mountrpc(m, r);
385     mntfree(r);
386     mclose(m, c);
387     poperror();
388   }
389
390   void
391   mclose(Mnt *m, Chan*)
392   {
393     Mntrpc *q, *r;
394
395     if(decref(m) != 0)
396             return;
397
398     for(q = m->queue; q; q = r) {
399             r = q->list;
400             mntfree(q);
401     }
402     m->id = 0;
403     cclose(m->c);
404     mntpntfree(m);
405   }
406
407   void
408   mntpntfree(Mnt *m)
```

```
{
  Mnt *f, **l;

  lock(&mntalloc);
  l = &mntalloc.list;
  for(f = *l; f; f = f->list) {
          if(f == m) {
                  *l = m->list;
                  break;
          }
          l = &f->list;
  }

  m->list = mntalloc.mntfree;
  mntalloc.mntfree = m;
  unlock(&mntalloc);
}

static void
mntclose(Chan *c)
{
  mntclunk(c, Tclunk);
}

static void
mntremove(Chan *c)
{
  mntclunk(c, Tremove);
}

static void
mntwstat(Chan *c, char *dp)
{
  Mnt *m;
  Mntrpc *r;

  m = mntchk(c);
  r = mntralloc(c);
  if(waserror()) {
          mntfree(r);
          nexterror();
  }
  r->request.type = Twstat;
  r->request.fid = c->fid;
  memmove(r->request.stat, dp, DIRLEN);
  mountrpc(m, r);
  poperror();
  mntfree(r);
}

long
mntread9p(Chan *c, void *buf, long n, vlong off)
{
  return mnt9prdwr(Tread, c, buf, n, off);
}

static long
mntread(Chan *c, void *buf, long n, vlong off)
{
  uchar *p, *e;
  int nc, cache, isdir;

  isdir = 0;
  cache = c->flag & CCACHE;
  if(c->qid.path & CHDIR) {
          cache = 0;
          isdir = 1;
  }

  p = buf;
  if(cache) {
          nc = cread(c, buf, n, off);
          if(nc > 0) {
                  n -= nc;
                  if(n == 0)
                          return nc;
                  p += nc;
                  off += nc;
          }
          n = mntrdwr(Tread, c, p, n, off);
          cupdate(c, p, n, off);
          return n + nc;
  }

  n = mntrdwr(Tread, c, buf, n, off);
  if(isdir) {
          for(e = &p[n]; p < e; p += DIRLEN)
                  mntdirfix(p, c);
  }

  return n;
}

long
mntwrite9p(Chan *c, void *buf, long n, vlong off)
{
  return mnt9prdwr(Twrite, c, buf, n, off);
}

static long
mntwrite(Chan *c, void *buf, long n, vlong off)
{
  return mntrdwr(Twrite, c, buf, n, off);
}

long
mnt9prdwr(int type, Chan *c, void *buf, long n, vlong off)
{
  Mnt *m;
  ulong nr;
  Mntrpc *r;

  if(n > MAXRPC-32) {
          if(type == Twrite)
                  error("write9p too long");
          n = MAXRPC-32;
  }

  m = mntchk(c);
  r = mntralloc(c);
  if(waserror()) {
          mntfree(r);
          nexterror();
  }
  r->request.type = type;
  r->request.fid = c->fid;
  r->request.offset = off;
  r->request.data = buf;
  r->request.count = n;
  mountrpc(m, r);
  nr = r->reply.count;
  if(nr > r->request.count)
          nr = r->request.count;

  if(type == Tread)
          memmove(buf, r->reply.data, nr);
```

```
545
546     poperror();
547     mntfree(r);
548     return nr;
549   }
550
551   long
552   mntrdwr(int type, Chan *c, void *buf, long n, vlong off)
553   {
554     Mnt *m;
555     Mntrpc *r;
556     char *uba;
557     int cache;
558     ulong cnt, nr, nreq;
559
560     m = mntchk(c);
561     uba = buf;
562     cnt = 0;
563     cache = c->flag & CCACHE;
564     if(c->qid.path & CHDIR)
565             cache = 0;
566     for(;;) {
567             r = mntralloc(c);
568             if(waserror()) {
569                     mntfree(r);
570                     nexterror();
571             }
572             r->request.type = type;
573             r->request.fid = c->fid;
574             r->request.offset = off;
575             r->request.data = uba;
576             if(n > m->blocksize){
577                     if(c->qid.path & CHDIR)
578                             r->request.count = (m->blocksize/DIRLEN)*DIRLEN;
579                     else
580                             r->request.count = m->blocksize;
581             } else
582                     r->request.count = n;
583             mountrpc(m, r);
584             nreq = r->request.count;
585             nr = r->reply.count;
586             if(nr > nreq)
587                     nr = nreq;
588
589             if(type == Tread)
590                     memmove(uba, r->reply.data, nr);
591             else if(cache)
592                     cwrite(c, (uchar*)uba, nr, off);
593
594             poperror();
595             mntfree(r);
596             off += nr;
597             uba += nr;
598             cnt += nr;
599             n -= nr;
600             if(nr != nreq || n == 0 || up->nnote)
601                     break;
602     }
603     return cnt;
604   }
605
606   void
607   mountrpc(Mnt *m, Mntrpc *r)
608   {
609     int t;
610
611     r->reply.tag = 0;
612     r->reply.type = Tmax;   /* can't ever be a valid message type */
613
614     mountio(m, r);
615
616     t = r->reply.type;
617     switch(t) {
618     case Rerror:
619             error(r->reply.ename);
620     case Rflush:
621             error(Eintr);
622     default:
623             if(t == r->request.type+1)
624                     break;
625             print("mnt: proc %s %lud: mismatch rep 0x%lux tag %d fid %d T%d R%d rp %d\n",
626                     up->text, up->pid,
627                     r, r->request.tag, r->request.fid, r->request.type, r->reply.type,
628                     r->reply.tag);
629             error(Emountrpc);
630     }
631   }
632
633   void
634   mountio(Mnt *m, Mntrpc *r)
635   {
636     int n;
637
638     while(waserror()) {
639             if(m->rip == up)
640                     mntgate(m);
641             if(strcmp(up->error, Eintr) != 0){
642                     mntflushfree(m, r);
643                     nexterror();
644             }
645             r = mntflushalloc(r);
646     }
647
648     lock(m);
649     r->m = m;
650     r->list = m->queue;
651     m->queue = r;
652     unlock(m);
653
654     /* Transmit a file system rpc */
655     n = convS2M(&r->request, r->rpc);
656     if(n < 0)
657             panic("bad message type in mountio");
658     if(devtab[m->c->type]->dc == L'M'){
659             if(mnt9prdwr(Twrite, m->c, r->rpc, n, 0) != n)
660                     error(Emountrpc);
661     }else{
662             if(devtab[m->c->type]->write(m->c, r->rpc, n, 0) != n)
663                     error(Emountrpc);
664     }
665     r->stime = fastticks(nil);
666     r->reqlen = n;
667
668     /* Gate readers onto the mount point one at a time */
669     for(;;) {
670             lock(m);
671             if(m->rip == 0)
672                     break;
673             unlock(m);
674             sleep(&r->r, rpcattn, r);
675             if(r->done){
676                     poperror();
677                     mntflushfree(m, r);
678                     return;
679             }
680     }
```

```
681     m->rip = up;
682     unlock(m);
683     while(r->done == 0) {
684             mntrpcread(m, r);
685             mountmux(m, r);
686     }
687     mntgate(m);
688     poperror();
689     mntflushfree(m, r);
690   }
691
692   void
693   mntrpcread(Mnt *m, Mntrpc *r)
694   {
695     int n;
696
697     for(;;) {
698             r->reply.type = 0;
699             r->reply.tag = 0;
700             if(devtab[m->c->type]->dc == L'M')
701                     n = mnt9prdwr(Tread, m->c, r->rpc, MAXRPC, 0);
702             else
703                     n = devtab[m->c->type]->read(m->c, r->rpc, MAXRPC, 0);
704             if(n == 0)
705                     continue;
706
707             r->replen = n;
708             if(convM2S(r->rpc, &r->reply, n) != 0)
709                     return;
710     }
711   }
712
713   void
714   mntgate(Mnt *m)
715   {
716     Mntrpc *q;
717
718     lock(m);
719     m->rip = 0;
720     for(q = m->queue; q; q = q->list) {
721             if(q->done == 0)
722             if(wakeup(&q->r))
723                     break;
724     }
725     unlock(m);
726   }
727
728   void
729   mountmux(Mnt *m, Mntrpc *r)
730   {
731     char *dp;
732     Mntrpc **l, *q;
733
734     lock(m);
735     l = &m->queue;
736     for(q = *l; q; q = q->list) {
737             /* look for a reply to a message */
738             if(q->request.tag == r->reply.tag) {
739                     *l = q->list;
740                     if(q != r) {
741                             /*
742                              * Completed someone else.
743                              * Trade pointers to receive buffer.
744                              */
745                             dp = q->rpc;
746                             q->rpc = r->rpc;
747                             r->rpc = dp;
748                             q->reply = r->reply;
749                     }
750                     q->done = 1;
751                     unlock(m);
752                     if(mntstats != nil)
753                             (*mntstats)(q->request.type,
754                                     m->c, q->stime,
755                                     q->reqlen + r->replen);
756                     if(q != r)
757                             wakeup(&q->r);
758                     return;
759             }
760             l = &q->list;
761     }
762     unlock(m);
763   }
764
765   /*
766    * Create a new flush request and chain the previous
767    * requests from it
768    */
769   Mntrpc*
770   mntflushalloc(Mntrpc *r)
771   {
772     Mntrpc *fr;
773
774     fr = mntralloc(0);
775
776     fr->request.type = Tflush;
777     if(r->request.type == Tflush)
778             fr->request.oldtag = r->request.oldtag;
779     else
780             fr->request.oldtag = r->request.tag;
781     fr->flushed = r;
782
783     return fr;
784   }
785
786   /*
787    *  Free a chain of flushes.  Remove each unanswered
788    *  flush and the original message from the unanswered
789    *  request queue.  Mark the original message as done
790    *  and if it hasn't been answered set the reply to to
791    *  Rflush.
792    */
793   void
794   mntflushfree(Mnt *m, Mntrpc *r)
795   {
796     Mntrpc *fr;
797
798     while(r){
799             fr = r->flushed;
800             if(!r->done){
801                     r->reply.type = Rflush;
802                     mntqrm(m, r);
803             }
804             if(fr)
805                     mntfree(r);
806             r = fr;
807     }
808   }
809
810   Mntrpc*
811   mntralloc(Chan *c)
812   {
813     Mntrpc *new;
814
815     lock(&mntalloc);
816     new = mntalloc.rpcfree;
```

```
817     if(new == nil){
818             new = malloc(sizeof(Mntrpc));
819             if(new == nil) {
820                     unlock(&mntalloc);
821                     exhausted("mount rpc header");
822             }
823             /*
824              * The header is split from the data buffer as
825              * mountmux may swap the buffer with another header.
826              */
827             new->rpc = mallocz(MAXRPC, 0);
828             if(new->rpc == nil){
829                     free(new);
830                     unlock(&mntalloc);
831                     exhausted("mount rpc buffer");
832             }
833             new->request.tag = mntalloc.rpctag++;
834     }
835     else {
836             mntalloc.rpcfree = new->list;
837             mntalloc.nrpcfree--;
838     }
839     mntalloc.nrpcused++;
840     unlock(&mntalloc);
841     new->c = c;
842     new->done = 0;
843     new->flushed = nil;
844     return new;
845   }
846
847   void
848   mntfree(Mntrpc *r)
849   {
850     lock(&mntalloc);
851     if(mntalloc.nrpcfree >= 10){
852             free(r->rpc);
853             free(r);
854     }
855     else{
856             r->list = mntalloc.rpcfree;
857             mntalloc.rpcfree = r;
858             mntalloc.nrpcfree++;
859     }
860     mntalloc.nrpcused--;
861     unlock(&mntalloc);
862   }
863
864   void
865   mntqrm(Mnt *m, Mntrpc *r)
866   {
867     Mntrpc **l, *f;
868
869     lock(m);
870     r->done = 1;
871
872     l = &m->queue;
873     for(f = *l; f; f = f->list) {
874             if(f == r) {
875                     *l = r->list;
876                     break;
877             }
878             l = &f->list;
879     }
880     unlock(m);
881   }
882
883   Mnt*
884   mntchk(Chan *c)
885   {
886     Mnt *m;
887
888     m = c->mntptr;
889
890     /*
891      * Was it closed and reused
892      */
893     if(m->id == 0 || m->id >= c->dev)
894             error(Eshutdown);
895
896     return m;
897   }
898
899   void
900   mntdirfix(uchar *dirbuf, Chan *c)
901   {
902     int r;
903
904     r = devtab[c->type]->dc;
905     dirbuf[DIRLEN-4] = r>>0;
906     dirbuf[DIRLEN-3] = r>>8;
907     dirbuf[DIRLEN-2] = c->dev;
908     dirbuf[DIRLEN-1] = c->dev>>8;
909   }
910
911   int
912   rpcattn(void *v)
913   {
914     Mntrpc *r;
915
916     r = v;
917     return r->done || r->m->rip == 0;
918   }
919
920   Dev mntdevtab = {
921     'M',
922     "mnt",
923
924     mntreset,
925     devinit,
926     mntattach,
927     mntclone,
928     mntwalk,
929     mntstat,
930     mntopen,
931     mntcreate,
932     mntclose,
933     mntread,
934     devbread,
935     mntwrite,
936     devbwrite,
937     mntremove,
938     mntwstat,
939   };
```

```
#include  "u.h"
#include  "../port/lib.h"
#include  "mem.h"
#include  "dat.h"
#include  "fns.h"
#include  "../port/error.h"

#define   Image   IMAGE
#include  <draw.h>
#include  <memdraw.h>
#include  <cursor.h>
#include  "screen.h"

typedef struct Mouseinfo  Mouseinfo;
typedef struct Mousestate Mousestate;

struct Mousestate
{
  Point   xy;                     /* mouse.xy */
  int     buttons;                /* mouse.buttons */
  ulong   counter;        /* increments every update */
  ulong   msec;   /* time of last event */
};

struct Mouseinfo
{
  Mousestate;
  int     dx;
  int     dy;
  int     track;          /* dx & dy updated */
  int     redraw;         /* update cursor on screen */
  ulong   lastcounter;    /* value when /dev/mouse read */
  Rendez  r;
  Ref;
  QLock;
  int     open;
  int     inopen;
  int     acceleration;
  int     maxacc;
  Mousestate      queue[16];      /* circular buffer of click events */
  int     ri;     /* read index into queue */
  int     wi;     /* write index into queue */
  uchar   qfull;  /* queue is full */
};

Mouseinfo mouse;
Cursorinfo        cursor;
int               mouseshifted;
Cursor            curs;

void      Cursortocursor(Cursor*);
int       mousechanged(void*);
static void mouseclock(void);

enum{
  Qdir,
  Qcursor,
  Qmouse,
  Qmousein,
  Qmousectl,
};

static Dirtab mousedir[]={
  "cursor",       {Qcursor},      0,                      0666,
  "mouse",        {Qmouse},       0,                      0666,
  "mousein",      {Qmousein},     0,                      0220,
  "mousectl",     {Qmousectl},    0,                      0220,
};

static uchar buttonmap[8] = {
  0, 1, 2, 3, 4, 5, 6, 7,
};
static int mouseswap;

extern    Memimage*       gscreen;

static void
mousereset(void)
{
  if(!conf.monitor)
          return;

  curs = arrow;
  Cursortocursor(&arrow);
  addclock0link(mouseclock);
}

static void
mouseinit(void)
{
  if(!conf.monitor)
          return;

  cursoron(1);
}

static Chan*
mouseattach(char *spec)
{
  if(!conf.monitor)
          error(Egreg);
  return devattach('m', spec);
}

static Chan*
mouseclone(Chan *c, Chan *nc)
{
  nc = devclone(c, nc);
  if(c->qid.path != CHDIR)
          incref(&mouse);
  return nc;
}

static int
mousewalk(Chan *c, char *name)
{
  return devwalk(c, name, mousedir, nelem(mousedir), devgen);
}

static void
mousestat(Chan *c, char *db)
{
  devstat(c, db, mousedir, nelem(mousedir), devgen);
}

static Chan*
mouseopen(Chan *c, int omode)
{
  switch(c->qid.path){
  case CHDIR:
          if(omode != OREAD)
                  error(Eperm);
          break;
  case Qmouse:
          lock(&mouse);
          if(mouse.open){
```

```
137                     unlock(&mouse);
138                     error(Einuse);
139             }
140             mouse.open = 1;
141             mouse.ref++;
142             unlock(&mouse);
143             break;
144     case Qmousein:
145             error("disabled");
146             lock(&mouse);
147             if(mouse.inopen){
148                     unlock(&mouse);
149                     error(Einuse);
150             }
151             mouse.inopen = 1;
152             unlock(&mouse);
153             break;
154     default:
155             incref(&mouse);
156     }
157     c->mode = openmode(omode);
158     c->flag |= COPEN;
159     c->offset = 0;
160     return c;
161   }
162
163   static void
164   mousecreate(Chan*, char*, int, ulong)
165   {
166     if(!conf.monitor)
167             error(Egreg);
168     error(Eperm);
169   }
170
171   static void
172   mouseclose(Chan *c)
173   {
174     if(c->qid.path!=CHDIR && (c->flag&COPEN)){
175             lock(&mouse);
176             if(c->qid.path == Qmouse)
177                     mouse.open = 0;
178             else if(c->qid.path == Qmousein){
179                     mouse.inopen = 0;
180                     unlock(&mouse);
181                     return;
182             }
183             if(--mouse.ref == 0){
184                     cursoroff(1);
185                     curs = arrow;
186                     Cursortocursor(&arrow);
187                     cursoron(1);
188             }
189             unlock(&mouse);
190     }
191   }
192
193
194   static long
195   mouseread(Chan *c, void *va, long n, vlong off)
196   {
197     char buf[4*12+1];
198     uchar *p;
199     static int map[8] = {0, 4, 2, 6, 1, 5, 3, 7 };
200     ulong offset = off;
201     Mousestate m;
202
203     p = va;
204     switch(c->qid.path){
205     case CHDIR:
206             return devdirread(c, va, n, mousedir, nelem(mousedir), devgen);
207
208     case Qcursor:
209             if(offset != 0)
210                     return 0;
211             if(n < 2*4+2*2*16)
212                     error(Eshort);
213             n = 2*4+2*2*16;
214             lock(&cursor);
215             BPLONG(p+0, curs.offset.x);
216             BPLONG(p+4, curs.offset.y);
217             memmove(p+8, curs.clr, 2*16);
218             memmove(p+40, curs.set, 2*16);
219             unlock(&cursor);
220             return n;
221
222     case Qmouse:
223             while(mousechanged(0) == 0)
224                     sleep(&mouse.r, mousechanged, 0);
225
226             mouse.qfull = 0;
227
228             /*
229              * No lock of the indicies is necessary here, because ri is only
230              * updated by us, and there is only one mouse reader
231              * at a time.  I suppose that more than one process
232              * could try to read the fd at one time, but such behavior
233              * is degenerate and already violates the calling
234              * conventions for sleep above.
235              */
236             if(mouse.ri != mouse.wi) {
237                     m = mouse.queue[mouse.ri];
238                     if(++mouse.ri == nelem(mouse.queue))
239                             mouse.ri = 0;
240             } else {
241                     while(!canlock(&cursor))
242                             tsleep(&up->sleep, return0, 0, TK2MS(1));
243
244                     m = mouse.Mousestate;
245                     unlock(&cursor);
246             }
247
248             sprint(buf, "m%11d %11d %11d %11lud",
249                     m.xy.x, m.xy.y,
250                     buttonmap[m.buttons&7],
251                     m.msec);
252             mouse.lastcounter = m.counter;
253             if(n > 1+4*12)
254                     n = 1+4*12;
255             memmove(va, buf, n);
256             return n;
257     }
258     return 0;
259   }
260
261   static void
262   setbuttonmap(char* map)
263   {
264     int i, x, one, two, three;
265
266     one = two = three = 0;
267     for(i = 0; i < 3; i++){
268             if(map[i] == 0)
269                     error(Ebadarg);
270             if(map[i] == '1'){
271                     if(one)
272                             error(Ebadarg);
```

```
273                     one = 1<<i;
274             }
275             else if(map[i] == '2'){
276                     if(two)
277                             error(Ebadarg);
278                     two = 1<<i;
279             }
280             else if(map[i] == '3'){
281                     if(three)
282                             error(Ebadarg);
283                     three = 1<<i;
284             }
285             else
286                     error(Ebadarg);
287     }
288     if(map[i])
289             error(Ebadarg);
290
291     memset(buttonmap, 0, 8);
292     for(i = 0; i < 8; i++){
293             x = 0;
294             if(i & 1)
295                     x |= one;
296             if(i & 2)
297                     x |= two;
298             if(i & 4)
299                     x |= three;
300             buttonmap[x] = i;
301     }
302   }
303
304   static long
305   mousewrite(Chan *c, void *va, long n, vlong)
306   {
307     char *p;
308     Point pt;
309     char buf[64], *field[3];
310     int nf, b;
311
312     p = va;
313     switch(c->qid.path){
314     case CHDIR:
315             error(Eisdir);
316
317     case Qcursor:
318             cursoroff(1);
319             if(n < 2*4+2*2*16){
320                     curs = arrow;
321                     Cursortocursor(&arrow);
322             }else{
323                     n = 2*4+2*2*16;
324                     curs.offset.x = BGLONG(p+0);
325                     curs.offset.y = BGLONG(p+4);
326                     memmove(curs.clr, p+8, 2*16);
327                     memmove(curs.set, p+40, 2*16);
328                     Cursortocursor(&curs);
329             }
330             qlock(&mouse);
331             mouse.redraw = 1;
332             mouseclock();
333             qunlock(&mouse);
334             cursoron(1);
335             return n;
336
337     case Qmousectl:
338             if(n >= sizeof(buf))
339                     n = sizeof(buf)-1;
340             strncpy(buf, va, n);
341             if(buf[n - 1] == '\n')
342                     buf[n-1] = 0;
343             else
344                     buf[n] = 0;
345             nf = getfields(buf, field, 3, 1, " ");
346             if(strcmp(field[0], "swap") == 0){
347                     if(mouseswap)
348                             setbuttonmap("123");
349                     else
350                             setbuttonmap("321");
351                     mouseswap ^= 1;
352             }
353             else if(strcmp(field[0], "buttonmap") == 0){
354                     if(nf == 1)
355                             setbuttonmap("123");
356                     else
357                             setbuttonmap(field[1]);
358             }
359             else
360                     mousectl(field, nf);
361             return n;
362
363     case Qmousein:
364             if(n > sizeof buf-1)
365                     n = sizeof buf -1;
366             memmove(buf, va, n);
367             buf[n] = 0;
368             p = 0;
369             pt.x = strtol(buf+1, &p, 0);
370             if(p == 0)
371                     error(Eshort);
372             pt.y = strtol(p, &p, 0);
373             if(p == 0)
374                     error(Eshort);
375             b = strtol(p, &p, 0);
376             mousetrack(b, pt.x, pt.y);
377             return n;
378
379     case Qmouse:
380             if(n > sizeof buf-1)
381                     n = sizeof buf -1;
382             memmove(buf, va, n);
383             buf[n] = 0;
384             p = 0;
385             pt.x = strtoul(buf+1, &p, 0);
386             if(p == 0)
387                     error(Eshort);
388             pt.y = strtoul(p, 0, 0);
389             qlock(&mouse);
390             if(ptinrect(pt, gscreen->r)){
391                     mouse.xy = pt;
392                     mouse.redraw = 1;
393                     mouse.track = 1;
394                     mouseclock();
395             }
396             qunlock(&mouse);
397             return n;
398     }
399
400     error(Egreg);
401     return -1;
402   }
403
404   Dev mousedevtab = {
405     'm',
406     "mouse",
407
408     mousereset,
```

```
409     mouseinit,
410     mouseattach,
411     mouseclone,
412     mousewalk,
413     mousestat,
414     mouseopen,
415     mousecreate,
416     mouseclose,
417     mouseread,
418     devbread,
419     mousewrite,
420     devbwrite,
421     devremove,
422     devwstat,
423   };
424
425   void
426   Cursortocursor(Cursor *c)
427   {
428     lock(&cursor);
429     memmove(&cursor.Cursor, c, sizeof(Cursor));
430     setcursor(c);
431     unlock(&cursor);
432   }
433
434
435   /*
436    *  called by the clock routine to redraw the cursor
437    */
438   static void
439   mouseclock(void)
440   {
441     if(mouse.track){
442             mousetrack(mouse.buttons, mouse.dx, mouse.dy);
443             mouse.track = 0;
444             mouse.dx = 0;
445             mouse.dy = 0;
446     }
447     if(mouse.redraw && canlock(&cursor)){
448             mouse.redraw = 0;
449             cursoroff(0);
450             mouse.redraw = cursoron(0);
451             unlock(&cursor);
452     }
453     drawactive(0);
454   }
455
456   static int
457   scale(int x)
458   {
459     int sign = 1;
460
461     if(x < 0){
462             sign = -1;
463             x = -x;
464     }
465     switch(x){
466     case 0:
467     case 1:
468     case 2:
469     case 3:
470             break;
471     case 4:
472             x = 6 + (mouse.acceleration>>2);
473             break;
474     case 5:
475             x = 9 + (mouse.acceleration>>1);
476             break;
477     default:
478             x *= mouse.maxacc;
479             break;
480     }
481     return sign*x;
482   }
483
484   /*
485    *  called at interrupt level to update the structure and
486    *  awaken any waiting procs.
487    */
488   void
489   mousetrack(int b, int dx, int dy)
490   {
491     int x, y, lastb;
492
493     if(gscreen==nil)
494             return;
495
496     if(mouse.acceleration){
497             dx = scale(dx);
498             dy = scale(dy);
499     }
500     x = mouse.xy.x + dx;
501     if(x < gscreen->r.min.x)
502             x = gscreen->r.min.x;
503     if(x >= gscreen->r.max.x)
504             x = gscreen->r.max.x;
505     y = mouse.xy.y + dy;
506     if(y < gscreen->r.min.y)
507             y = gscreen->r.min.y;
508     if(y >= gscreen->r.max.y)
509             y = gscreen->r.max.y;
510
511     lastb = mouse.buttons;
512     mouse.xy = Pt(x, y);
513     mouse.buttons = b;
514     mouse.redraw = 1;
515     mouse.counter++;
516     mouse.msec = TK2MS(MACHP(0)->ticks);
517
518     /*
519      * if the queue fills, we discard the entire queue and don't
520      * queue any more events until a reader polls the mouse.
521      */
522     if(!mouse.qfull && lastb != b) {        /* add to ring */
523             mouse.queue[mouse.wi] = mouse.Mousestate;
524             if(++mouse.wi == nelem(mouse.queue))
525                     mouse.wi = 0;
526             if(mouse.wi == mouse.ri)
527                     mouse.qfull = 1;
528     }
529     wakeup(&mouse.r);
530     drawactive(1);
531   }
532
533   /*
534    *  microsoft 3 button, 7 bit bytes
535    *
536    *        byte 0 -        1  L  R Y7 Y6 X7 X6
537    *        byte 1 -        0 X5 X4 X3 X2 X1 X0
538    *        byte 2 -        0 Y5 Y4 Y3 Y2 Y1 Y0
539    *        byte 3 -        0  M  x  x  x  x  x     (optional)
540    *
541    *  shift & right button is the same as middle button (for 2 button mice)
542    */
543   int
544   m3mouseputc(Queue*, int c)
```

```
{
  static uchar msg[3];
  static int nb;
  static int middle;
  static uchar b[] = { 0, 4, 1, 5, 0, 2, 1, 5 };
  short x;
  int dx, dy, newbuttons;

  if(nb==0){
          /*
           * an extra byte comes for middle button motion.
           * only two possible values for the extra byte.
           */
          if(c == 0x00 || c == 0x20){
                  /* an extra byte gets sent for the middle button */
                  middle = (c&0x20) ? 2 : 0;
                  newbuttons = (mouse.buttons & ~2) | middle;
                  mousetrack(newbuttons, 0, 0);
                  return 0;
          }
  }
  msg[nb] = c;
  if(++nb == 3){
          nb = 0;
          newbuttons = middle | b[(msg[0]>>4)&3 | (mouseshifted ? 4 : 0)];
          x = (msg[0]&0x3)<<14;
          dx = (x>>8) | msg[1];
          x = (msg[0]&0xc)<<12;
          dy = (x>>8) | msg[2];
          mousetrack(newbuttons, dx, dy);
  }
  return 0;
}

/*
 *  Logitech 5 byte packed binary mouse format, 8 bit bytes
 *
 *  shift & right button is the same as middle button (for 2 button mice)
 */
int
mouseputc(Queue*, int c)
{
  static short msg[5];
  static int nb;
  static uchar b[] = {0, 4, 2, 6, 1, 5, 3, 7, 0, 2, 2, 6, 1, 5, 3, 7};
  int dx, dy, newbuttons;

  if((c&0xF0) == 0x80)
          nb=0;
  msg[nb] = c;
  if(c & 0x80)
          msg[nb] |= ~0xFF;       /* sign extend */
  if(++nb == 5){
          newbuttons = b[((msg[0]&7)^7) | (mouseshifted ? 8 : 0)];
          dx = msg[1]+msg[3];
          dy = -(msg[2]+msg[4]);
          mousetrack(newbuttons, dx, dy);
          nb = 0;
  }
  return 0;
}

int
mousechanged(void*)
{
  return mouse.lastcounter != mouse.counter;
}

Point
mousexy(void)
{
  return mouse.xy;
}

void
mouseaccelerate(int x)
{
  mouse.acceleration = x;
  if(mouse.acceleration < 3)
          mouse.maxacc = 2;
  else
          mouse.maxacc = mouse.acceleration;
}
```

```
#include  "u.h"
#include  "../port/lib.h"
#include  "mem.h"
#include  "dat.h"
#include  "fns.h"
#include  "io.h"
#include  "../port/error.h"

#include  "../port/netif.h"

/*
 *  Driver for the ns16552.
 */
enum
{
  /*
   *  register numbers
   */
  Data=   0,              /* xmit/rcv buffer */
  Iena=   1,              /* interrupt enable */
   Ircv=  (1<<0),         /*  for char rcv'd */
   Ixmt=  (1<<1),         /*  for xmit buffer empty */
   Irstat=(1<<2),         /*  for change in rcv'er status */
   Imstat=(1<<3),         /*  for change in modem status */
   Isleep=(1<<4),         /*  put in sleep mode; 16C650 */
   Ixoff= (1<<5),         /*  for xoff received; 16C650 */
   Irts=  (1<<6),         /*  for !rts asserted in auto mode; 16C650 */
   Icts=  (1<<7),         /*  for !cts asserted in auto mode; 16C650 */
  Istat=  2,              /* interrupt flag (read) */
   Fenabd=(3<<6),         /*  on if fifo's enabled */
   Irctsd=(1<<4),         /*  auto rts or cts changed state */
  Fifoctl=2,              /* fifo control (write) */
   Fena=  (1<<0),         /*  enable xmit/rcv fifos */
   Ftrig4=        (1<<6),         /*  trigger after 4 input characters */
   Ftrig24=       (2<<6),         /*  trigger after 24 input characters */
   Fclear=(3<<1),         /*  clear xmit & rcv fifos */
  Format= 3,              /* byte format */
   Bits8= (3<<0),         /*  8 bits/byte */
   Stop2= (1<<2),         /*  2 stop bits */
   Pena=  (1<<3),         /*  generate parity */
   Peven= (1<<4),         /*  even parity */
   Pforce=(1<<5),         /*  force parity */
   Break= (1<<6),         /*  generate a break */
   Dra=   (1<<7),         /*  address the divisor */
   Ers=   0xbf,           /*  enable enhaced register set on 16C650 */
  Mctl=   4,              /* modem control */
   Dtr=   (1<<0),         /*  data terminal ready */
   Rts=   (1<<1),         /*  request to send */
   Ri=    (1<<2),         /*  ring */
   Inton= (1<<3),         /*  turn on interrupts */
   Loop=  (1<<4),         /*  loop back */
   Intsel=(1<<5),         /*  NO! open source interrupts; 16C650 */
   Irda=  (1<<6),         /*  infrared interface; 16C650 */
   Cdiv=  (1<<7),         /*  divide clock by four; 16C650 */
  Lstat=  5,              /* line status */
   Inready=(1<<0),        /*  receive buffer full */
   Oerror=(1<<1),         /*  receiver overrun */
   Perror=(1<<2),         /*  receiver parity error */
   Ferror=(1<<3),         /*  rcv framing error */
   Outready=(1<<5),       /*  output buffer full */
  Mstat=  6,              /* modem status */
   Ctsc=  (1<<0),         /*  clear to send changed */
   Dsrc=  (1<<1),         /*  data set ready changed */
   Rire=  (1<<2),         /*  rising edge of ring indicator */
   Dcdc=  (1<<3),         /*  data carrier detect changed */
   Cts=   (1<<4),         /*  complement of clear to send line */
   Dsr=   (1<<5),         /*  complement of data set ready line */
   Ring=  (1<<6),         /*  complement of ring indicator line */
   Dcd=   (1<<7),         /*  complement of data carrier detect line */
  Scratch=7,              /* scratchpad */
  Dlsb=   0,              /* divisor lsb */
  Dmsb=   1,              /* divisor msb */
  Efr=    2,              /* enhanced features for 16C650 */
   Eena=  (1<<4),         /* enable enhanced bits in normal registers */
   Arts=  (1<<6),         /* auto rts */
   Acts=  (1<<7),         /* auto cts */

  CTLS= 023,
  CTLQ= 021,

  Stagesize= 1024,
  Nuart=  32,             /* max per machine */

  Ns450=  0,
  Ns550,
  Ns650,
};

typedef struct Uart Uart;
struct Uart
{
  QLock;
  int     opens;

  int     enabled;
  Uart    *elist;                 /* next enabled interface */
  char    name[NAMELEN];

  uchar   sticky[8];              /* sticky write register values */
  uchar   osticky[8];             /* kernel saved sticky write register values */
  ulong   port;                   /* io ports */
  ulong   freq;                   /* clock frequency */
  uchar   mask;                   /* bits/char */
  int     dev;
  int     baud;                   /* baud rate */

  uchar   istat;                  /* last istat read */
  int     frame;                  /* framing errors */
  int     overrun;                /* rcvr overruns */

  /* buffers */
  int     (*putc)(Queue*, int);
  Queue   *iq;
  Queue   *oq;

  Lock    flock;                  /* fifo */
  uchar   fifoon;                 /* fifo's enabled */
  uchar   type;                   /* chip version */

  Lock    rlock;                  /* receive */
  uchar   istage[Stagesize];
  uchar   *ip;
  uchar   *ie;

  int     haveinput;

  Lock    tlock;                  /* transmit */
  uchar   ostage[Stagesize];
  uchar   *op;
  uchar   *oe;

  int     modem;                  /* hardware flow control on */
  int     xonoff;                 /* software flow control on */
  int     blocked;
  int     cts, dsr, dcd, dcdts;           /* keep track of modem status */
  int     ctsbackoff;
```

```
137     int     hup_dsr, hup_dcd;       /* send hangup upstream? */
138     int     dohup;
139
140     int     kinuse;         /* device in use by kernel */
141
142     Rendez  r;
143   };
144
145   static Uart *uart[Nuart];
146   static int nuart;
147
148   struct Uartalloc {
149     Lock;
150     Uart *elist;    /* list of enabled interfaces */
151   } uartalloc;
152
153   void ns16552intr(int);
154
155   /*
156    * means the kernel is using this for debugging output
157    */
158   static char       Ekinuse[] = "device in use by kernel";
159
160   /*
161    *  pick up architecture specific routines and definitions
162    */
163   #include "ns16552.h"
164
165   /*
166    *  set the baud rate by calculating and setting the baudrate
167    *  generator constant.  This will work with fairly non-standard
168    *  baud rates.
169    */
170   static void
171   ns16552setbaud(Uart *p, int rate)
172   {
173     ulong brconst;
174
175     if(rate <= 0)
176             return;
177
178     brconst = (p->freq+8*rate-1)/(16*rate);
179
180     uartwrreg(p, Format, Dra);
181     outb(p->port + Dmsb, (brconst>>8) & 0xff);
182     outb(p->port + Dlsb, brconst & 0xff);
183     uartwrreg(p, Format, 0);
184
185     p->baud = rate;
186   }
187
188   /*
189    * decide if we should hangup when dsr or dcd drops.
190    */
191   static void
192   ns16552dsrhup(Uart *p, int n)
193   {
194     p->hup_dsr = n;
195   }
196
197   static void
198   ns16552dcdhup(Uart *p, int n)
199   {
200     p->hup_dcd = n;
201   }
202
203   static void
204   ns16552parity(Uart *p, char type)
205   {
206     switch(type){
207     case 'e':
208             p->sticky[Format] |= Pena|Peven;
209             break;
210     case 'o':
211             p->sticky[Format] &= ~Peven;
212             p->sticky[Format] |= Pena;
213             break;
214     default:
215             p->sticky[Format] &= ~(Pena|Peven);
216             break;
217     }
218     uartwrreg(p, Format, 0);
219   }
220
221   /*
222    *  set bits/character, default 8
223    */
224   void
225   ns16552bits(Uart *p, int bits)
226   {
227     if(bits < 5 || bits > 8)
228             error(Ebadarg);
229
230     p->sticky[Format] &= ~3;
231     p->sticky[Format] |= bits-5;
232
233     uartwrreg(p, Format, 0);
234   }
235
236   /*
237    *  toggle DTR
238    */
239   void
240   ns16552dtr(Uart *p, int n)
241   {
242     if(n)
243             p->sticky[Mctl] |= Dtr;
244     else
245             p->sticky[Mctl] &= ~Dtr;
246
247     uartwrreg(p, Mctl, 0);
248   }
249
250   /*
251    *  toggle RTS
252    */
253   void
254   ns16552rts(Uart *p, int n)
255   {
256     if(n)
257             p->sticky[Mctl] |= Rts;
258     else
259             p->sticky[Mctl] &= ~Rts;
260
261     uartwrreg(p, Mctl, 0);
262   }
263
264   /*
265    *  save dcd timestamps for gps clock
266    */
267   static void
268   ns16552dcdts(Uart *p, int n)
269   {
270     p->dcdts = n;
271   }
272
```

```
/*
 *  send break
 */
static void
ns16552break(Uart *p, int ms)
{
  if(ms == 0)
          ms = 200;

  uartwrreg(p, Format, Break);
  tsleep(&up->sleep, return0, 0, ms);
  uartwrreg(p, Format, 0);
}

static void
ns16552fifoon(Uart *p)
{
  ulong i, x;

  if(p->type < Ns550)
          return;

  x = splhi();

  /* reset fifos */
  uartwrreg(p, Fifoctl, Fclear);

  /* empty buffer and interrupt conditions */
  for(i = 0; i < 16; i++){
          if(uartrdreg(p, Istat))
                  ;
          if(uartrdreg(p, Data))
                  ;
  }

  /* turn on fifo */
  p->fifoon = 1;
  if(p->type == Ns650)
          uartwrreg(p, Fifoctl, Fena|Ftrig24);
  else
          uartwrreg(p, Fifoctl, Fena|Ftrig4);

  if((p->istat & Fenabd) == 0){
          /* didn't work, must be an earlier chip type */
          p->type = Ns450;
  }

  splx(x);
}

/*
 *  modem flow control on/off (rts/cts)
 */
static void
ns16552mflow(Uart *p, int n)
{

  ilock(&p->tlock);
  if(n){
          if(p->type == Ns650){
                  outb(p->port + Format, 0xbf);
                  outb(p->port + Efr, Eena|Arts|Acts);
                  uartwrreg(p, Format, 0);
                  p->cts = 1;
          } else {
                  p->sticky[Iena] |= Imstat;
                  uartwrreg(p, Iena, 0);
                  p->modem = 1;
                  p->cts = uartrdreg(p, Mstat) & Cts;
          }
  } else {
          if(p->type == Ns650){
                  outb(p->port + Format, 0xbf);
                  outb(p->port + Efr, Eena|Arts|Acts);
                  uartwrreg(p, Format, 0);
          } else {
                  p->sticky[Iena] &= ~Imstat;
                  uartwrreg(p, Iena, 0);
                  p->modem = 0;
          }
          p->cts = 1;
  }
  iunlock(&p->tlock);

  ilock(&p->flock);
  if(n)
          /* turn on fifo's */
          ns16552fifoon(p);
  else {
          /* turn off fifo's */
          p->fifoon = 0;
          uartwrreg(p, Fifoctl, 0);
  }
  iunlock(&p->flock);
}

/*
 *  turn on a port's interrupts.  set DTR and RTS
 */
static void
ns16552enable(Uart *p)
{
  Uart **l;

  if(p->enabled)
          return;

  uartpower(p->dev, 1);

  p->hup_dsr = p->hup_dcd = 0;
  p->cts = p->dsr = p->dcd = 0;

  /*
   *  turn on interrupts
   */
  p->sticky[Iena] = Ircv | Ixmt | Irstat;
  uartwrreg(p, Iena, 0);

  /*
   *  turn on DTR and RTS
   */
  ns16552dtr(p, 1);
  ns16552rts(p, 1);

  /*
   *  assume we can send
   */
  ilock(&p->tlock);
  p->cts = 1;
  p->blocked = 0;
  iunlock(&p->tlock);

  /*
   *  set baud rate to the last used
   */
  ns16552setbaud(p, p->baud);
```

```
409
410     lock(&uartalloc);
411     for(l = &uartalloc.elist; *l; l = &(*l)->elist){
412             if(*l == p)
413                     break;
414     }
415     if(*l == 0){
416             p->elist = uartalloc.elist;
417             uartalloc.elist = p;
418     }
419     p->enabled = 1;
420     unlock(&uartalloc);
421   }
422
423   /*
424    *  turn off a port's interrupts.  reset DTR and RTS
425    */
426   static void
427   ns16552disable(Uart *p)
428   {
429     Uart **l;
430
431     /*
432      *  turn off interrupts
433      */
434     p->sticky[Iena] = 0;
435     uartwrreg(p, Iena, 0);
436
437     /*
438      *  revert to default settings
439      */
440     p->sticky[Format] = Bits8;
441     uartwrreg(p, Format, 0);
442
443     /*
444      *  turn off DTR, RTS, hardware flow control & fifo's
445      */
446     ns16552dtr(p, 0);
447     ns16552rts(p, 0);
448     ns16552mflow(p, 0);
449     ilock(&p->tlock);
450     p->xonoff = p->blocked = 0;
451     iunlock(&p->tlock);
452
453     uartpower(p->dev, 0);
454
455     lock(&uartalloc);
456     for(l = &uartalloc.elist; *l; l = &(*l)->elist){
457             if(*l == p){
458                     *l = p->elist;
459                     break;
460             }
461     }
462     p->enabled = 0;
463     unlock(&uartalloc);
464   }
465
466   /*
467    *  put some bytes into the local queue to avoid calling
468    *  qconsume for every character
469    */
470   static int
471   stageoutput(Uart *p)
472   {
473     int n;
474
475     n = qconsume(p->oq, p->ostage, Stagesize);
476     if(n <= 0)
477             return 0;
478     p->op = p->ostage;
479     p->oe = p->ostage + n;
480     return n;
481   }
482
483   /*
484    *  (re)start output
485    */
486   static void
487   ns16552kick0(void *v)
488   {
489     int i;
490     Uart *p;
491
492     p = v;
493     if(p->cts == 0 || p->blocked)
494             return;
495
496     /*
497      *  128 here is an arbitrary limit to make sure
498      *  we don't stay in this loop too long.  If the
499      *  chips output queue is longer than 128, too
500      *  bad -- presotto
501      */
502     for(i = 0; i < 128; i++){
503             if(!(uartrdreg(p, Lstat) & Outready))
504                     break;
505             if(p->op >= p->oe && stageoutput(p) == 0)
506                     break;
507             outb(p->port + Data, *(p->op++));
508     }
509   }
510
511   static void
512   ns16552kick(void *v)
513   {
514     Uart *p;
515
516     p = v;
517     ilock(&p->tlock);
518     ns16552kick0(p);
519     iunlock(&p->tlock);
520   }
521
522   /*
523    *  restart input if it's off
524    */
525   static void
526   ns16552flow(void *v)
527   {
528     Uart *p;
529
530     p = v;
531     if(p->modem){
532             ns16552rts(p, 1);
533             ilock(&p->rlock);
534             p->haveinput = 1;
535             iunlock(&p->rlock);
536     }
537   }
538
539   /*
540    *  default is 9600 baud, 1 stop bit, 8 bit chars, no interrupts,
541    *  transmit and receive enabled, interrupts disabled.
542    */
543   static void
544   ns16552setup0(Uart *p)
```

```
{
  memset(p->sticky, 0, sizeof(p->sticky));
  /*
   *  set rate to 9600 baud.
   *  8 bits/character.
   *  1 stop bit.
   *  interrupts enabled.
   */
  p->sticky[Format] = Bits8;
  uartwrreg(p, Format, 0);
  p->sticky[Mctl] |= Inton;
  uartwrreg(p, Mctl, 0x0);

  ns16552setbaud(p, 9600);

  p->iq = qopen(4*1024, 0, ns16552flow, p);
  p->oq = qopen(4*1024, 0, ns16552kick, p);
  if(p->iq == nil || p->oq == nil)
          panic("ns16552setup0");

  p->ip = p->istage;
  p->ie = &p->istage[Stagesize];
  p->op = p->ostage;
  p->oe = p->ostage;
}

/*
 *  called by main() to create a new duart
 */
void
ns16552setup(ulong port, ulong freq, char *name, int type)
{
  Uart *p;

  if(nuart >= Nuart)
          return;

  p = xalloc(sizeof(Uart));
  uart[nuart] = p;
  strcpy(p->name, name);
  p->dev = nuart;
  nuart++;
  p->port = port;
  p->freq = freq;
  p->type = type;
  ns16552setup0(p);
}

/*
 *  called by main() to configure a duart port as a console or a mouse
 */
void
ns16552special(int port, int baud, Queue **in, Queue **out, int (*putc)(Queue*, int))
{
  Uart *p = uart[port];

  ns16552enable(p);
  if(baud)
          ns16552setbaud(p, baud);
  p->putc = putc;
  if(in)
          *in = p->iq;
  if(out)
          *out = p->oq;
  p->opens++;
}

/*
 *  handle an interrupt to a single uart
 */
void
ns16552intr(int dev)
{
  uchar ch;
  int s, l, loops;
  Uart *p = uart[dev];

  for(loops = 0;; loops++){
          p->istat = s = uartrdreg(p, Istat);
          if(loops > 10000000){
                  print("lstat %ux mstat %ux istat %ux iena %ux ferr %d oerr %d",
                          uartrdreg(p, Lstat), uartrdreg(p, Mstat),
                          s, uartrdreg(p, Iena), p->frame, p->overrun);
                  panic("ns16552intr");
          }
          switch(s&0x3f){
          case 6: /* receiver line status */
                  l = uartrdreg(p, Lstat);
                  if(l & Ferror)
                          p->frame++;
                  if(l & Oerror)
                          p->overrun++;
                  break;

          case 4: /* received data available */
          case 12:
                  ch = uartrdreg(p, Data) & 0xff;
                  if(p->xonoff){
                          if(ch == CTLS){
                                  p->blocked = 1;
                          }else if (ch == CTLQ){
                                  p->blocked = 0;
                                  p->ctsbackoff = 2; /* clock gets output going again */
                          }
                  }
                  if(p->putc)
                          p->putc(p->iq, ch);
                  else {
                          ilock(&p->rlock);
                          if(p->ip < p->ie)
                                  *p->ip++ = ch;
                          p->haveinput = 1;
                          iunlock(&p->rlock);
                  }
                  break;

          case 2: /* transmitter not full */
                  ns16552kick(p);
                  break;

          case 0: /* modem status */
                  ch = uartrdreg(p, Mstat);
                  if(ch & Ctsc){
                          l = p->cts;
                          p->cts = ch & Cts;
                          if(l == 0 && p->cts)
                                  p->ctsbackoff = 2; /* clock gets output going again */
                  }
                  if (ch & Dsrc) {
                          l = ch & Dsr;
                          if(p->hup_dsr && p->dsr && !l)
                                  p->dohup = 1;    /* clock peforms hangup */
                          p->dsr = l;
                  }
                  if (ch & Dcdc) {
                          l = ch & Dcd;
```

```
681                             if(l == 0 && p->dcd != 0 && p->dcdts && saveintrts != nil)
682                                     (*saveintrts)();
683                             if(p->hup_dcd && p->dcd && !l)
684                                     p->dohup = 1;    /* clock peforms hangup */
685                             p->dcd = l;
686                     }
687                     break;
688
689             default:
690                     if(s&1)
691                             return;
692                     print("weird modem interrupt #%2.2ux\n", s);
693                     break;
694             }
695       }
696     }
697
698     /*
699      *  we save up input characters till clock time
700      *
701      *  There's also a bit of code to get a stalled print going.
702      *  It shouldn't happen, but it does.  Obviously I don't
703      *  understand something.  Since it was there, I bundled a
704      *  restart after flow control with it to give some hysteresis
705      *  to the hardware flow control.  This makes compressing
706      *  modems happier but will probably bother something else.
707      *          -- presotto
708      */
709     void
710     uartclock(void)
711     {
712       int n;
713       Uart *p;
714
715       for(p = uartalloc.elist; p; p = p->elist){
716
717               /* this amortizes cost of qproduce to many chars */
718               if(p->haveinput){
719                       ilock(&p->rlock);
720                       if(p->haveinput){
721                               n = p->ip - p->istage;
722                               if(n > 0 && p->iq){
723                                       if(n > Stagesize)
724                                               panic("uartclock");
725                                       if(qproduce(p->iq, p->istage, n) < 0)
726                                               ns16552rts(p, 0);
727                                       else
728                                               p->ip = p->istage;
729                               }
730                               p->haveinput = 0;
731                       }
732                       iunlock(&p->rlock);
733               }
734               if(p->dohup){
735                       ilock(&p->rlock);
736                       if(p->dohup){
737                               qhangup(p->iq, 0);
738                               qhangup(p->oq, 0);
739                       }
740                       p->dohup = 0;
741                       iunlock(&p->rlock);
742               }
743
744               /* this adds hysteresis to hardware/software flow control */
745               if(p->ctsbackoff){
746                       ilock(&p->tlock);
747                       if(p->ctsbackoff){
748                               if(--(p->ctsbackoff) == 0)
749                                       ns16552kick0(p);
750                       }
751                       iunlock(&p->tlock);
752               }
753       }
754     }
755
756     Dirtab *ns16552dir;
757     int ndir;
758
759     static void
760     setlength(int i)
761     {
762       Uart *p;
763
764       if(i > 0){
765               p = uart[i];
766               if(p && p->opens && p->iq)
767                       ns16552dir[3*i].length = qlen(p->iq);
768       } else for(i = 0; i < nuart; i++){
769               p = uart[i];
770               if(p && p->opens && p->iq)
771                       ns16552dir[3*i].length = qlen(p->iq);
772       }
773     }
774
775     /*
776      *  all uarts must be ns16552setup() by this point or inside of ns16552install()
777      */
778     static void
779     ns16552reset(void)
780     {
781       int i;
782       Dirtab *dp;
783
784       ns16552install();       /* architecture specific */
785
786       ndir = 3*nuart;
787       ns16552dir = xalloc(ndir * sizeof(Dirtab));
788       dp = ns16552dir;
789       for(i = 0; i < nuart; i++){
790               /* 3 directory entries per port */
791               strcpy(dp->name, uart[i]->name);
792               dp->qid.path = NETQID(i, Ndataqid);
793               dp->perm = 0660;
794               dp++;
795               sprint(dp->name, "%sctl", uart[i]->name);
796               dp->qid.path = NETQID(i, Nctlqid);
797               dp->perm = 0660;
798               dp++;
799               sprint(dp->name, "%sstat", uart[i]->name);
800               dp->qid.path = NETQID(i, Nstatqid);
801               dp->perm = 0444;
802               dp++;
803       }
804     }
805
806     static Chan*
807     ns16552attach(char *spec)
808     {
809       return devattach('t', spec);
810     }
811
812     static int
813     ns16552walk(Chan *c, char *name)
814     {
815       return devwalk(c, name, ns16552dir, ndir, devgen);
816     }
```

```
817
818   static void
819   ns16552stat(Chan *c, char *dp)
820   {
821     if(NETTYPE(c->qid.path) == Ndataqid)
822             setlength(NETID(c->qid.path));
823     devstat(c, dp, ns16552dir, ndir, devgen);
824   }
825
826   static Chan*
827   ns16552open(Chan *c, int omode)
828   {
829     Uart *p;
830
831     c = devopen(c, omode, ns16552dir, ndir, devgen);
832
833     switch(NETTYPE(c->qid.path)){
834     case Nctlqid:
835     case Ndataqid:
836             p = uart[NETID(c->qid.path)];
837             if(p->kinuse)
838                     error(Ekinuse);
839             qlock(p);
840             if(p->opens++ == 0){
841                     ns16552enable(p);
842                     qreopen(p->iq);
843                     qreopen(p->oq);
844             }
845             qunlock(p);
846             break;
847     }
848
849     return c;
850   }
851
852   static void
853   ns16552close(Chan *c)
854   {
855     Uart *p;
856
857     if(c->qid.path & CHDIR)
858             return;
859     if((c->flag & COPEN) == 0)
860             return;
861     switch(NETTYPE(c->qid.path)){
862     case Ndataqid:
863     case Nctlqid:
864             p = uart[NETID(c->qid.path)];
865             if(p->kinuse)
866                     error(Ekinuse);
867             qlock(p);
868             if(--(p->opens) == 0){
869                     ns16552disable(p);
870                     qclose(p->iq);
871                     qclose(p->oq);
872                     p->ip = p->istage;
873                     p->dcd = p->dsr = p->dohup = 0;
874             }
875             qunlock(p);
876             break;
877     }
878   }
879
880   static long
881   uartstatus(Chan*, Uart *p, void *buf, long n, long offset)
882   {
883     uchar mstat, fstat, istat, tstat;
884     char str[256];
885
886     str[0] = 0;
887     tstat = p->sticky[Mctl];
888     mstat = uartrdreg(p, Mstat);
889     istat = p->sticky[Iena];
890     fstat = p->sticky[Format];
891     snprint(str, sizeof str,
892             "b%d c%d d%d e%d l%d m%d p%c r%d s%d\n"
893             "%d %d %d%s%s%s%s%s\n",
894
895             p->baud,
896             p->hup_dcd,
897             (tstat & Dtr) != 0,
898             p->hup_dsr,
899             (fstat & Bits8) + 5,
900             (istat & Imstat) != 0,
901             (fstat & Pena) ? ((fstat & Peven) ? 'e' : 'o') : 'n',
902             (tstat & Rts) != 0,
903             (fstat & Stop2) ? 2 : 1,
904
905             p->dev,
906             p->frame,
907             p->overrun,
908             p->fifoon       ? " fifo" : "",
909             (mstat & Cts)    ? " cts"  : "",
910             (mstat & Dsr)    ? " dsr"  : "",
911             (mstat & Dcd)    ? " dcd"  : "",
912             (mstat & Ring)   ? " ring" : ""
913     );
914     return readstr(offset, buf, n, str);
915   }
916
917   static long
918   ns16552read(Chan *c, void *buf, long n, vlong off)
919   {
920     Uart *p;
921     ulong offset = off;
922
923     if(c->qid.path & CHDIR){
924             setlength(-1);
925             return devdirread(c, buf, n, ns16552dir, ndir, devgen);
926     }
927
928     p = uart[NETID(c->qid.path)];
929     if(p->kinuse)
930             error(Ekinuse);
931     switch(NETTYPE(c->qid.path)){
932     case Ndataqid:
933             return qread(p->iq, buf, n);
934     case Nctlqid:
935             return readnum(offset, buf, n, NETID(c->qid.path), NUMSIZE);
936     case Nstatqid:
937             return uartstatus(c, p, buf, n, offset);
938     }
939
940     return 0;
941   }
942
943   static void
944   ns16552ctl(Uart *p, char *cmd)
945   {
946     int i, n;
947
948     /* let output drain for a while */
949     for(i = 0; i < 16 && qlen(p->oq); i++)
950             tsleep(&p->r, (int(*)(void*))qlen, p->oq, 125);
951
952     if(strncmp(cmd, "break", 5) == 0){
```

```
953             ns16552break(p, 0);
954             return;
955     }
956
957
958     n = atoi(cmd+1);
959     switch(*cmd){
960     case 'B':
961     case 'b':
962             ns16552setbaud(p, n);
963             break;
964     case 'C':
965     case 'c':
966             ns16552dcdhup(p, n);
967             break;
968     case 'D':
969     case 'd':
970             ns16552dtr(p, n);
971             break;
972     case 'E':
973     case 'e':
974             ns16552dsrhup(p, n);
975             break;
976     case 'f':
977     case 'F':
978             qflush(p->oq);
979             break;
980     case 'H':
981     case 'h':
982             qhangup(p->iq, 0);
983             qhangup(p->oq, 0);
984             break;
985     case 'L':
986     case 'l':
987             ns16552bits(p, n);
988             break;
989     case 'm':
990     case 'M':
991             ns16552mflow(p, n);
992             break;
993     case 'n':
994     case 'N':
995             qnoblock(p->oq, n);
996             break;
997     case 'P':
998     case 'p':
999             ns16552parity(p, *(cmd+1));
1000            break;
1001    case 'K':
1002    case 'k':
1003            ns16552break(p, n);
1004            break;
1005    case 'R':
1006    case 'r':
1007            ns16552rts(p, n);
1008            break;
1009    case 'Q':
1010    case 'q':
1011            qsetlimit(p->iq, n);
1012            qsetlimit(p->oq, n);
1013            break;
1014    case 'T':
1015    case 't':
1016            ns16552dcdts(p, n);
1017            break;
1018    case 'W':
1019    case 'w':
1020            /* obsolete */
1021            break;
1022    case 'X':
1023    case 'x':
1024            ilock(&p->tlock);
1025            p->xonoff = n;
1026            iunlock(&p->tlock);
1027            break;
1028    }
1029  }
1030
1031  static long
1032  ns16552write(Chan *c, void *buf, long n, vlong)
1033  {
1034    Uart *p;
1035    char cmd[32];
1036
1037    if(c->qid.path & CHDIR)
1038            error(Eperm);
1039
1040    p = uart[NETID(c->qid.path)];
1041    if(p->kinuse)
1042            error(Ekinuse);
1043
1044    /*
1045     *  The fifo's turn themselves off sometimes.
1046     *  It must be something I don't understand. -- presotto
1047     */
1048    lock(&p->flock);
1049    if((p->istat & Fenabd) == 0 && p->fifoon && p->type < Ns550)
1050            ns16552fifoon(p);
1051    unlock(&p->flock);
1052
1053    switch(NETTYPE(c->qid.path)){
1054    case Ndataqid:
1055            return qwrite(p->oq, buf, n);
1056    case Nctlqid:
1057            if(n >= sizeof(cmd))
1058                    n = sizeof(cmd)-1;
1059            memmove(cmd, buf, n);
1060            cmd[n] = 0;
1061            ns16552ctl(p, cmd);
1062            return n;
1063    }
1064  }
1065
1066  static void
1067  ns16552wstat(Chan *c, char *dp)
1068  {
1069    Dir d;
1070    Dirtab *dt;
1071
1072    if(!iseve())
1073            error(Eperm);
1074    if(CHDIR & c->qid.path)
1075            error(Eperm);
1076    if(NETTYPE(c->qid.path) == Nstatqid)
1077            error(Eperm);
1078
1079    dt = &ns16552dir[3 * NETID(c->qid.path)];
1080    convM2D(dp, &d);
1081    d.mode &= 0666;
1082    dt[0].perm = dt[1].perm = d.mode;
1083  }
1084
1085  Dev ns16552devtab = {
1086    't',
1087    "ns16552",
1088
```

```
1089    ns16552reset,
1090    devinit,
1091    ns16552attach,
1092    devclone,
1093    ns16552walk,
1094    ns16552stat,
1095    ns16552open,
1096    devcreate,
1097    ns16552close,
1098    ns16552read,
1099    devbread,
1100    ns16552write,
1101    devbwrite,
1102    devremove,
1103    ns16552wstat,
1104  };
1105
1106  /*
1107   * Polling I/O routines for kernel debugging helps.
1108   */
1109  static void
1110  ns16552setuppoll(Uart *p, int rate)
1111  {
1112    ulong brconst;
1113
1114    /*
1115     * 8 bits/character, 1 stop bit;
1116     * set rate to rate baud;
1117     * turn on Rts and Dtr.
1118     */
1119    memmove(p->osticky, p->sticky, sizeof p->osticky);
1120
1121    p->sticky[Format] = Bits8;
1122    uartwrreg(p, Format, 0);
1123
1124    brconst = (UartFREQ+8*rate-1)/(16*rate);
1125    uartwrreg(p, Format, Dra);
1126    outb(p->port+Dmsb, (brconst>>8) & 0xff);
1127    outb(p->port+Dlsb, brconst & 0xff);
1128    uartwrreg(p, Format, 0);
1129
1130    p->sticky[Mctl] = Rts|Dtr;
1131    uartwrreg(p, Mctl, 0);
1132
1133    p->kinuse = 1;
1134  }
1135
1136  /*
1137   * Restore serial state from before kernel took over.
1138   */
1139  static void
1140  ns16552restore(Uart *p)
1141  {
1142    ulong brconst;
1143
1144    memmove(p->sticky, p->osticky, sizeof p->sticky);
1145    uartwrreg(p, Format, 0);
1146
1147    brconst = (UartFREQ+8*p->baud-1)/(16*p->baud);
1148    uartwrreg(p, Format, Dra);
1149    outb(p->port+Dmsb, (brconst>>8) & 0xff);
1150    outb(p->port+Dlsb, brconst & 0xff);
1151    uartwrreg(p, Format, 0);
1152
1153    uartwrreg(p, Mctl, 0);
1154    p->kinuse = 0;
1155  }
1156
1157  /* BUG should be configurable */
1158  enum {
1159    WHICH = 0,
1160    RATE = 9600,
1161  };
1162
1163  int
1164  serialgetc(void)
1165  {
1166    Uart *p;
1167    int c;
1168
1169    if((p=uart[WHICH]) == nil)
1170            return -1;
1171    if(!p->kinuse)
1172            ns16552setuppoll(p, RATE);
1173
1174    while((uartrdreg(p, Lstat)&Inready) == 0)
1175            ;
1176    c = inb(p->port+Data) & 0xFF;
1177    return c;
1178    // there should be a restore here but i think it will slow things down too much.
1179  }
1180
1181  static void
1182  serialputc(Uart *p, int c)
1183  {
1184    while((uartrdreg(p, Lstat)&Outready) == 0)
1185            ;
1186    outb(p->port+Data, c);
1187    while((uartrdreg(p, Lstat)&Outready) == 0)
1188            ;
1189  }
1190
1191  void
1192  serialputs(char *s, int n)
1193  {
1194    Uart *p;
1195
1196    if((p=uart[WHICH]) == nil)
1197            return;
1198    if(!p->kinuse)
1199            ns16552setuppoll(p, RATE);
1200
1201    while(n-- > 0){
1202            serialputc(p, *s);
1203            if(*s == '\n')
1204                    serialputc(p, '\r');
1205            s++;
1206    }
1207    ns16552restore(p);
1208  }
```

```
#include  "u.h"
#include  "../port/lib.h"
#include  "mem.h"
#include  "dat.h"
#include  "fns.h"
#include  "../port/error.h"

#include  "netif.h"

typedef struct Pipe       Pipe;
struct Pipe
{
  QLock;
  Pipe    *next;
  int     ref;
  ulong   path;
  Queue   *q[2];
  int     qref[2];
};

struct
{
  Lock;
  ulong   path;
} pipealloc;

enum
{
  Qdir,
  Qdata0,
  Qdata1,
};

Dirtab pipedir[] =
{
  "data",         {Qdata0},       0,                      0600,
  "data1",        {Qdata1},       0,                      0600,
};
#define NPIPEDIR 2

static void
pipeinit(void)
{
  if(conf.pipeqsize == 0){
          if(conf.nmach > 1)
                  conf.pipeqsize = 256*1024;
          else
                  conf.pipeqsize = 32*1024;
  }
}

/*
 *  create a pipe, no streams are created until an open
 */
static Chan*
pipeattach(char *spec)
{
  Pipe *p;
  Chan *c;

  c = devattach('|', spec);
  p = malloc(sizeof(Pipe));
  if(p == 0)
          exhausted("memory");
  p->ref = 1;

  p->q[0] = qopen(conf.pipeqsize, 0, 0, 0);
  if(p->q[0] == 0){
          free(p);
          exhausted("memory");
  }
  p->q[1] = qopen(conf.pipeqsize, 0, 0, 0);
  if(p->q[1] == 0){
          free(p->q[0]);
          free(p);
          exhausted("memory");
  }

  lock(&pipealloc);
  p->path = ++pipealloc.path;
  unlock(&pipealloc);

  c->qid = (Qid){CHDIR|NETQID(2*p->path, Qdir), 0};
  c->aux = p;
  c->dev = 0;
  return c;
}

static Chan*
pipeclone(Chan *c, Chan *nc)
{
  Pipe *p;

  p = c->aux;
  nc = devclone(c, nc);
  qlock(p);
  p->ref++;
  if(c->flag & COPEN){
          switch(NETTYPE(c->qid.path)){
          case Qdata0:
                  p->qref[0]++;
                  break;
          case Qdata1:
                  p->qref[1]++;
                  break;
          }
  }
  qunlock(p);
  return nc;
}

static int
pipegen(Chan *c, Dirtab *tab, int ntab, int i, Dir *dp)
{
  int id;
  int len;
  Pipe *p;

  if(i == DEVDOTDOT){
          devdir(c, c->qid, "#|", 0, eve, CHDIR|0555, dp);
          return 1;
  }

  id = NETID(c->qid.path);
  if(i > 1)
          id++;
  if(tab==0 || i>=ntab)
          return -1;
  tab += i;
  p = c->aux;
  switch(tab->qid.path){
  case Qdata0:
          len = qlen(p->q[0]);
          break;
  case Qdata1:
          len = qlen(p->q[1]);
```

```
137             break;
138     default:
139             len = tab->length;
140             break;
141     }
142     devdir(c, (Qid){NETQID(id, tab->qid.path),0}, tab->name, len, eve, tab->perm, dp);
143     return 1;
144   }
145
146
147   static int
148   pipewalk(Chan *c, char *name)
149   {
150     return devwalk(c, name, pipedir, NPIPEDIR, pipegen);
151   }
152
153   static void
154   pipestat(Chan *c, char *db)
155   {
156     Pipe *p;
157     Dir dir;
158
159     p = c->aux;
160
161     switch(NETTYPE(c->qid.path)){
162     case Qdir:
163             devdir(c, c->qid, ".", 2*DIRLEN, eve, CHDIR|0555, &dir);
164             break;
165     case Qdata0:
166             devdir(c, c->qid, "data", qlen(p->q[0]), eve, 0660, &dir);
167             break;
168     case Qdata1:
169             devdir(c, c->qid, "data1", qlen(p->q[1]), eve, 0660, &dir);
170             break;
171     default:
172             panic("pipestat");
173     }
174     convD2M(&dir, db);
175   }
176
177   /*
178    *  if the stream doesn't exist, create it
179    */
180   static Chan*
181   pipeopen(Chan *c, int omode)
182   {
183     Pipe *p;
184
185     if(c->qid.path & CHDIR){
186             if(omode != OREAD)
187                     error(Ebadarg);
188             c->mode = omode;
189             c->flag |= COPEN;
190             c->offset = 0;
191             return c;
192     }
193
194     p = c->aux;
195     qlock(p);
196     switch(NETTYPE(c->qid.path)){
197     case Qdata0:
198             p->qref[0]++;
199             break;
200     case Qdata1:
201             p->qref[1]++;
202             break;
203     }
204     qunlock(p);
205
206     c->mode = openmode(omode);
207     c->flag |= COPEN;
208     c->offset = 0;
209     return c;
210   }
211
212   static void
213   pipeclose(Chan *c)
214   {
215     Pipe *p;
216
217     p = c->aux;
218     qlock(p);
219
220     if(c->flag & COPEN){
221             /*
222              *  closing either side hangs up the stream
223              */
224             switch(NETTYPE(c->qid.path)){
225             case Qdata0:
226                     p->qref[0]--;
227                     if(p->qref[0] == 0){
228                             qhangup(p->q[1], 0);
229                             qclose(p->q[0]);
230                     }
231                     break;
232             case Qdata1:
233                     p->qref[1]--;
234                     if(p->qref[1] == 0){
235                             qhangup(p->q[0], 0);
236                             qclose(p->q[1]);
237                     }
238                     break;
239             }
240     }
241
242
243     /*
244      *  if both sides are closed, they are reusable
245      */
246     if(p->qref[0] == 0 && p->qref[1] == 0){
247             qreopen(p->q[0]);
248             qreopen(p->q[1]);
249     }
250
251     /*
252      *  free the structure on last close
253      */
254     p->ref--;
255     if(p->ref == 0){
256             qunlock(p);
257             free(p->q[0]);
258             free(p->q[1]);
259             free(p);
260     } else
261             qunlock(p);
262   }
263
264   static long
265   piperead(Chan *c, void *va, long n, vlong)
266   {
267     Pipe *p;
268
269     p = c->aux;
270
271     switch(NETTYPE(c->qid.path)){
272     case Qdir:
```

```
273             return devdirread(c, va, n, pipedir, NPIPEDIR, pipegen);
274     case Qdata0:
275             return qread(p->q[0], va, n);
276     case Qdata1:
277             return qread(p->q[1], va, n);
278     default:
279             panic("piperead");
280     }
281     return -1;      /* not reached */
282   }
283
284   static Block*
285   pipebread(Chan *c, long n, ulong offset)
286   {
287     Pipe *p;
288
289     p = c->aux;
290
291     switch(NETTYPE(c->qid.path)){
292     case Qdata0:
293             return qbread(p->q[0], n);
294     case Qdata1:
295             return qbread(p->q[1], n);
296     }
297
298     return devbread(c, n, offset);
299   }
300
301   /*
302    *  a write to a closed pipe causes a note to be sent to
303    *  the process.
304    */
305   static long
306   pipewrite(Chan *c, void *va, long n, vlong)
307   {
308     Pipe *p;
309
310     if(!islo())
311             print("pipewrite hi %lux\n", getcallerpc(&c));
312     if(waserror()) {
313             /* avoid notes when pipe is a mounted queue */
314             if((c->flag & CMSG) == 0)
315                     postnote(up, 1, "sys: write on closed pipe", NUser);
316             nexterror();
317     }
318
319     p = c->aux;
320
321     switch(NETTYPE(c->qid.path)){
322     case Qdata0:
323             n = qwrite(p->q[1], va, n);
324             break;
325
326     case Qdata1:
327             n = qwrite(p->q[0], va, n);
328             break;
329
330     default:
331             panic("pipewrite");
332     }
333
334     poperror();
335     return n;
336   }
337
338   static long
339   pipebwrite(Chan *c, Block *bp, ulong)
340   {
341     long n;
342     Pipe *p;
343
344     if(waserror()) {
345             /* avoid notes when pipe is a mounted queue */
346             if((c->flag & CMSG) == 0)
347                     postnote(up, 1, "sys: write on closed pipe", NUser);
348             nexterror();
349     }
350
351     p = c->aux;
352     switch(NETTYPE(c->qid.path)){
353     case Qdata0:
354             n = qbwrite(p->q[1], bp);
355             break;
356
357     case Qdata1:
358             n = qbwrite(p->q[0], bp);
359             break;
360
361     default:
362             n = 0;
363             panic("pipebwrite");
364     }
365
366     poperror();
367     return n;
368   }
369
370   Dev pipedevtab = {
371     '|',
372     "pipe",
373
374     devreset,
375     pipeinit,
376     pipeattach,
377     pipeclone,
378     pipewalk,
379     pipestat,
380     pipeopen,
381     devcreate,
382     pipeclose,
383     piperead,
384     pipebread,
385     pipewrite,
386     pipebwrite,
387     devremove,
388     devwstat,
389   };
```

```
#include  "u.h"
#include  "../port/lib.h"
#include  "mem.h"
#include  "dat.h"
#include  "fns.h"
#include  "../port/error.h"
#include  "ureg.h"

enum
{
  Qctl,
  Qdir,
  Qfd,
  Qfpregs,
  Qkregs,
  Qmem,
  Qnote,
  Qnoteid,
  Qnotepg,
  Qns,
  Qproc,
  Qregs,
  Qsegment,
  Qstatus,
  Qtext,
  Qwait,
  Qprofile,
};

#define   STATSIZE        (2*NAMELEN+12+9*12)
Dirtab procdir[] =
{
  "ctl",          {Qctl},         0,                      0000,
  "fd",           {Qfd},          0,                      0000,
  "fpregs",       {Qfpregs},      sizeof(FPsave),         0000,
  "kregs",        {Qkregs},       sizeof(Ureg),           0440,
  "mem",          {Qmem},         0,                      0000,
  "note",         {Qnote},        0,                      0000,
  "noteid",       {Qnoteid},      0,                      0666,
  "notepg",       {Qnotepg},      0,                      0000,
  "ns",           {Qns},          0,                      0444,
  "proc",         {Qproc},        0,                      0400,
  "regs",         {Qregs},        sizeof(Ureg),           0000,
  "segment",      {Qsegment},     0,                      0444,
  "status",       {Qstatus},      STATSIZE,               0444,
  "text",         {Qtext},        0,                      0000,
  "wait",         {Qwait},        0,                      0400,
  "profile",      {Qprofile},     0,                      0400,
};

/* Segment type from portdat.h */
char *sname[]={ "Text", "Data", "Bss", "Stack", "Shared", "Phys", "Shdata", "Map" };

/*
 * Qids are, in path:
 *         4 bits of file type (qids above)
 *        23 bits of process slot number + 1
 *             in vers,
 *        32 bits of pid, for consistency checking
 * If notepg, c->pgrpid.path is pgrp slot, .vers is noteid.
 */
#define   QSHIFT  5       /* location in qid of proc slot # */

#define   QID(q)          (((q).path&0x0000001F)>>0)
#define   SLOT(q)         ((((q).path&0x07FFFFE0)>>QSHIFT)-1)
#define   PID(q)          ((q).vers)
#define   NOTEID(q)       ((q).vers)

void      procctlreq(Proc*, char*, int);
int       procctlmemio(Proc*, ulong, int, void*, int);
Chan*     proctext(Chan*, Proc*);
Segment* txt2data(Proc*, Segment*);
int       procstopped(void*);
void      mntscan(Mntwalk*, Proc*);

static int
procgen(Chan *c, Dirtab *tab, int, int s, Dir *dp)
{
  Qid qid;
  Proc *p;
  Segment *q;
  char buf[NAMELEN];
  ulong pid, path, perm, len;

  if(s == DEVDOTDOT){
          c->qid.path = CHDIR;
          devdir(c, c->qid, "#p", 0, eve, 0555, dp);
          return 1;
  }

  if(c->qid.path == CHDIR){
          if(s >= conf.nproc)
                  return -1;
          p = proctab(s);
          pid = p->pid;
          if(pid == 0)
                  return 0;
          sprint(buf, "%lud", pid);
          qid = (Qid){CHDIR|((s+1)<<QSHIFT), pid};
          devdir(c, qid, buf, 0, p->user, CHDIR|0555, dp);
          return 1;
  }
  if(s >= nelem(procdir))
          return -1;
  if(tab)
          panic("procgen");

  tab = &procdir[s];
  path = c->qid.path&~(CHDIR|((1<<QSHIFT)-1));    /* slot component */

  p = proctab(SLOT(c->qid));
  perm = tab->perm;
  if(perm == 0)
          perm = p->procmode;

  len = tab->length;
  switch(QID(c->qid)) {
  case Qwait:
          len = p->nwait * sizeof(Waitmsg);
          break;
  case Qprofile:
          q = p->seg[TSEG];
          if(q && q->profile) {
                  len = (q->top-q->base)>>LRESPROF;
                  len *= sizeof(*q->profile);
          }
          break;
  }

  qid = (Qid){path|tab->qid.path, c->qid.vers};
  devdir(c, qid, tab->name, len, p->user, perm, dp);
  return 1;
}

static void
procinit(void)
```

```
{
  if(conf.nproc >= (1<<(16-QSHIFT))-1)
          print("warning: too many procs for devproc\n");
}

static Chan*
procattach(char *spec)
{
  return devattach('p', spec);
}

static int
procwalk(Chan *c, char *name)
{
  return devwalk(c, name, 0, 0, procgen);
}

static void
procstat(Chan *c, char *db)
{
  devstat(c, db, 0, 0, procgen);
}

static Chan*
procopen(Chan *c, int omode)
{
  Proc *p;
  Pgrp *pg;
  Chan *tc;
  int pid;

  if(c->qid.path & CHDIR)
          return devopen(c, omode, 0, 0, procgen);

  p = proctab(SLOT(c->qid));
  qlock(&p->debug);
  if(waserror()){
          qunlock(&p->debug);
          nexterror();
  }
  pid = PID(c->qid);
  if(p->pid != pid)
          error(Eprocdied);

  omode = openmode(omode);

  switch(QID(c->qid)){
  case Qtext:
          if(omode != OREAD)
                  error(Eperm);
          tc = proctext(c, p);
          tc->offset = 0;
          qunlock(&p->debug);
          poperror();
          return tc;

  case Qproc:
  case Qkregs:
  case Qsegment:
  case Qprofile:
  case Qfd:
          if(omode != OREAD)
                  error(Eperm);
          break;

  case Qctl:
  case Qnote:
  case Qnoteid:
  case Qmem:
  case Qstatus:
  case Qwait:
  case Qregs:
  case Qfpregs:
          break;

  case Qns:
          if(omode != OREAD)
                  error(Eperm);
          c->aux = malloc(sizeof(Mntwalk));
          break;

  case Qnotepg:
          pg = p->pgrp;
          if(pg == nil)
                  error(Eprocdied);
          if(omode!=OWRITE || pg->pgrpid == 1)
                  error(Eperm);
          c->pgrpid.path = pg->pgrpid+1;
          c->pgrpid.vers = p->noteid;
          break;

  default:
          pprint("procopen %lux\n", c->qid);
          error(Egreg);
  }

  /* Affix pid to qid */
  if(p->state != Dead)
          c->qid.vers = p->pid;

  /* make sure the process slot didn't get reallocated while we were playing */
  coherence();
  if(p->pid != pid)
          error(Eprocdied);

  tc = devopen(c, omode, 0, 0, procgen);
  qunlock(&p->debug);
  poperror();

  return tc;
}

static void
procwstat(Chan *c, char *db)
{
  Proc *p;
  Dir d;

  if(c->qid.path&CHDIR)
          error(Eperm);

  p = proctab(SLOT(c->qid));
  if(waserror()){
          qunlock(&p->debug);
          nexterror();
  }
  qlock(&p->debug);

  if(p->pid != PID(c->qid))
          error(Eprocdied);

  if(strcmp(up->user, p->user) != 0 && strcmp(up->user, eve) != 0)
          error(Eperm);

  convM2D(db, &d);
  if(strcmp(d.uid, p->user) != 0){
```

```
273             if(strcmp(up->user, eve) != 0)
274                     error(Eperm);
275             else {
276                     strncpy(p->user, d.uid, sizeof(p->user));
277                     p->user[sizeof(p->user)-1] = 0;
278             }
279     }
280     p->procmode = d.mode&0777;
281
282     poperror();
283     qunlock(&p->debug);
284   }
285
286   static int
287   procfds(Proc *p, char *va, int count, long offset)
288   {
289     Fgrp *f;
290     Chan *c;
291     int n, i;
292
293     qlock(&p->debug);
294     f = p->fgrp;
295     if(f == nil){
296             qunlock(&p->debug);
297             return 0;
298     }
299     lock(f);
300     if(waserror()){
301             unlock(f);
302             qunlock(&p->debug);
303             nexterror();
304     }
305
306     n = readstr(0, va, count, p->dot->name->s);
307     n += snprint(va+n, count-n, "\n");
308     for(i = 0; i <= f->maxfd; i++) {
309             c = f->fd[i];
310             if(c == nil)
311                     continue;
312             n += snprint(va+n, count-n, "%3d %.2s %C %4ld %.8lux.%.8lud %8lld ",
313                     i,
314                     &"r w rw"[(c->mode&3)<<1],
315                     devtab[c->type]->dc, c->dev,
316                     c->qid.path, c->qid.vers,
317                     c->offset);
318             n += readstr(0, va+n, count-n, c->name->s);
319             n += snprint(va+n, count-n, "\n");
320             if(offset > 0) {
321                     offset -= n;
322                     if(offset < 0) {
323                             memmove(va, va+n+offset, -offset);
324                             n = -offset;
325                     }
326                     else
327                             n = 0;
328             }
329     }
330     unlock(f);
331     qunlock(&p->debug);
332     poperror();
333
334     return n;
335   }
336
337   static void
338   procclose(Chan * c)
339   {
340     if(QID(c->qid) == Qns && c->aux != 0)
341             free(c->aux);
342   }
343
344   static void
345   int2flag(int flag, char *s)
346   {
347     if(flag == 0){
348             *s = '\0';
349             return;
350     }
351     *s++ = '-';
352     if(flag & MAFTER)
353             *s++ = 'a';
354     if(flag & MBEFORE)
355             *s++ = 'b';
356     if(flag & MCREATE)
357             *s++ = 'c';
358     if(flag & MCACHE)
359             *s++ = 'C';
360     *s = '\0';
361   }
362
363   static long
364   procread(Chan *c, void *va, long n, vlong off)
365   {
366     long l;
367     Proc *p;
368     Waitq *wq;
369     Ureg kur;
370     uchar *rptr;
371     Mntwalk *mw;
372     Segment *sg, *s;
373     char *a = va, *sps;
374     int i, j, rsize, pid;
375     char statbuf[NSEG*32], *srv, flag[10];
376     ulong offset = off;
377
378     if(c->qid.path & CHDIR)
379             return devdirread(c, a, n, 0, 0, procgen);
380
381     p = proctab(SLOT(c->qid));
382     if(p->pid != PID(c->qid))
383             error(Eprocdied);
384
385     switch(QID(c->qid)){
386     case Qmem:
387             if(offset < KZERO
388             || (offset >= USTKTOP-USTKSIZE && offset < USTKTOP))
389                     return procctlmemio(p, offset, n, va, 1);
390
391             /* Protect crypt key memory */
392             if(offset+n >= palloc.cmembase && offset < palloc.cmemtop)
393                     error(Eperm);
394
395             /* validate physical kernel addresses */
396             if(offset < (ulong)end) {
397                     if(offset+n > (ulong)end)
398                             n = (ulong)end - offset;
399                     memmove(a, (char*)offset, n);
400                     return n;
401             }
402             if(offset >= conf.base0 && offset < conf.npage0){
403                     if(offset+n > conf.npage0)
404                             n = conf.npage0 - offset;
405                     memmove(a, (char*)offset, n);
406                     return n;
407             }
408             if(offset >= conf.base1 && offset < conf.npage1){
```

```
409                     if(offset+n > conf.npage1)
410                             n = conf.npage1 - offset;
411                     memmove(a, (char*)offset, n);
412                     return n;
413             }
414             error(Ebadarg);
415     case Qprofile:
416             s = p->seg[TSEG];
417             if(s == 0 || s->profile == 0)
418                     error("profile is off");
419             i = (s->top-s->base)>>LRESPROF;
420             i *= sizeof(*s->profile);
421             if(offset >= i)
422                     return 0;
423             if(offset+n > i)
424                     n = i - offset;
425             memmove(a, ((char*)s->profile)+offset, n);
426             return n;
427
428     case Qnote:
429             qlock(&p->debug);
430             if(waserror()){
431                     qunlock(&p->debug);
432                     nexterror();
433             }
434             if(p->pid != PID(c->qid))
435                     error(Eprocdied);
436             if(n < ERRLEN)
437                     error(Etoosmall);
438             if(p->nnote == 0)
439                     n = 0;
440             else {
441                     memmove(va, p->note[0].msg, ERRLEN);
442                     p->nnote--;
443                     memmove(p->note, p->note+1, p->nnote*sizeof(Note));
444                     n = ERRLEN;
445             }
446             if(p->nnote == 0)
447                     p->notepending = 0;
448             poperror();
449             qunlock(&p->debug);
450             return n;
451
452     case Qproc:
453             if(offset >= sizeof(Proc))
454                     return 0;
455             if(offset+n > sizeof(Proc))
456                     n = sizeof(Proc) - offset;
457             memmove(a, ((char*)p)+offset, n);
458             return n;
459
460     case Qregs:
461             rptr = (uchar*)p->dbgreg;
462             rsize = sizeof(Ureg);
463             goto regread;
464
465     case Qkregs:
466             memset(&kur, 0, sizeof(Ureg));
467             setkernur(&kur, p);
468             rptr = (uchar*)&kur;
469             rsize = sizeof(Ureg);
470             goto regread;
471
472     case Qfpregs:
473             rptr = (uchar*)&p->fpsave;
474             rsize = sizeof(FPsave);
475     regread:
476             if(rptr == 0)
477                     error(Enoreg);
478             if(offset >= rsize)
479                     return 0;
480             if(offset+n > rsize)
481                     n = rsize - offset;
482             memmove(a, rptr+offset, n);
483             return n;
484
485     case Qstatus:
486             if(offset >= STATSIZE)
487                     return 0;
488             if(offset+n > STATSIZE)
489                     n = STATSIZE - offset;
490
491             sps = p->psstate;
492             if(sps == 0)
493                     sps = statename[p->state];
494             memset(statbuf, ' ', sizeof statbuf);
495             memmove(statbuf+0*NAMELEN, p->text, strlen(p->text));
496             memmove(statbuf+1*NAMELEN, p->user, strlen(p->user));
497             memmove(statbuf+2*NAMELEN, sps, strlen(sps));
498             j = 2*NAMELEN + 12;
499
500             for(i = 0; i < 6; i++) {
501                     l = p->time[i];
502                     if(i == TReal)
503                             l = MACHP(0)->ticks - l;
504                     l = TK2MS(l);
505                     readnum(0, statbuf+j+NUMSIZE*i, NUMSIZE, l, NUMSIZE);
506             }
507             /* ignore stack, which is mostly non-existent */
508             l = 0;
509             for(i=1; i<NSEG; i++){
510                     s = p->seg[i];
511                     if(s)
512                             l += s->top - s->base;
513             }
514             readnum(0, statbuf+j+NUMSIZE*6, NUMSIZE, l>>10, NUMSIZE);
515             readnum(0, statbuf+j+NUMSIZE*7, NUMSIZE, p->basepri, NUMSIZE);
516             readnum(0, statbuf+j+NUMSIZE*8, NUMSIZE, p->priority, NUMSIZE);
517             memmove(a, statbuf+offset, n);
518             return n;
519
520     case Qsegment:
521             j = 0;
522             for(i = 0; i < NSEG; i++) {
523                     sg = p->seg[i];
524                     if(sg == 0)
525                             continue;
526                     j += sprint(statbuf+j, "%-6s %c%c %.8lux %.8lux %4ld\n",
527                             sname[sg->type&SG_TYPE],
528                             sg->type&SG_RONLY ? 'R' : ' ',
529                             sg->profile ? 'P' : ' ',
530                             sg->base, sg->top, sg->ref);
531             }
532             if(offset >= j)
533                     return 0;
534             if(offset+n > j)
535                     n = j-offset;
536             if(n == 0 && offset == 0)
537                     exhausted("segments");
538             memmove(a, &statbuf[offset], n);
539             return n;
540
541     case Qwait:
542             if(n < sizeof(Waitmsg))
543                     error(Etoosmall);
544
```

```
545             if(!canqlock(&p->qwaitr))
546                     error(Einuse);
547
548             if(waserror()) {
549                     qunlock(&p->qwaitr);
550                     nexterror();
551             }
552
553             lock(&p->exl);
554             if(up == p && p->nchild == 0 && p->waitq == 0) {
555                     unlock(&p->exl);
556                     error(Enochild);
557             }
558             pid = p->pid;
559             while(p->waitq == 0) {
560                     unlock(&p->exl);
561                     sleep(&p->waitr, haswaitq, p);
562                     if(p->pid != pid)
563                             error(Eprocdied);
564                     lock(&p->exl);
565             }
566             wq = p->waitq;
567             p->waitq = wq->next;
568             p->nwait--;
569             unlock(&p->exl);
570
571             qunlock(&p->qwaitr);
572             poperror();
573             memmove(a, &wq->w, sizeof(Waitmsg));
574             free(wq);
575             return sizeof(Waitmsg);
576
577     case Qns:
578             qlock(&p->debug);
579             if(waserror()){
580                     qunlock(&p->debug);
581                     nexterror();
582             }
583             if(p->pgrp == nil || p->pid != PID(c->qid))
584                     error(Eprocdied);
585             mw = c->aux;
586             if(mw->cddone){
587                     qunlock(&p->debug);
588                     poperror();
589                     return 0;
590             }
591             mntscan(mw, p);
592             if(mw->mh == 0){
593                     mw->cddone = 1;
594                     i = snprint(a, n, "cd %s\n", p->dot->name->s);
595                     qunlock(&p->debug);
596                     poperror();
597                     return i;
598             }
599             int2flag(mw->cm->flag, flag);
600             if(strcmp(mw->cm->to->name->s, "#M") == 0){
601                     srv = srvname(mw->cm->to->mchan);
602                     i = snprint(a, n, "mount %s %s %s %.*s\n", flag,
603                             srv==nil? mw->cm->to->mchan->name->s : srv,
604                             mw->mh->from->name->s, NAMELEN, mw->cm->spec);
605                     free(srv);
606             }else
607                     i = snprint(a, n, "bind %s %s %s\n", flag,
608                             mw->cm->to->name->s, mw->mh->from->name->s);
609             qunlock(&p->debug);
610             poperror();
611             return i;
612
613     case Qnoteid:
614             return readnum(offset, va, n, p->noteid, NUMSIZE);
615     case Qfd:
616             return procfds(p, va, n, offset);
617     }
618     error(Egreg);
619     return 0;               /* not reached */
620   }
621
622   void
623   mntscan(Mntwalk *mw, Proc *p)
624   {
625     Pgrp *pg;
626     Mount *t;
627     Mhead *f;
628     int nxt, i;
629     ulong last, bestmid;
630
631     pg = p->pgrp;
632     rlock(&pg->ns);
633
634     nxt = 0;
635     bestmid = ~0;
636
637     last = 0;
638     if(mw->mh)
639             last = mw->cm->mountid;
640
641     for(i = 0; i < MNTHASH; i++) {
642             for(f = pg->mnthash[i]; f; f = f->hash) {
643                     for(t = f->mount; t; t = t->next) {
644                             if(mw->mh == 0 ||
645                               (t->mountid > last && t->mountid < bestmid)) {
646                                     mw->cm = t;
647                                     mw->mh = f;
648                                     bestmid = mw->cm->mountid;
649                                     nxt = 1;
650                             }
651                     }
652             }
653     }
654     if(nxt == 0)
655             mw->mh = 0;
656
657     runlock(&pg->ns);
658   }
659
660   static long
661   procwrite(Chan *c, void *va, long n, vlong off)
662   {
663     int id;
664     Proc *p, *t, *et;
665     char *a, buf[ERRLEN];
666     ulong offset = off;
667
668     a = va;
669     if(c->qid.path & CHDIR)
670             error(Eisdir);
671
672     p = proctab(SLOT(c->qid));
673
674     /* Use the remembered noteid in the channel rather
675      * than the process pgrpid
676      */
677     if(QID(c->qid) == Qnotepg) {
678             pgrpnote(NOTEID(c->pgrpid), va, n, NUser);
679             return n;
680     }
```

```
681
682     qlock(&p->debug);
683     if(waserror()){
684             qunlock(&p->debug);
685             nexterror();
686     }
687     if(p->pid != PID(c->qid))
688             error(Eprocdied);
689
690     switch(QID(c->qid)){
691     case Qmem:
692             if(p->state != Stopped)
693                     error(Ebadctl);
694
695             n = procctlmemio(p, offset, n, va, 0);
696             break;
697
698     case Qregs:
699             if(offset >= sizeof(Ureg))
700                     return 0;
701             if(offset+n > sizeof(Ureg))
702                     n = sizeof(Ureg) - offset;
703             if(p->dbgreg == 0)
704                     error(Enoreg);
705             setregisters(p->dbgreg, (char*)(p->dbgreg)+offset, va, n);
706             break;
707
708     case Qfpregs:
709             if(offset >= sizeof(FPsave))
710                     return 0;
711             if(offset+n > sizeof(FPsave))
712                     n = sizeof(FPsave) - offset;
713             memmove((uchar*)&p->fpsave+offset, va, n);
714             break;
715
716     case Qctl:
717             procctlreq(p, va, n);
718             break;
719
720     case Qnote:
721             if(p->kp)
722                     error(Eperm);
723             if(n >= ERRLEN-1)
724                     error(Etoobig);
725             memmove(buf, va, n);
726             buf[n] = 0;
727             if(!postnote(p, 0, buf, NUser))
728                     error("note not posted");
729             break;
730     case Qnoteid:
731             id = atoi(a);
732             if(id == p->pid) {
733                     p->noteid = id;
734                     break;
735             }
736             t = proctab(0);
737             for(et = t+conf.nproc; t < et; t++) {
738                     if(id == t->noteid) {
739                             if(strcmp(p->user, t->user) != 0)
740                                     error(Eperm);
741                             p->noteid = id;
742                             break;
743                     }
744             }
745             if(p->noteid != id)
746                     error(Ebadarg);
747             break;
748     default:
749             pprint("unknown qid in procwrite\n");
750             error(Egreg);
751     }
752     poperror();
753     qunlock(&p->debug);
754     return n;
755   }
756
757   Dev procdevtab = {
758     'p',
759     "proc",
760
761     devreset,
762     procinit,
763     procattach,
764     devclone,
765     procwalk,
766     procstat,
767     procopen,
768     devcreate,
769     procclose,
770     procread,
771     devbread,
772     procwrite,
773     devbwrite,
774     devremove,
775     procwstat,
776   };
777
778   Chan*
779   proctext(Chan *c, Proc *p)
780   {
781     Chan *tc;
782     Image *i;
783     Segment *s;
784
785     s = p->seg[TSEG];
786     if(s == 0)
787             error(Enonexist);
788     if(p->state==Dead)
789             error(Eprocdied);
790
791     lock(s);
792     i = s->image;
793     if(i == 0) {
794             unlock(s);
795             error(Eprocdied);
796     }
797     unlock(s);
798
799     lock(i);
800     if(waserror()) {
801             unlock(i);
802             nexterror();
803     }
804
805     tc = i->c;
806     if(tc == 0)
807             error(Eprocdied);
808
809     if(incref(tc) == 1 || (tc->flag&COPEN) == 0 || tc->mode!=OREAD) {
810             cclose(tc);
811             error(Eprocdied);
812     }
813
814     if(p->pid != PID(c->qid))
815             error(Eprocdied);
816
```

```
817     unlock(i);
818     poperror();
819
820     return tc;
821   }
822
823   void
824   procstopwait(Proc *p, int ctl)
825   {
826     int pid;
827
828     if(p->pdbg)
829             error(Einuse);
830     if(procstopped(p))
831             return;
832
833     if(ctl != 0)
834             p->procctl = ctl;
835     p->pdbg = up;
836     pid = p->pid;
837     qunlock(&p->debug);
838     up->psstate = "Stopwait";
839     if(waserror()) {
840             p->pdbg = 0;
841             qlock(&p->debug);
842             nexterror();
843     }
844     sleep(&up->sleep, procstopped, p);
845     poperror();
846     qlock(&p->debug);
847     if(p->pid != pid)
848             error(Eprocdied);
849   }
850
851   void
852   procctlfgrp(Fgrp *f)
853   {
854     int i;
855     Chan *c;
856
857     lock(f);
858     f->ref++;
859     for(i = 0; i < f->maxfd; i++) {
860             c = f->fd[i];
861             if(c != 0) {
862                     f->fd[i] = 0;
863                     unlock(f);
864                     cclose(c);
865                     lock(f);
866             }
867     }
868     unlock(f);
869     closefgrp(f);
870   }
871
872   void
873   procctlreq(Proc *p, char *va, int n)
874   {
875     Segment *s;
876     int i, npc;
877     char buf[NAMELEN];
878
879     if(n > NAMELEN)
880             n = NAMELEN;
881     strncpy(buf, va, n);
882
883     if(strncmp(buf, "stop", 4) == 0)
884             procstopwait(p, Proc_stopme);
885     else
886     if(strncmp(buf, "kill", 4) == 0) {
887             switch(p->state) {
888             case Broken:
889                     unbreak(p);
890                     break;
891             case Stopped:
892                     postnote(p, 0, "sys: killed", NExit);
893                     p->procctl = Proc_exitme;
894                     ready(p);
895                     break;
896             default:
897                     postnote(p, 0, "sys: killed", NExit);
898                     p->procctl = Proc_exitme;
899             }
900     }
901     else
902     if(strncmp(buf, "hang", 4) == 0)
903             p->hang = 1;
904     else
905     if(strncmp(buf, "nohang", 6) == 0)
906             p->hang = 0;
907     else
908     if(strncmp(buf, "waitstop", 8) == 0)
909             procstopwait(p, 0);
910     else
911     if(strncmp(buf, "startstop", 9) == 0) {
912             if(p->state != Stopped)
913                     error(Ebadctl);
914             p->procctl = Proc_traceme;
915             ready(p);
916             procstopwait(p, Proc_traceme);
917     }
918     else
919     if(strncmp(buf, "start", 5) == 0) {
920             if(p->state != Stopped)
921                     error(Ebadctl);
922             ready(p);
923     }
924     else
925     if(strncmp(buf, "closefiles", 10) == 0){
926             qlock(&p->debug);
927             if(waserror()){
928                     qunlock(&p->debug);
929                     nexterror();
930             }
931             if(p->fgrp == nil)
932                     error(Eprocdied);
933             procctlfgrp(p->fgrp);
934             qunlock(&p->debug);
935             poperror();
936     }
937     else
938     if(strncmp(buf, "pri", 3) == 0) {
939             if(n < 4)
940                     error(Ebadctl);
941             i = atoi(buf+4);
942             if(i < 0)
943                     i = 0;
944             if(i >= Nrq)
945                     i = Nrq - 1;
946             if(i > p->basepri && !iseve())
947                     error(Eperm);
948             p->basepri = i;
949     }
950     else
951     if(strncmp(buf, "wired", 5) == 0) {
952             if(n < 6)
```

```
953                     error(Ebadctl);
954             i = atoi(buf+6);
955             procwired(p, i);
956     }
957     else
958     if(strncmp(buf, "profile", 7) == 0) {
959             s = p->seg[TSEG];
960             if(s == 0 || (s->type&SG_TYPE) != SG_TEXT)
961                     error(Ebadctl);
962             if(s->profile != 0)
963                     free(s->profile);
964             npc = (s->top-s->base)>>LRESPROF;
965             s->profile = malloc(npc*sizeof(*s->profile));
966             if(s->profile == 0)
967                     error(Enomem);
968     }
969     else
970             error(Ebadctl);
971   }
972
973   int
974   procstopped(void *a)
975   {
976     Proc *p = a;
977     return p->state == Stopped;
978   }
979
980   int
981   procctlmemio(Proc *p, ulong offset, int n, void *va, int read)
982   {
983     KMap *k;
984     Pte *pte;
985     Page *pg;
986     Segment *s;
987     ulong soff, l;
988     char *a = va, *b;
989
990     for(;;) {
991             s = seg(p, offset, 1);
992             if(s == 0)
993                     error(Ebadarg);
994
995             if(offset+n >= s->top)
996                     n = s->top-offset;
997
998             if(!read && (s->type&SG_TYPE) == SG_TEXT)
999                     s = txt2data(p, s);
1000
1001            s->steal++;
1002            soff = offset-s->base;
1003            if(waserror()) {
1004                    s->steal--;
1005                    nexterror();
1006            }
1007            if(fixfault(s, offset, read, 0) == 0)
1008                    break;
1009            poperror();
1010            s->steal--;
1011    }
1012    poperror();
1013    pte = s->map[soff/PTEMAPMEM];
1014    if(pte == 0)
1015            panic("procctlmemio");
1016    pg = pte->pages[(soff&(PTEMAPMEM-1))/BY2PG];
1017    if(pagedout(pg))
1018            panic("procctlmemio1");
1019
1020    l = BY2PG - (offset&(BY2PG-1));
1021    if(n > l)
1022            n = l;
1023
1024    k = kmap(pg);
1025    if(waserror()) {
1026            s->steal--;
1027            kunmap(k);
1028            nexterror();
1029    }
1030    b = (char*)VA(k);
1031    b += offset&(BY2PG-1);
1032    if(read == 1)
1033            memmove(a, b, n);       /* This can fault */
1034    else
1035            memmove(b, a, n);
1036    kunmap(k);
1037    poperror();
1038
1039    /* Ensure the process sees text page changes */
1040    if(s->flushme)
1041            memset(pg->cachectl, PG_TXTFLUSH, sizeof(pg->cachectl));
1042
1043    s->steal--;
1044
1045    if(read == 0)
1046            p->newtlb = 1;
1047
1048    return n;
1049  }
1050
1051  Segment*
1052  txt2data(Proc *p, Segment *s)
1053  {
1054    int i;
1055    Segment *ps;
1056
1057    ps = newseg(SG_DATA, s->base, s->size);
1058    ps->image = s->image;
1059    incref(ps->image);
1060    ps->fstart = s->fstart;
1061    ps->flen = s->flen;
1062    ps->flushme = 1;
1063
1064    qlock(&p->seglock);
1065    for(i = 0; i < NSEG; i++)
1066            if(p->seg[i] == s)
1067                    break;
1068    if(p->seg[i] != s)
1069            panic("segment gone");
1070
1071    qunlock(&s->lk);
1072    putseg(s);
1073    qlock(&ps->lk);
1074    p->seg[i] = ps;
1075    qunlock(&p->seglock);
1076
1077    return ps;
1078  }
1079
1080  Segment*
1081  data2txt(Segment *s)
1082  {
1083    Segment *ps;
1084
1085    ps = newseg(SG_TEXT, s->base, s->size);
1086    ps->image = s->image;
1087    incref(ps->image);
1088    ps->fstart = s->fstart;
```

```
1089    ps->flen = s->flen;
1090    ps->flushme = 1;
1091
1092    return ps;
1093  }
```

```
#include  "u.h"
#include  "../port/lib.h"
#include  "mem.h"
#include  "dat.h"
#include  "fns.h"
#include  "../port/error.h"

enum{
  Qdir=   0,

  Nfiles=32,      /* max root files */
};

extern ulong      bootlen;
extern uchar      bootcode[];

Dirtab rootdir[Nfiles];

static uchar      *rootdata[Nfiles];
static int        nroot = 0;

typedef struct Recover Recover;
struct Recover
{
  int     len;
  char    *req;
  Recover *next;
};

struct
{
  Lock;
  QLock;
  Rendez;
  Recover *q;
}reclist;

/*
 *  add a root file
 */
static void
addroot(char *name, uchar *contents, ulong len, int perm)
{
  Dirtab *d;

  if(nroot >= Nfiles)
          panic("too many root files");
  rootdata[nroot] = contents;
  d = &rootdir[nroot];
  strcpy(d->name, name);
  d->length = len;
  d->perm = perm;
  d->qid.path = nroot+1;
  if(perm & CHDIR)
          d->qid.path |= CHDIR;
  nroot++;
}

/*
 *  add a root file
 */
void
addrootfile(char *name, uchar *contents, ulong len)
{
  addroot(name, contents, len, 0555);
}

/*
 *  add a root file
 */
static void
addrootdir(char *name)
{
  addroot(name, nil, 0, CHDIR|0555);
}

static void
rootreset(void)
{
  addrootdir("bin");
  addrootdir("dev");
  addrootdir("env");
  addrootdir("net");
  addrootdir("net.alt");
  addrootdir("proc");
  addrootdir("root");
  addrootdir("srv");

  addrootfile("boot", bootcode, bootlen); /* always have a boot file */
}

static Chan*
rootattach(char *spec)
{
  return devattach('/', spec);
}

static int
rootwalk(Chan *c, char *name)
{
  if(strcmp(name, "..") == 0) {
          c->qid.path = Qdir|CHDIR;
          return 1;
  }
  if((c->qid.path & ~CHDIR) != Qdir)
          return 0;
  return devwalk(c, name, rootdir, nroot, devgen);
}

static void
rootstat(Chan *c, char *dp)
{
  devstat(c, dp, rootdir, nroot, devgen);
}

static Chan*
rootopen(Chan *c, int omode)
{
  switch(c->qid.path & ~CHDIR) {
  default:
          break;
  }

  return devopen(c, omode, rootdir, nroot, devgen);
}

/*
 * sysremove() knows this is a nop
 */
static void
rootclose(Chan *c)
{
  switch(c->qid.path) {
  default:
          break;
  }
```

```
}

static int
rdrdy(void*)
{
  return reclist.q != 0;
}

static long
rootread(Chan *c, void *buf, long n, vlong off)
{
  ulong t;
  Dirtab *d;
  uchar *data;
  ulong offset = off;

  t = c->qid.path & ~CHDIR;
  switch(t){
  case Qdir:
          return devdirread(c, buf, n, rootdir, nroot, devgen);
  }

  d = &rootdir[t-1];
  data = rootdata[t-1];
  if(offset >= d->length)
          return 0;
  if(offset+n > d->length)
          n = d->length - offset;
  memmove(buf, data+offset, n);
  return n;
}

static long
rootwrite(Chan *c, void*, long, vlong)
{
  switch(c->qid.path & ~CHDIR){
  default:
          error(Egreg);
  }
  return 0;
}

static void
rootcreate(Chan *c, char *name, int mode, ulong perm)
{
  if(!iseve() || c->qid.path != (CHDIR|Qdir) ||
     (perm & CHDIR) == 0 || mode != OREAD)
          error(Eperm);
  addrootdir(name);
  c->flag |= COPEN;
  c->mode = OREAD;
}

Dev rootdevtab = {
  '/',
  "root",

  rootreset,
  devinit,
  rootattach,
  devclone,
  rootwalk,
  rootstat,
  rootopen,
  rootcreate,
  rootclose,
  rootread,
  devbread,
  rootwrite,
  devbwrite,
  devremove,
  devwstat,
};
```

```
#include  "u.h"
#include  "../port/lib.h"
#include  "mem.h"
#include  "dat.h"
#include  "fns.h"
#include  "../port/error.h"

int
fault(ulong addr, int read)
{
  Segment *s;
  char *sps;

  sps = up->psstate;
  up->psstate = "Fault";
  spllo();

  m->pfault++;
  for(;;) {
          s = seg(up, addr, 1);           /* leaves s->lk qlocked if seg != nil */
          if(s == 0) {
                  up->psstate = sps;
                  return -1;
          }

          if(!read && (s->type&SG_RONLY)) {
                  qunlock(&s->lk);
                  up->psstate = sps;
                  return -1;
          }

          if(fixfault(s, addr, read, 1) == 0)
                  break;
  }

  up->psstate = sps;
  return 0;
}

static void
faulterror(char *s, int freemem)
{
  if(up->nerrlab) {
          postnote(up, 1, s, NDebug);
          error(s);
  }
  pexit(s, freemem);
}

int
fixfault(Segment *s, ulong addr, int read, int doputmmu)
{
  int type;
  int ref;
  Pte **p, *etp;
  char buf[ERRLEN];
  ulong va, mmuphys=0, soff;
  Page **pg, *lkp, *new;
  Page *(*fn)(Segment*, ulong);

  va = addr;
  addr &= ~(BY2PG-1);
  soff = addr-s->base;
  p = &s->map[soff/PTEMAPMEM];
  if(*p == 0)
          *p = ptealloc();

  etp = *p;
  pg = &etp->pages[(soff&(PTEMAPMEM-1))/BY2PG];
  type = s->type&SG_TYPE;

  if(pg < etp->first)
          etp->first = pg;
  if(pg > etp->last)
          etp->last = pg;

  switch(type) {
  default:
          panic("fault");
          break;

  case SG_TEXT:                   /* Demand load */
          if(pagedout(*pg))
                  pio(s, addr, soff, pg);

          mmuphys = PPN((*pg)->pa) | PTERONLY|PTEVALID;
          (*pg)->modref = PG_REF;
          break;

  case SG_BSS:
  case SG_SHARED:                 /* Zero fill on demand */
  case SG_STACK:
  case SG_MAP:
          if(*pg == 0) {
                  if(type == SG_MAP) {
                          sprint(buf, "map 0x%lux %c", va, read ? 'r' : 'w');
                          postnote(up, 1, buf, NDebug);
                  }
                  new = newpage(1, &s, addr);
                  if(s == 0)
                          return -1;

                  *pg = new;
          }
          goto common;

  case SG_DATA:
  case SG_SHDATA:
  common:                 /* Demand load/pagein/copy on write */
          if(pagedout(*pg))
                  pio(s, addr, soff, pg);

          if(read && conf.copymode == 0) {
                  mmuphys = PPN((*pg)->pa)|PTERONLY|PTEVALID;
                  (*pg)->modref |= PG_REF;
                  break;
          }

          lkp = *pg;
          lock(lkp);

          if(lkp->image == &swapimage)
                  ref = lkp->ref + swapcount(lkp->daddr);
          else
                  ref = lkp->ref;
          if(ref > 1) {
                  unlock(lkp);

                  if(swapfull()){
                          qunlock(&s->lk);
                          pprint("swap space full\n");
                          faulterror(Enoswap, 1);
                  }

                  new = newpage(0, &s, addr);
                  if(s == 0)
```

```
137                             return -1;
138                     *pg = new;
139                     copypage(lkp, *pg);
140                     putpage(lkp);
141             }
142             else {
143                     /* save a copy of the original for the image cache */
144                     if(lkp->image && !swapfull())
145                             duppage(lkp);
146
147                     unlock(lkp);
148             }
149             mmuphys = PPN((*pg)->pa) | PTEWRITE|PTEVALID;
150             (*pg)->modref = PG_MOD|PG_REF;
151             break;
152
153     case SG_PHYSICAL:
154             if(*pg == 0) {
155                     fn = s->pseg->pgalloc;
156                     if(fn)
157                             *pg = (*fn)(s, addr);
158                     else {
159                             new = smalloc(sizeof(Page));
160                             new->va = addr;
161                             new->pa = s->pseg->pa+(addr-s->base);
162                             new->ref = 1;
163                             *pg = new;
164                     }
165             }
166
167             mmuphys = PPN((*pg)->pa) |PTEWRITE|PTEUNCACHED|PTEVALID;
168             (*pg)->modref = PG_MOD|PG_REF;
169             break;
170     }
171     qunlock(&s->lk);
172
173     if(doputmmu)
174             putmmu(addr, mmuphys, *pg);
175
176     return 0;
177   }
178
179   void
180   pio(Segment *s, ulong addr, ulong soff, Page **p)
181   {
182     Page *new;
183     KMap *k;
184     Chan *c;
185     int n, ask;
186     char *kaddr;
187     ulong daddr;
188     Page *loadrec;
189
190   retry:
191     loadrec = *p;
192     if(loadrec == 0) {      /* from a text/data image */
193             daddr = s->fstart+soff;
194             new = lookpage(s->image, daddr);
195             if(new != nil) {
196                     *p = new;
197                     return;
198             }
199     }
200     else {                  /* from a swap image */
201             daddr = swapaddr(loadrec);
202             new = lookpage(&swapimage, daddr);
203             if(new != nil) {
204                     putswap(loadrec);
205                     *p = new;
206                     return;
207             }
208   }
209
210
211     qunlock(&s->lk);
212
213     new = newpage(0, 0, addr);
214     k = kmap(new);
215     kaddr = (char*)VA(k);
216
217     if(loadrec == 0) {                      /* This is demand load */
218             c = s->image->c;
219             while(waserror()) {
220                     if(strcmp(up->error, Eintr) == 0)
221                             continue;
222                     kunmap(k);
223                     putpage(new);
224                     faulterror("sys: demand load I/O error", 0);
225             }
226
227             ask = s->flen-soff;
228             if(ask > BY2PG)
229                     ask = BY2PG;
230
231             n = devtab[c->type]->read(c, kaddr, ask, daddr);
232             if(n != ask)
233                     faulterror(Eioload, 0);
234             if(ask < BY2PG)
235                     memset(kaddr+ask, 0, BY2PG-ask);
236
237             poperror();
238             kunmap(k);
239             qlock(&s->lk);
240
241             /*
242              *  race, another proc may have gotten here first while
243              *  s->lk was unlocked
244              */
245             if(*p == 0) {
246                     new->daddr = daddr;
247                     cachepage(new, s->image);
248                     *p = new;
249             }
250             else
251                     putpage(new);
252     }
253     else {                          /* This is paged out */
254             c = swapimage.c;
255             if(waserror()) {
256                     kunmap(k);
257                     putpage(new);
258                     qlock(&s->lk);
259                     qunlock(&s->lk);
260                     faulterror("sys: page in I/O error", 0);
261             }
262
263             n = devtab[c->type]->read(c, kaddr, BY2PG, daddr);
264             if(n != BY2PG)
265                     faulterror(Eioload, 0);
266
267             poperror();
268             kunmap(k);
269             qlock(&s->lk);
270
271             /*
272              *  race, another proc may have gotten here first
```

```
273             *  (and the pager may have run on that page) while
274             *  s->lk was unlocked
275             */
276             if(*p != loadrec){
277                     if(!pagedout(*p)){
278                             /* another process did it for me */
279                             putpage(new);
280                             goto done;
281                     } else {
282                             /* another process and the pager got in */
283                             putpage(new);
284                             goto retry;
285                     }
286             }
287
288             new->daddr = daddr;
289             cachepage(new, &swapimage);
290             *p = new;
291             putswap(loadrec);
292     }
293
294     done:
295       if(s->flushme)
296             memset((*p)->cachectl, PG_TXTFLUSH, sizeof((*p)->cachectl));
297     }
298
299     /*
300      * Called only in a system call
301      */
302     int
303     okaddr(ulong addr, ulong len, int write)
304     {
305       Segment *s;
306
307       if((long)len >= 0) {
308             for(;;) {
309                     s = seg(up, addr, 0);
310                     if(s == 0 || (write && (s->type&SG_RONLY)))
311                             break;
312
313                     if(addr+len > s->top) {
314                             len -= s->top - addr;
315                             addr = s->top;
316                             continue;
317                     }
318                     return 1;
319             }
320       }
321       pprint("suicide: invalid address 0x%lux in sys call pc=0x%lux\n", addr, userpc());
322       return 0;
323     }
324
325     void
326     validaddr(ulong addr, ulong len, int write)
327     {
328       if(!okaddr(addr, len, write))
329             pexit("Suicide", 0);
330     }
331
332     /*
333      * &s[0] is known to be a valid address.
334      */
335     void*
336     vmemchr(void *s, int c, int n)
337     {
338       int m;
339       char *t;
340       ulong a;
341
342       a = (ulong)s;
343       m = BY2PG - (a & (BY2PG-1));
344       if(m < n){
345             t = vmemchr(s, c, m);
346             if(t)
347                     return t;
348             if(!(a & KZERO))
349                     validaddr(a+m, 1, 0);
350             return vmemchr((void*)(a+m), c, n-m);
351       }
352       /*
353        * All in one page
354        */
355       return memchr(s, c, n);
356     }
357
358     Segment*
359     seg(Proc *p, ulong addr, int dolock)
360     {
361       Segment **s, **et, *n;
362
363       et = &p->seg[NSEG];
364       for(s = p->seg; s < et; s++) {
365             n = *s;
366             if(n == 0)
367                     continue;
368             if(addr >= n->base && addr < n->top) {
369                     if(dolock == 0)
370                             return n;
371
372                     qlock(&n->lk);
373                     if(addr >= n->base && addr < n->top)
374                             return n;
375                     qunlock(&n->lk);
376             }
377       }
378
379       return 0;
380     }
```

```
#include  "u.h"
#include  "../port/lib.h"
#include  "mem.h"
#include  "dat.h"
#include  "fns.h"
#include  "../port/error.h"

#define   pghash(daddr)   palloc.hash[(daddr>>PGSHIFT)&(PGHSIZE-1)]

struct    Palloc palloc;

void
pageinit(void)
{
  int color;
  Page *p;
  ulong np, vm, pm;

  np = palloc.np0+palloc.np1;
  palloc.head = xalloc(np*sizeof(Page));
  if(palloc.head == 0)
          panic("pageinit");

  color = 0;
  p = palloc.head;
  while(palloc.np0 > 0) {
          p->prev = p-1;
          p->next = p+1;
          p->pa = palloc.p0;
          p->color = color;
          palloc.freecount++;
          color = (color+1)%NCOLOR;
          palloc.p0 += BY2PG;
          palloc.np0--;
          p++;
  }
  while(palloc.np1 > 0) {
          p->prev = p-1;
          p->next = p+1;
          p->pa = palloc.p1;
          p->color = color;
          palloc.freecount++;
          color = (color+1)%NCOLOR;
          palloc.p1 += BY2PG;
          palloc.np1--;
          p++;
  }
  palloc.tail = p - 1;
  palloc.head->prev = 0;
  palloc.tail->next = 0;

  palloc.user = p - palloc.head;
  pm = palloc.user*BY2PG/1024;
  vm = pm + (conf.nswap*BY2PG)/1024;

  /* Pageing numbers */
  swapalloc.highwater = (palloc.user*5)/100;
  swapalloc.headroom = swapalloc.highwater + (swapalloc.highwater/4);

  print("%lud free pages\n", palloc.user);
  print("%ludK bytes\n", pm);
  print("%ludK swap\n", vm);
}

static void
pageunchain(Page *p)
{
  if(canlock(&palloc))
          panic("pageunchain");
  if(p->prev)
          p->prev->next = p->next;
  else
          palloc.head = p->next;
  if(p->next)
          p->next->prev = p->prev;
  else
          palloc.tail = p->prev;
  p->prev = p->next = nil;
  palloc.freecount--;
}

void
pagechaintail(Page *p)
{
  if(canlock(&palloc))
          panic("pagechaintail");
  if(palloc.tail) {
          p->prev = palloc.tail;
          palloc.tail->next = p;
  }
  else {
          palloc.head = p;
          p->prev = 0;
  }
  palloc.tail = p;
  p->next = 0;
  palloc.freecount++;
}

void
pagechainhead(Page *p)
{
  if(canlock(&palloc))
          panic("pagechainhead");
  if(palloc.head) {
          p->next = palloc.head;
          palloc.head->prev = p;
  }
  else {
          palloc.tail = p;
          p->next = 0;
  }
  palloc.head = p;
  p->prev = 0;
  palloc.freecount++;
}

Page*
newpage(int clear, Segment **s, ulong va)
{
  Page *p;
  KMap *k;
  uchar ct;
  int i, hw, dontalloc, color;

  lock(&palloc);
  color = getpgcolor(va);
  hw = swapalloc.highwater;
  for(;;) {
          if(palloc.freecount > hw)
                  break;
          if(up->kp && palloc.freecount > 0)
                  break;

          unlock(&palloc);
          dontalloc = 0;
```

```
137             if(s && *s) {
138                     qunlock(&((*s)->lk));
139                     *s = 0;
140                     dontalloc = 1;
141             }
142             qlock(&palloc.pwait);   /* Hold memory requesters here */
143
144             while(waserror())       /* Ignore interrupts */
145                     ;
146
147             kickpager();
148             tsleep(&palloc.r, ispages, 0, 1000);
149
150             poperror();
151
152             qunlock(&palloc.pwait);
153
154             /*
155              * If called from fault and we lost the segment from
156              * underneath don't waste time allocating and freeing
157              * a page. Fault will call newpage again when it has
158              * reacquired the segment locks
159              */
160             if(dontalloc)
161                     return 0;
162
163             lock(&palloc);
164     }
165
166     /* First try for our colour */
167     for(p = palloc.head; p; p = p->next)
168             if(p->color == color)
169                     break;
170
171     ct = PG_NOFLUSH;
172     if(p == 0) {
173             p = palloc.head;
174             p->color = color;
175             ct = PG_NEWCOL;
176     }
177
178     pageunchain(p);
179
180     lock(p);
181     if(p->ref != 0)
182             panic("newpage");
183
184     uncachepage(p);
185     p->ref++;
186     p->va = va;
187     p->modref = 0;
188     for(i = 0; i < MAXMACH; i++)
189             p->cachectl[i] = ct;
190     unlock(p);
191     unlock(&palloc);
192
193     if(clear) {
194             k = kmap(p);
195             memset((void*)VA(k), 0, BY2PG);
196             kunmap(k);
197     }
198
199     return p;
200   }
201
202   int
203   ispages(void*)
204   {
205     return palloc.freecount >= swapalloc.highwater;
206   }
207
208   void
209   putpage(Page *p)
210   {
211     if(onswap(p)) {
212             putswap(p);
213             return;
214     }
215
216     lock(&palloc);
217     lock(p);
218
219     if(p->ref == 0)
220             panic("putpage");
221
222     if(--p->ref > 0) {
223             unlock(p);
224             unlock(&palloc);
225             return;
226     }
227
228     if(p->image && p->image != &swapimage)
229             pagechaintail(p);
230     else
231             pagechainhead(p);
232
233     if(palloc.r.p != 0)
234             wakeup(&palloc.r);
235
236     unlock(p);
237     unlock(&palloc);
238   }
239
240   Page*
241   auxpage()
242   {
243     Page *p;
244
245     lock(&palloc);
246     p = palloc.head;
247     if(palloc.freecount < swapalloc.highwater) {
248             unlock(&palloc);
249             return 0;
250     }
251     pageunchain(p);
252
253     lock(p);
254     if(p->ref != 0)
255             panic("auxpage");
256     p->ref++;
257     uncachepage(p);
258     unlock(p);
259     unlock(&palloc);
260
261     return p;
262   }
263
264   void
265   duppage(Page *p)                          /* Always call with p locked */
266   {
267     Page *np;
268     int color;
269     int retries;
270
271     retries = 0;
272   retry:
```

```
273
274     if(retries++ > 10000)
275             panic("duppage %d", retries);
276
277     /* don't dup pages with no image */
278     if(p->ref == 0 || p->image == nil || p->image->notext)
279             return;
280
281     /*
282      *  normal lock ordering is to call
283      *  lock(&palloc) before lock(p).
284      *  To avoid deadlock, we have to drop
285      *  our locks and try again.
286      */
287     if(!canlock(&palloc)){
288             unlock(p);
289             if(up)
290                     sched();
291             lock(p);
292             goto retry;
293     }
294
295     /* No freelist cache when memory is very low */
296     if(palloc.freecount < swapalloc.highwater) {
297             unlock(&palloc);
298             uncachepage(p);
299             return;
300     }
301
302     color = getpgcolor(p->va);
303     for(np = palloc.head; np; np = np->next)
304             if(np->color == color)
305                     break;
306
307     /* No page of the correct color */
308     if(np == 0) {
309             unlock(&palloc);
310             uncachepage(p);
311             return;
312     }
313
314     pageunchain(np);
315     pagechaintail(np);
316
317     lock(np);
318     unlock(&palloc);
319
320     /* Cache the new version */
321     uncachepage(np);
322     np->va = p->va;
323     np->daddr = p->daddr;
324     copypage(p, np);
325     cachepage(np, p->image);
326     unlock(np);
327     uncachepage(p);
328   }
329
330   void
331   copypage(Page *f, Page *t)
332   {
333     KMap *ks, *kd;
334
335     ks = kmap(f);
336     kd = kmap(t);
337     memmove((void*)VA(kd), (void*)VA(ks), BY2PG);
338     kunmap(ks);
339     kunmap(kd);
340   }
341
342   void
343   uncachepage(Page *p)                    /* Always called with a locked page */
344   {
345     Page **l, *f;
346
347     if(p->image == 0)
348             return;
349
350     lock(&palloc.hashlock);
351     l = &pghash(p->daddr);
352     for(f = *l; f; f = f->hash) {
353             if(f == p) {
354                     *l = p->hash;
355                     break;
356             }
357             l = &f->hash;
358     }
359     unlock(&palloc.hashlock);
360     putimage(p->image);
361     p->image = 0;
362     p->daddr = 0;
363   }
364
365   void
366   cachepage(Page *p, Image *i)
367   {
368     Page **l;
369
370     /* If this ever happens it should be fixed by calling
371      * uncachepage instead of panic. I think there is a race
372      * with pio in which this can happen. Calling uncachepage is
373      * correct - I just wanted to see if we got here.
374      */
375     if(p->image)
376             panic("cachepage");
377
378     incref(i);
379     lock(&palloc.hashlock);
380     p->image = i;
381     l = &pghash(p->daddr);
382     p->hash = *l;
383     *l = p;
384     unlock(&palloc.hashlock);
385   }
386
387   void
388   cachedel(Image *i, ulong daddr)
389   {
390     Page *f, **l;
391
392     lock(&palloc.hashlock);
393     l = &pghash(daddr);
394     for(f = *l; f; f = f->hash) {
395             if(f->image == i && f->daddr == daddr) {
396                     lock(f);
397                     if(f->image == i && f->daddr == daddr){
398                             *l = f->hash;
399                             putimage(f->image);
400                             f->image = 0;
401                             f->daddr = 0;
402                     }
403                     unlock(f);
404                     break;
405             }
406             l = &f->hash;
407     }
408     unlock(&palloc.hashlock);
```

```
}

Page *
lookpage(Image *i, ulong daddr)
{
  Page *f;

  lock(&palloc.hashlock);
  for(f = pghash(daddr); f; f = f->hash) {
          if(f->image == i && f->daddr == daddr) {
                  unlock(&palloc.hashlock);

                  lock(&palloc);
                  lock(f);
                  if(f->image != i || f->daddr != daddr) {
                          unlock(f);
                          unlock(&palloc);
                          return 0;
                  }
                  if(++f->ref == 1)
                          pageunchain(f);
                  unlock(&palloc);
                  unlock(f);

                  return f;
          }
  }
  unlock(&palloc.hashlock);

  return 0;
}

Pte*
ptecpy(Pte *old)
{
  Pte *new;
  Page **src, **dst;

  new = ptealloc();
  dst = &new->pages[old->first-old->pages];
  new->first = dst;
  for(src = old->first; src <= old->last; src++, dst++)
          if(*src) {
                  if(onswap(*src))
                          dupswap(*src);
                  else {
                          lock(*src);
                          (*src)->ref++;
                          unlock(*src);
                  }
                  new->last = dst;
                  *dst = *src;
          }

  return new;
}

Pte*
ptealloc(void)
{
  Pte *new;

  new = smalloc(sizeof(Pte));
  new->first = &new->pages[PTEPERTAB];
  new->last = new->pages;
  return new;
}

void
freepte(Segment *s, Pte *p)
{
  int ref;
  void (*fn)(Page*);
  Page *pt, **pg, **ptop;

  switch(s->type&SG_TYPE) {
  case SG_PHYSICAL:
          fn = s->pseg->pgfree;
          ptop = &p->pages[PTEPERTAB];
          if(fn) {
                  for(pg = p->pages; pg < ptop; pg++) {
                          if(*pg == 0)
                                  continue;
                          (*fn)(*pg);
                          *pg = 0;
                  }
                  break;
          }
          for(pg = p->pages; pg < ptop; pg++) {
                  pt = *pg;
                  if(pt == 0)
                          continue;
                  lock(pt);
                  ref = --pt->ref;
                  unlock(pt);
                  if(ref == 0)
                          free(pt);
          }
          break;
  default:
          for(pg = p->first; pg <= p->last; pg++)
                  if(*pg) {
                          putpage(*pg);
                          *pg = 0;
                  }
  }
  free(p);
}
```

```
#include  "u.h"
#include  "../port/lib.h"
#include  "mem.h"
#include  "dat.h"
#include  "fns.h"
#include  "../port/error.h"

static Ref pgrpid;
static Ref mountid;

void
pgrpnote(ulong noteid, char *a, long n, int flag)
{
  Proc *p, *ep;
  char buf[ERRLEN];

  if(n >= ERRLEN-1)
          error(Etoobig);

  memmove(buf, a, n);
  buf[n] = 0;
  p = proctab(0);
  ep = p+conf.nproc;
  for(; p < ep; p++) {
          if(p->state == Dead)
                  continue;
          if(up != p && p->noteid == noteid && p->kp == 0) {
                  qlock(&p->debug);
                  if(p->pid == 0 || p->noteid != noteid){
                          qunlock(&p->debug);
                          continue;
                  }
                  if(!waserror()) {
                          postnote(p, 0, buf, flag);
                          poperror();
                  }
                  qunlock(&p->debug);
          }
  }
}

Pgrp*
newpgrp(void)
{
  Pgrp *p;

  p = smalloc(sizeof(Pgrp));
  p->ref = 1;
  p->pgrpid = incref(&pgrpid);
  return p;
}

Rgrp*
newrgrp(void)
{
  Rgrp *r;

  r = smalloc(sizeof(Rgrp));
  r->ref = 1;
  return r;
}

void
closergrp(Rgrp *r)
{
  if(decref(r) == 0)
          free(r);
}

void
closepgrp(Pgrp *p)
{
  Mhead **h, **e, *f, *next;

  if(decref(p) != 0)
          return;

  qlock(&p->debug);
  wlock(&p->ns);
  p->pgrpid = -1;

  e = &p->mnthash[MNTHASH];
  for(h = p->mnthash; h < e; h++) {
          for(f = *h; f; f = next) {
                  wlock(&f->lock);
                  cclose(f->from);
                  mountfree(f->mount);
                  f->mount = nil;
                  next = f->hash;
                  wunlock(&f->lock);
                  putmhead(f);
          }
  }
  wunlock(&p->ns);
  qunlock(&p->debug);
  free(p);
}

void
pgrpinsert(Mount **order, Mount *m)
{
  Mount *f;

  m->order = 0;
  if(*order == 0) {
          *order = m;
          return;
  }
  for(f = *order; f; f = f->order) {
          if(m->mountid < f->mountid) {
                  m->order = f;
                  *order = m;
                  return;
          }
          order = &f->order;
  }
  *order = m;
}

/*
 * pgrpcpy MUST preserve the mountid allocation order of the parent group
 */
void
pgrpcpy(Pgrp *to, Pgrp *from)
{
  int i;
  Mount *n, *m, **link, *order;
  Mhead *f, **tom, **l, *mh;

  wlock(&from->ns);
  order = 0;
  tom = to->mnthash;
  for(i = 0; i < MNTHASH; i++) {
          l = tom++;
          for(f = from->mnthash[i]; f; f = f->hash) {
                  rlock(&f->lock);
```

```
137                     mh = smalloc(sizeof(Mhead));
138                     mh->from = f->from;
139                     mh->ref = 1;
140                     incref(mh->from);
141                     *l = mh;
142                     l = &mh->hash;
143                     link = &mh->mount;
144                     for(m = f->mount; m; m = m->next) {
145                             n = smalloc(sizeof(Mount));
146                             n->to = m->to;
147                             incref(n->to);
148                             n->head = mh;
149                             n->flag = m->flag;
150                             if(m->spec[0] != 0)
151                                     strncpy(n->spec, m->spec, NAMELEN);
152                             m->copy = n;
153                             pgrpinsert(&order, m);
154                             *link = n;
155                             link = &n->next;
156                     }
157                     runlock(&f->lock);
158             }
159     }
160     /*
161      * Allocate mount ids in the same sequence as the parent group
162      */
163     lock(&mountid);
164     for(m = order; m; m = m->order)
165             m->copy->mountid = mountid.ref++;
166     unlock(&mountid);
167     wunlock(&from->ns);
168   }
169
170   Fgrp*
171   dupfgrp(Fgrp *f)
172   {
173     Fgrp *new;
174     Chan *c;
175     int i;
176
177     new = smalloc(sizeof(Fgrp));
178     if(f == nil){
179             new->fd = smalloc(DELTAFD*sizeof(Chan*));
180             new->nfd = DELTAFD;
181             new->ref = 1;
182             return new;
183     }
184
185     lock(f);
186     /* Make new fd list shorter if possible, preserving quantization */
187     new->nfd = f->maxfd+1;
188     i = new->nfd%DELTAFD;
189     if(i != 0)
190             new->nfd += DELTAFD - i;
191     new->fd = malloc(new->nfd*sizeof(Chan*));
192     if(new->fd == 0){
193             unlock(f);
194             error("no memory for fgrp");
195     }
196     new->ref = 1;
197
198     new->maxfd = f->maxfd;
199     for(i = 0; i <= f->maxfd; i++) {
200             if(c = f->fd[i]){
201                     incref(c);
202                     new->fd[i] = c;
203             }
204     }
205     unlock(f);
206
207     return new;
208   }
209
210   void
211   closefgrp(Fgrp *f)
212   {
213     int i;
214     Chan *c;
215
216     if(f == 0)
217             return;
218
219     if(decref(f) != 0)
220             return;
221
222     for(i = 0; i <= f->maxfd; i++)
223             if(c = f->fd[i])
224                     cclose(c);
225
226     free(f->fd);
227     free(f);
228   }
229
230   Mount*
231   newmount(Mhead *mh, Chan *to, int flag, char *spec)
232   {
233     Mount *m;
234
235     m = smalloc(sizeof(Mount));
236     m->to = to;
237     m->head = mh;
238     incref(to);
239     m->mountid = incref(&mountid);
240     m->flag = flag;
241     if(spec != 0)
242             strncpy(m->spec, spec, NAMELEN);
243
244     return m;
245   }
246
247   void
248   mountfree(Mount *m)
249   {
250     Mount *f;
251
252     while(m) {
253             f = m->next;
254             cclose(m->to);
255             m->mountid = 0;
256             free(m);
257             m = f;
258     }
259   }
260
261   void
262   resrcwait(char *reason)
263   {
264     char *p;
265
266     if(up == 0)
267             panic("resrcwait");
268
269     p = up->psstate;
270     if(reason) {
271             up->psstate = reason;
272             print("%s\n", reason);
```

```
273     }
274
275     tsleep(&up->sleep, return0, 0, 300);
276     up->psstate = p;
277   }
```

```
typedef struct Alarms     Alarms;
typedef struct Block      Block;
typedef struct Chan       Chan;
typedef struct Cmdbuf     Cmdbuf;
typedef struct Cname      Cname;
typedef struct Crypt      Crypt;
typedef struct Dev        Dev;
typedef struct Dirtab     Dirtab;
typedef struct Egrp       Egrp;
typedef struct Evalue     Evalue;
typedef struct Fgrp       Fgrp;
typedef struct Image      Image;
typedef struct List       List;
typedef struct Log        Log;
typedef struct Logflag    Logflag;
typedef struct Mntcache Mntcache;
typedef struct Mount      Mount;
typedef struct Mntrpc     Mntrpc;
typedef struct Mntwalk    Mntwalk;
typedef struct Mnt        Mnt;
typedef struct Mhead      Mhead;
typedef struct Note       Note;
typedef struct Page       Page;
typedef struct Palloc     Palloc;
typedef struct Pgrps      Pgrps;
typedef struct Pgrp       Pgrp;
typedef struct Physseg    Physseg;
typedef struct Proc       Proc;
typedef struct Pte        Pte;
typedef struct Pthash     Pthash;
typedef struct QLock      QLock;
typedef struct Queue      Queue;
typedef struct Ref        Ref;
typedef struct Rendez     Rendez;
typedef struct Rgrp       Rgrp;
typedef struct RWlock     RWlock;
typedef struct Sargs      Sargs;
typedef struct Segment    Segment;
typedef struct Session    Session;
typedef struct Talarm     Talarm;
typedef struct Target     Target;
typedef struct Waitq      Waitq;
typedef int    Devgen(Chan*, Dirtab*, int, int, Dir*);


#include <auth.h>
#include <fcall.h>

struct Ref
{
  Lock;
  long    ref;
};

struct Rendez
{
  Proc    *p;
};

struct QLock
{
  Lock    use;                  /* to access Qlock structure */
  Proc    *head;                /* next process waiting for object */
  Proc    *tail;                /* last process waiting for object */
  int     locked;               /* flag */
};

struct RWlock
{
  Lock    use;
  Proc    *head;          /* list of waiting processes */
  Proc    *tail;
  ulong   wpc;            /* pc of writer */
  Proc    *wproc;         /* writing proc */
  int     readers;        /* number of readers */
  int     writer;         /* number of writers */
};

struct Talarm
{
  Lock;
  Proc    *list;
};

struct Alarms
{
  QLock;
  Proc    *head;
};

#define MAXSYSARG 5       /* for mount(fd, mpt, flag, arg, srv) */
struct Sargs
{
  ulong   args[MAXSYSARG];
};

/*
 * Access types in namec & channel flags
 */
enum
{
  Aaccess,                      /* as in access, stat */
  Atodir,                       /* as in chdir */
  Aopen,                        /* for i/o */
  Amount,                       /* to be mounted upon */
  Acreate,                      /* file is to be created */

  COPEN   = 0x0001,             /* for i/o */
  CMSG    = 0x0002,             /* the message channel for a mount */
  CCREATE = 0x0004,             /* permits creation if c->mnt */
  CCEXEC  = 0x0008,             /* close on exec */
  CFREE   = 0x0010,             /* not in use */
  CRCLOSE = 0x0020,             /* remove on close */
  CCACHE  = 0x0080,             /* client cache */
};

enum
{
  BINTR   =       (1<<0),
  BFREE   =       (1<<1),
};

struct Block
{
  Block*  next;
  Block*  list;
  uchar*  rp;                   /* first unconsumed byte */
  uchar*  wp;                   /* first empty byte */
  uchar*  lim;                  /* 1 past the end of the buffer */
  uchar*  base;                 /* start of the buffer */
  void    (*free)(Block*);
  ulong   flag;
};
#define BLEN(s)   ((s)->wp - (s)->rp)
#define BALLOC(s) ((s)->lim - (s)->base)
```

```
137  struct Chan
138  {
139    Ref;
140    Chan*   next;                   /* allocation */
141    Chan*   link;
142    vlong   offset;                 /* in file */
143    ushort  type;
144    ulong   dev;
145    ushort  mode;                   /* read/write */
146    ushort  flag;
147    Qid     qid;
148    int     fid;                    /* for devmnt */
149    Mhead*  mh;                     /* mount point that derived Chan */
150    Mhead*  xmh;                    /* Last mount point crossed */
151    int     uri;                    /* union read index */
152    ulong   mountid;
153    Mntcache *mcp;                  /* Mount cache pointer */
154    union {
155            void*   aux;
156            Qid     pgrpid;         /* for #p/notepg */
157            Mnt*    mntptr;         /* for devmnt */
158            ulong   mid;            /* for ns in devproc */
159            char    tag[4];         /* for iproute */
160    };
161    Chan*   mchan;                  /* channel to mounted server */
162    Qid     mqid;                   /* qid of root of mount point */
163    Session*session;
164    Cname   *name;
165  };
166
167  struct Cname
168  {
169    Ref;
170    int     alen;                   /* allocated length */
171    int     len;                    /* strlen(s) */
172    char    *s;
173  };
174
175  struct Dev
176  {
177    int     dc;
178    char*   name;
179
180    void    (*reset)(void);
181    void    (*init)(void);
182    Chan*   (*attach)(char*);
183    Chan*   (*clone)(Chan*, Chan*);
184    int     (*walk)(Chan*, char*);
185    void    (*stat)(Chan*, char*);
186    Chan*   (*open)(Chan*, int);
187    void    (*create)(Chan*, char*, int, ulong);
188    void    (*close)(Chan*);
189    long    (*read)(Chan*, void*, long, vlong);
190    Block*  (*bread)(Chan*, long, ulong);
191    long    (*write)(Chan*, void*, long, vlong);
192    long    (*bwrite)(Chan*, Block*, ulong);
193    void    (*remove)(Chan*);
194    void    (*wstat)(Chan*, char*);
195  };
196
197  struct Dirtab
198  {
199    char    name[NAMELEN];
200    Qid     qid;
201    Length;
202    long    perm;
203  };
204
```

```
205  enum
206  {
207    NSMAX   =       1000,
208    NSLOG   =       7,
209    NSCACHE =       (1<<NSLOG),
210  };
211
212  struct Mntwalk                          /* state for /proc/#/ns */
213  {
214    int             cddone;
215    ulong   id;
216    Mhead*  mh;
217    Mount*  cm;
218  };
219
220  struct Mount
221  {
222    ulong   mountid;
223    Mount*  next;
224    Mhead*  head;
225    Mount*  copy;
226    Mount*  order;
227    Chan*   to;                     /* channel replacing channel */
228    int     flag;
229    char    spec[NAMELEN];
230  };
231
232  struct Mhead
233  {
234    Ref;
235    RWlock  lock;
236    Chan*   from;                   /* channel mounted upon */
237    Mount*  mount;                  /* what's mounted upon it */
238    Mhead*  hash;                   /* Hash chain */
239  };
240
241  struct Mnt
242  {
243    Ref;                    /* Count of attached channels */
244    Chan    *c;             /* Channel to file service */
245    Proc    *rip;           /* Reader in progress */
246    Mntrpc  *queue;         /* Queue of pending requests on this channel */
247    ulong   id;             /* Multiplexer id for channel check */
248    Mnt     *list;          /* Free list */
249    int     flags;          /* cache */
250    int     blocksize;      /* read/write block size */
251    char    *partial;       /* Outstanding read data */
252    int     npart;          /* Sizeof remains */
253  };
254
255  enum
256  {
257    NUser,                          /* note provided externally */
258    NExit,                          /* deliver note quietly */
259    NDebug,                         /* print debug message */
260  };
261
262  struct Note
263  {
264    char    msg[ERRLEN];
265    int     flag;                   /* whether system posted it */
266  };
267
268  enum
269  {
270    PG_NOFLUSH      = 0,
271    PG_TXTFLUSH     = 1,            /* flush dcache and invalidate icache */
272    PG_DATFLUSH     = 2,            /* flush both i & d caches (UNUSED) */
```

```
273     PG_NEWCOL       = 3,            /* page has been recolored */
274
275     PG_MOD          = 0x01,         /* software modified bit */
276     PG_REF          = 0x02,         /* software referenced bit */
277   };
278
279   struct Page
280   {
281     Lock;
282     ulong   pa;                     /* Physical address in memory */
283     ulong   va;                     /* Virtual address for user */
284     ulong   daddr;                  /* Disc address on swap */
285     ushort  ref;                    /* Reference count */
286     char    modref;                 /* Simulated modify/reference bits */
287     char    color;                  /* Cache coloring */
288     char    cachectl[MAXMACH];      /* Cache flushing control for putmmu */
289     Image   *image;                 /* Associated text or swap image */
290     Page    *next;                  /* Lru free list */
291     Page    *prev;
292     Page    *hash;                  /* Image hash chains */
293   };
294
295   struct Swapalloc
296   {
297     Lock;                           /* Free map lock */
298     int     free;                   /* currently free swap pages */
299     uchar*  swmap;                  /* Base of swap map in memory */
300     uchar*  alloc;                  /* Round robin allocator */
301     uchar*  last;                   /* Speed swap allocation */
302     uchar*  top;                    /* Top of swap map */
303     Rendez  r;                      /* Pager kproc idle sleep */
304     Rendez  pause;
305     ulong   highwater;              /* Pager start threshold */
306     ulong   headroom;               /* Space pager frees under highwater */
307   }swapalloc;
308
309   struct Image
310   {
311     Ref;
312     Chan    *c;                     /* channel to text file */
313     Qid     qid;                    /* Qid for page cache coherence */
314     Qid     mqid;
315     Chan    *mchan;
316     ushort  type;                   /* Device type of owning channel */
317     Segment *s;                     /* TEXT segment for image if running */
318     Image   *hash;                  /* Qid hash chains */
319     Image   *next;                  /* Free list */
320     int     notext;                 /* no file associated */
321   };
322
323   struct Pte
324   {
325     Page    *pages[PTEPERTAB];      /* Page map for this chunk of pte */
326     Page    **first;                /* First used entry */
327     Page    **last;                 /* Last used entry */
328     Pte     *next;                  /* Free list */
329   };
330
331   /* Segment types */
332   enum
333   {
334     SG_TYPE         = 07,           /* Mask type of segment */
335     SG_TEXT         = 00,
336     SG_DATA         = 01,
337     SG_BSS          = 02,
338     SG_STACK        = 03,
339     SG_SHARED       = 04,
340     SG_PHYSICAL     = 05,
341     SG_SHDATA       = 06,
342     SG_MAP          = 07,
343
344     SG_RONLY        = 0040,         /* Segment is read only */
345     SG_CEXEC        = 0100,         /* Detach at exec */
346   };
347
348   #define PG_ONSWAP 1
349   #define onswap(s) (((ulong)s)&PG_ONSWAP)
350   #define pagedout(s)       (((ulong)s)==0 || onswap(s))
351   #define swapaddr(s)       (((ulong)s)&~PG_ONSWAP)
352
353   #define SEGMAXSIZE        (SEGMAPSIZE*PTEMAPMEM)
354
355   struct Physseg
356   {
357     ulong   attr;                   /* Segment attributes */
358     char    *name;                  /* Attach name */
359     ulong   pa;                     /* Physical address */
360     ulong   size;                   /* Maximum segment size in pages */
361     Page    *(*pgalloc)(Segment*, ulong);   /* Allocation if we need it */
362     void    (*pgfree)(Page*);
363   };
364
365   struct Segment
366   {
367     Ref;
368     QLock   lk;
369     ushort  steal;          /* Page stealer lock */
370     ushort  type;           /* segment type */
371     ulong   base;           /* virtual base */
372     ulong   top;            /* virtual top */
373     ulong   size;           /* size in pages */
374     ulong   fstart;         /* start address in file for demand load */
375     ulong   flen;           /* length of segment in file */
376     int     flushme;        /* maintain icache for this segment */
377     Image   *image;         /* text in file attached to this segment */
378     Physseg *pseg;
379     ulong*  profile;        /* Tick profile area */
380     Pte     **map;
381     int     mapsize;
382     Pte     *ssegmap[SSEGMAPSIZE];
383   };
384
385   enum
386   {
387     RENDHASH =      32,             /* Hash to lookup rendezvous tags */
388     MNTHASH =       32,             /* Hash to walk mount table */
389     NFD =           100,            /* per process file descriptors */
390     PGHLOG  =       9,
391     PGHSIZE =       1<<PGHLOG,      /* Page hash for image lookup */
392   };
393   #define REND(p,s) ((p)->rendhash[(s)%RENDHASH])
394   #define MOUNTH(p,s)       ((p)->mnthash[(s)->qid.path%MNTHASH])
395
396   struct Pgrp
397   {
398     Ref;                            /* also used as a lock when mounting */
399     int     noattach;
400     ulong   pgrpid;
401     QLock   debug;                  /* single access via devproc.c */
402     RWlock  ns;                     /* Namespace n read/one write lock */
403     QLock   nsh;
404     Mhead   *mnthash[MNTHASH];
405   };
406
407   struct Rgrp
408   {
```

```
409     Ref;
410     Proc    *rendhash[RENDHASH];    /* Rendezvous tag hash */
411   };
412
413   struct Egrp
414   {
415     Ref;
416     QLock;
417     Evalue  *entries;
418     ulong   path;   /* qid.path of next Evalue to be allocated */
419     ulong   vers;   /* of Egrp */
420   };
421
422   struct Evalue
423   {
424     char    *name;
425     char    *value;
426     int     len;
427     Evalue  *link;
428     Qid     qid;
429   };
430
431   struct Fgrp
432   {
433     Ref;
434     Chan    **fd;
435     int     nfd;                    /* number allocated */
436     int     maxfd;                  /* highest fd in use */
437   };
438
439   enum
440   {
441     DELTAFD = 20            /* incremental increase in Fgrp.fd's */
442   };
443
444   struct Palloc
445   {
446     Lock;
447     ulong   p0, p1;                 /* base of pages in bank 0/1 */
448     ulong   np0, np1;               /* number of pages in bank 0/1 */
449     Page    *head;                  /* most recently used */
450     Page    *tail;                  /* least recently used */
451     ulong   freecount;              /* how many pages on free list now */
452     ulong   user;                   /* how many user pages */
453     Page    *hash[PGHSIZE];
454     Lock    hashlock;
455     Rendez  r;                      /* Sleep for free mem */
456     QLock   pwait;                  /* Queue of procs waiting for memory */
457     ulong   cmembase;               /* Key memory */
458     ulong   cmemtop;
459   };
460
461   struct Waitq
462   {
463     Waitmsg w;
464     Waitq   *next;
465   };
466
467   enum
468   {
469     RFNAMEG         = (1<<0),
470     RFENVG          = (1<<1),
471     RFFDG           = (1<<2),
472     RFNOTEG         = (1<<3),
473     RFPROC          = (1<<4),
474     RFMEM           = (1<<5),
475     RFNOWAIT        = (1<<6),
476     RFCNAMEG        = (1<<10),
477     RFCENVG         = (1<<11),
478     RFCFDG          = (1<<12),
479     RFREND          = (1<<13),
480     RFNOMNT         = (1<<14),
481   };
482
483   /*
484    *  process memory segments - NSEG always last !
485    */
486   enum
487   {
488     SSEG, TSEG, DSEG, BSEG, ESEG, LSEG, SEG1, SEG2, SEG3, SEG4, NSEG
489   };
490
491   enum
492   {
493     Dead = 0,               /* Process states */
494     Moribund,
495     Ready,
496     Scheding,
497     Running,
498     Queueing,
499     QueueingR,
500     QueueingW,
501     Wakeme,
502     Broken,
503     Stopped,
504     Rendezvous,
505
506     Proc_stopme = 1,        /* devproc requests */
507     Proc_exitme,
508     Proc_traceme,
509     Proc_exitbig,
510
511     TUser = 0,              /* Proc.time */
512     TSys,
513     TReal,
514     TCUser,
515     TCSys,
516     TCReal,
517
518     NERR = 15,
519     NNOTE = 5,
520
521     Nrq             = 20,   /* number of scheduler priority levels */
522     PriLock         = 0,    /* priority for processes waiting on a lock */
523     PriNormal       = 10,   /* base priority for normal processes */
524     PriKproc        = 13,   /* base priority for kernel processes */
525     PriRoot         = 13,   /* base priority for root processes */
526   };
527
528   struct Proc
529   {
530     Label   sched;          /* known to l.s */
531     char    *kstack;        /* known to l.s */
532     Mach    *mach;          /* machine running this proc */
533     char    text[NAMELEN];
534     char    user[NAMELEN];
535     Proc    *rnext;         /* next process in run queue */
536     Proc    *qnext;         /* next process on queue for a QLock */
537     QLock   *qlock;         /* addrof qlock being queued for DEBUG */
538     int     state;
539     char    *psstate;       /* What /proc/#/status reports */
540     Segment *seg[NSEG];
541     QLock   seglock;        /* locked whenever seg[] changes */
542     ulong   pid;
543     ulong   noteid;         /* Equivalent of note group */
544
```

```
545     Lock    exl;            /* Lock count and waitq */
546     Waitq   *waitq;         /* Exited processes wait children */
547     int     nchild;         /* Number of living children */
548     int     nwait;          /* Number of uncollected wait records */
549     QLock   qwaitr;
550     Rendez  waitr;          /* Place to hang out in wait */
551     Proc    *parent;
552
553     Pgrp    *pgrp;          /* Process group for namespace */
554     Egrp    *egrp;          /* Environment group */
555     Fgrp    *fgrp;          /* File descriptor group */
556     Rgrp    *rgrp;          /* Rendez group */
557
558     ulong   parentpid;
559     ulong   time[6];        /* User, Sys, Real; child U, S, R */
560     short   insyscall;
561     int     fpstate;
562
563     QLock   debug;          /* to access debugging elements of User */
564     Proc    *pdbg;          /* the debugging process */
565     ulong   procmode;       /* proc device file mode */
566     int     hang;           /* hang at next exec for debug */
567     int     procctl;        /* Control for /proc debugging */
568     ulong   pc;             /* DEBUG only */
569
570     Lock    rlock;          /* sync sleep/wakeup with postnote */
571     Rendez  *r;             /* rendezvous point slept on */
572     Rendez  sleep;          /* place for syssleep/debug */
573     int     notepending;    /* note issued but not acted on */
574     int     kp;             /* true if a kernel process */
575     Proc    *palarm;        /* Next alarm time */
576     ulong   alarm;          /* Time of call */
577     int     newtlb;         /* Pager has changed my pte's, I must flush */
578
579     ulong   rendtag;        /* Tag for rendezvous */
580     ulong   rendval;        /* Value for rendezvous */
581     Proc    *rendhash;      /* Hash list for tag values */
582
583     ulong   twhen;
584     Rendez  *trend;
585     Proc    *tlink;
586     int     (*tfn)(void*);
587     void    (*kpfun)(void*);
588     void    *kparg;
589
590     FPsave  fpsave;         /* address of this is known by db */
591     int     scallnr;        /* sys call number - known by db */
592     Sargs   s;              /* address of this is known by db */
593     int     nerrlab;
594     Label   errlab[NERR];
595     char    error[ERRLEN];
596     char    elem[NAMELEN];  /* last name element from namec */
597     Chan    *slash;
598     Chan    *dot;
599
600     Note    note[NNOTE];
601     short   nnote;
602     short   notified;       /* sysnoted is due */
603     Note    lastnote;
604     int     (*notify)(void*, char*);
605
606     int     lockwait;       /* waiting for lock to be released */
607
608     Mach    *wired;
609     Mach    *mp;            /* machine this process last ran on */
610     ulong   priority;       /* priority level */
611     ulong   basepri;        /* base priority level */
612     ulong   rt;             /* # ticks used since last blocked */
613     ulong   art;            /* avg # ticks used since last blocked */
614     ulong   movetime;       /* last time process switched processors */
615     ulong   readytime;      /* time process went ready */
616     int     preempted;      /* true if this process hasn't finished the interrupt
617                                *  that last preempted it
618                                */
619     ulong   qpc;            /* pc calling last blocking qlock */
620
621     void    *ureg;          /* User registers for notes */
622     void    *dbgreg;        /* User registers for devproc */
623     Notsave;
624
625     /*
626      *  machine specific MMU
627      */
628     PMMU;
629   };
630
631   enum
632   {
633     PRINTSIZE =     256,
634     MAXCRYPT =      127,
635     NUMSIZE =       12,             /* size of formatted number */
636     MB =            (1024*1024),
637     READSTR =       1000,           /* temporary buffer size for device reads */
638   };
639
640   extern    Conf    conf;
641   extern    char*   conffile;
642   extern    int     cpuserver;
643   extern    Dev*    devtab[];
644   extern  char      eve[];
645   extern    char    hostdomain[];
646   extern    uchar   initcode[];
647   extern    FPsave  initfp;
648   extern  Queue*    kbdq;
649   extern  Ref       noteidalloc;
650   extern    int     nrdy;
651   extern    Palloc  palloc;
652   extern  Queue     *printq;
653   extern    char*   statename[];
654   extern  Image     swapimage;
655   extern    char    sysname[NAMELEN];
656   extern    Pthash  syspt;
657   extern    Talarm  talarm;
658
659   enum
660   {
661     CHDIR =         0x80000000L,
662     CHAPPEND =      0x40000000L,
663     CHEXCL =        0x20000000L,
664     CHMOUNT =       0x10000000L,
665   };
666
667   /*
668    * auth messages
669    */
670   enum
671   {
672     FScchal = 1,
673     FSschal,
674     FSok,
675     FSctick,
676     FSstick,
677     FSerr,
678
679     RXschal = 0,
680     RXstick = 1,
```

```
681
682     AUTHLEN = 8,
683   };
684
685   enum
686   {
687     LRESPROF        = 3,
688   };
689
690   /*
691    *  action log
692    */
693   struct Log {
694     Lock;
695     int     opens;
696     char*   buf;
697     char    *end;
698     char    *rptr;
699     int     len;
700
701     int     logmask;        /* mask of things to debug */
702
703     QLock   readq;
704     Rendez  readr;
705   };
706
707   struct Logflag {
708     char*   name;
709     int     mask;
710   };
711
712   struct Cmdbuf
713   {
714     char    buf[256];
715     char    *f[16];
716     int     nf;
717   };
718
719   extern int nsyscall;
720
721   #define DEVDOTDOT -1
722
723   #pragma   varargck        argpos  print   1
724   #pragma   varargck        argpos  snprint 3
725   #pragma   varargck        argpos  sprint  2
726   #pragma   varargck        argpos  fprint  2
727
728   #pragma   varargck        type    "lld"   vlong
729   #pragma   varargck        type    "llx"   vlong
730   #pragma   varargck        type    "lld"   uvlong
731   #pragma   varargck        type    "llx"   uvlong
732   #pragma   varargck        type    "lx"    void*
733   #pragma   varargck        type    "ld"    long
734   #pragma   varargck        type    "lx"    long
735   #pragma   varargck        type    "ld"    ulong
736   #pragma   varargck        type    "lx"    ulong
737   #pragma   varargck        type    "d"     int
738   #pragma   varargck        type    "x"     int
739   #pragma   varargck        type    "c"     int
740   #pragma   varargck        type    "C"     int
741   #pragma   varargck        type    "d"     uint
742   #pragma   varargck        type    "x"     uint
743   #pragma   varargck        type    "c"     uint
744   #pragma   varargck        type    "C"     uint
745   #pragma   varargck        type    "f"     double
746   #pragma   varargck        type    "e"     double
747   #pragma   varargck        type    "g"     double
748   #pragma   varargck        type    "s"     char*
749   #pragma   varargck        type    "S"     Rune*
750   #pragma   varargck        type    "r"     void
751   #pragma   varargck        type    "%"     void
752   #pragma   varargck        type    "I"     uchar*
753   #pragma   varargck        type    "V"     uchar*
754   #pragma   varargck        type    "E"     uchar*
755   #pragma   varargck        type    "M"     uchar*
756   #pragma   varargck        type    "p"     void*
```

```
1     void              accounttime(void);
2     void              addclock0link(void (*)(void));
3     int               addphysseg(Physseg*);
4     void              addrootfile(char*, uchar*, ulong);
5     Block*            adjustblock(Block*, int);
6     void              alarmkproc(void*);
7     Block*            allocb(int);
8     int               anyhigher(void);
9     int               anyready(void);
10    #define   assert(x)       if(x);else panic("assert(x) failed");
11    void              _assert(char*);
12    Image*            attachimage(int, Chan*, ulong, ulong);
13    long              authcheck(Chan*, char*, int);
14    long              authcheckread(Chan*, char*, int n);
15    void              authclose(Chan*);
16    long              authentread(Chan*, char*, int);
17    long              authentwrite(Chan*, char*, int);
18    long              authread(Chan*, char*, int);
19    void              authreply(Session*, ulong, Fcall*);
20    ulong             authrequest(Session*, Fcall*);
21    long              authwrite(Chan*, char*, int);
22    Page*             auxpage(void);
23    int               blocklen(Block*);
24    void              cachedel(Image*, ulong);
25    void              cachepage(Page*, Image*);
26    void              callwithureg(void(*)(Ureg*));
27    int               cangetc(void*);
28    int               canlock(Lock*);
29    int               canpage(Proc*);
30    int               canputc(void*);
31    int               canqlock(QLock*);
32    int               canrlock(RWlock*);
33    void              chandevinit(void);
34    void              chandevreset(void);
35    void              chanfree(Chan*);
36    void              chanrec(Mnt*);
37    void              checkalarms(void);
38    void              checkb(Block*, char*);
39    void              cinit(void);
40    Chan*             cclone(Chan*, Chan*);
41    void              cclose(Chan*);
42    void              closeegrp(Egrp*);
43    void              closefgrp(Fgrp*);
44    void              closemount(Mount*);
45    void              closepgrp(Pgrp*);
46    void              closergrp(Rgrp*);
47    long              clrfpintr(void);
48    int               cmount(Chan*, Chan*, int, char*);
49    void              cnameclose(Cname*);
50    void              confinit(void);
51    void              confinit1(int);
52    int               consactive(void);
53    void              (*consdebug)(void);
54    void              copen(Chan*);
55    Block*            concatblock(Block*);
56    Block*            copyblock(Block*, int);
57    void              copypage(Page*, Page*);
58    int               cread(Chan*, uchar*, int, vlong);
59    void              cunmount(Chan*, Chan*);
60    void              cupdate(Chan*, uchar*, int, vlong);
61    void              cwrite(Chan*, uchar*, int, vlong);
62    ulong             dbgpc(Proc*);
63    int               decref(Ref*);
64    int               decrypt(void*, void*, int);
65    void              delay(int);
66    Chan*             devattach(int, char*);
67    Block*            devbread(Chan*, long, ulong);
68    long              devbwrite(Chan*, Block*, ulong);
69    Chan*             devclone(Chan*, Chan*);
70    void              devcreate(Chan*, char*, int, ulong);
71    void              devdir(Chan*, Qid, char*, vlong, char*, long, Dir*);
72    long              devdirread(Chan*, char*, long, Dirtab*, int, Devgen*);
73    Devgen            devgen;
74    void              devinit(void);
75    int               devno(int, int);
76    Chan*             devopen(Chan*, int, Dirtab*, int, Devgen*);
77    void              devremove(Chan*);
78    void              devreset(void);
79    void              devstat(Chan*, char*, Dirtab*, int, Devgen*);
80    int               devwalk(Chan*, char*, Dirtab*, int, Devgen*);
81    void              devwstat(Chan*, char*);
82    Chan*             domount(Chan*);
83    void              drawactive(int);
84    void              drawcmap(void);
85    void              dumpaproc(Proc*);
86    void              dumpqueues(void);
87    void              dumpregs(Ureg*);
88    void              dumpstack(void);
89    Fgrp*             dupfgrp(Fgrp*);
90    void              duppage(Page*);
91    void              dupswap(Page*);
92    int               encrypt(void*, void*, int);
93    void              envcpy(Egrp*, Egrp*);
94    int               eqchan(Chan*, Chan*, int);
95    int               eqqid(Qid, Qid);
96    void              error(char*);
97    long              execregs(ulong, ulong, ulong);
98    void              exhausted(char*);
99    void              exit(int);
100   vlong             fastticks(uvlong*);
101   int               fault(ulong, int);
102   void              fdclose(int, int);
103   Chan*             fdtochan(int, int, int, int);
104   int               fixfault(Segment*, ulong, int, int);
105   void              flushmmu(void);
106   void              forkchild(Proc*, Ureg*);
107   void              forkret(void);
108   void              free(void*);
109   void              freeb(Block*);
110   void              freeblist(Block*);
111   int               freebroken(void);
112   void              freechan(Chan*);
113   void              freepte(Segment*, Pte*);
114   void              freesegs(int);
115   void              freesession(Session*);
116   void              getcolor(ulong, ulong*, ulong*, ulong*);
117   ulong             getmalloctag(void*);
118   ulong             getrealloctag(void*);
119   void              gotolabel(Label*);
120   int               haswaitq(void*);
121   long              hostdomainwrite(char*, int);
122   long              hostownerwrite(char*, int);
123   void              iallocinit(void);
124   Block*            iallocb(int);
125   void              iallocsummary(void);
126   long              ibrk(ulong, int);
127   void              ilock(Lock*);
128   void              iunlock(Lock*);
129   int               incref(Ref*);
130   void              initseg(void);
131   void              isdir(Chan*);
132   int               iseve(void);
133   int               islo(void);
134   int               ispages(void*);
135   void              ixsummary(void);
136   void              kbdclock(void);
```

```
int             kbdcr2nl(Queue*, int);
int             kbdputc(Queue*, int);
void            kbdrepeat(int);
long            keyread(char*, int, long);
long            keywrite(char*, int);
void            kickpager(void);
void            killbig(void);
int             kprint(char*, ...);
void            kproc(char*, void(*)(void*), void*);
void            kprocchild(Proc*, void (*)(void*), void*);
void            (*kproftimer)(ulong);
void            ksetenv(char*, char*);
long            latin1(Rune*, int);
void            lock(Lock*);
void            lockinit(void);
void            logopen(Log*);
void            logclose(Log*);
char*           logctl(Log*, int, char**, Logflag*);
long            logread(Log*, void*, ulong, long);
void            log(Log*, int, char*, ...);
Page*           lookpage(Image*, ulong);
void            machinit(void);
void*           mallocz(ulong, int);
void*           malloc(ulong);
void            mallocsummary(void);
void            mfreeseg(Segment*, ulong, int);
void            microdelay(int);
void            mmurelease(Proc*);
void            mmuswitch(Proc*);
void            mntdump(void);
long            mntread9p(Chan*, void*, long, vlong);
long            mntwrite9p(Chan*, void*, long, vlong);
void            mountfree(Mount*);
ulong           msize(void*);
Chan*           namec(char*, int, int, ulong);
void            nameok(char*, int);
#define         nelem(x)        (sizeof(x)/sizeof(x[0]))
Chan*           newchan(void);
Mount*          newmount(Mhead*, Chan*, int, char*);
Page*           newpage(int, Segment **, ulong);
Pgrp*           newpgrp(void);
Rgrp*           newrgrp(void);
Proc*           newproc(void);
char*           nextelem(char*, char*);
void            nexterror(void);
Cname*          newcname(char*);
int             notify(Ureg*);
int             nrand(int);
int             okaddr(ulong, ulong, int);
int             openmode(ulong);
Block*          packblock(Block*);
Block*          padblock(Block*, int);
void            pagechainhead(Page*);
void            pageinit(void);
void            pagersummary(void);
void            panic(char*, ...);
Cmdbuf*         parsecmd(char *a, int n);
void            pexit(char*, int);
void            pgrpcpy(Pgrp*, Pgrp*);
void            pgrpnote(ulong, char*, long, int);
Pgrp*           pgrptab(int);
void            pio(Segment *, ulong, ulong, Page **);
#define         poperror()              up->nerrlab--
int             postnote(Proc*, int, char*, int);
int             pprint(char*, ...);
void            printinit(void);
ulong           procalarm(ulong);
int             proccounter(char *name);
void            procctl(Proc*);
void            procdump(void);
void            procinit0(void);
void            procflushseg(Segment*);
Proc*           proctab(int);
void            procwired(Proc*, int);
Pte*            ptealloc(void);
Pte*            ptecpy(Pte*);
int             pullblock(Block**, int);
Block*          pullupblock(Block*, int);
void            putimage(Image*);
void            putmhead(Mhead*);
void            putmmu(ulong, ulong, Page*);
void            putpage(Page*);
void            putseg(Segment*);
void            putstr(char*);
void            putstr(char*);
void            putstrn(char*, int);
void            putswap(Page*);
ulong           pwait(Waitmsg*);
Block*          qbread(Queue*, int);
long            qbwrite(Queue*, Block*);
int             qcanread(Queue*);
void            qclose(Queue*);
int             qconsume(Queue*, void*, int);
Block*          qcopy(Queue*, int, ulong);
void            qdiscard(Queue*, int);
void            qflush(Queue*);
int             qfull(Queue*);
Block*          qget(Queue*);
void            qhangup(Queue*, char*);
int             qisclosed(Queue*);
void            qinit(void);
int             qiwrite(Queue*, void*, int);
int             qlen(Queue*);
void            qlock(QLock*);
Queue*          qopen(int, int, void (*)(void*), void*);
int             qpass(Queue*, Block*);
int             qpassnolim(Queue*, Block*);
int             qproduce(Queue*, void*, int);
long            qread(Queue*, void*, int);
void            qreopen(Queue*);
void            qunlock(QLock*);
int             qwindow(Queue*);
int             qwrite(Queue*, void*, int);
void            qsetlimit(Queue*, int);
void            qnoblock(Queue*, int);
int             rand(void);
void            rdb(void);
int             readnum(ulong, char*, ulong, ulong, int);
int             readstr(ulong, char*, ulong, char*);
void            ready(Proc*);
void            relocateseg(Segment*, ulong);
void            renameuser(char*, char*);
void            resched(char*);
void            resrcwait(char*);
int             return0(void*);
void            rlock(RWlock*);
void            rootreq(Chan*, Mnt*);
long            rtctime(void);
void            runlock(RWlock*);
Proc*           runproc(void);
void            savefpregs(FPsave*);
void            (*saveintrts)(void);
void            sched(void);
void            scheddump(void);
void            schedinit(void);
int             screenbits(void);
```

```
273   long              seconds(void);
274   ulong             segattach(Proc*, ulong, char *, ulong, ulong);
275   void              segclock(ulong);
276   void              segpage(Segment*, Page*);
277   int               serialgetc(void);
278   void              serialputs(char*, int);
279   int               setcolor(ulong, ulong, ulong, ulong);
280   void              setkernur(Ureg*, Proc*);
281   int               setlabel(Label*);
282   void              setmalloctag(void*, ulong);
283   void              setrealloctag(void*, ulong);
284   void              setregisters(Ureg*, char*, char*, int);
285   void              setswapchan(Chan*);
286   char*             skipslash(char*);
287   void              sleep(Rendez*, int(*)(void*), void*);
288   void*             smalloc(ulong);
289   int               splhi(void);
290   int               spllo(void);
291   void              splx(int);
292   void              splxpc(int);
293   char*     srvname(Chan*);
294   int               swapcount(ulong);
295   int               swapfull(void);
296   void              swapinit(void);
297   vlong             todget(vlong*);
298   void              todfix(void);
299   void              todsetfreq(vlong);
300   void              todinit(void);
301   void              todset(vlong, vlong, int);
302   Block*            trimblock(Block*, int, int);
303   void              tsleep(Rendez*, int (*)(void*), void*, int);
304   void              unbreak(Proc*);
305   void              uncachepage(Page*);
306   long              unionread(Chan*, void*, long);
307   void              unlock(Lock*);
308   Chan*             undomount(Chan*);
309   void              userinit(void);
310   ulong             userpc(void);
311   long              userwrite(char*, int);
312   void              validaddr(ulong, ulong, int);
313   void              vcacheinval(Page*, ulong);
314   void*             vmemchr(void*, int, int);
315   int               wakeup(Rendez*);
316   int               walk(Chan**, char*, int);
317   int               walkname(Chan**, char*, int);
318   void              wlock(RWlock*);
319   void              wunlock(RWlock*);
320   void*             xalloc(ulong);
321   void*             xallocz(ulong, int);
322   void              xfree(void*);
323   void              xhole(ulong, ulong);
324   void              xinit(void);
325   int               xmerge(void*, void*);
326   void*             xspanalloc(ulong, int, ulong);
327   void              xsummary(void);
328   Segment*  data2txt(Segment*);
329   Segment*  dupseg(Segment**, int, int);
330   Segment*  newseg(int, ulong, ulong);
331   Segment*  seg(Proc*, ulong, int);
332   void              hnputv(void*, vlong);
333   void              hnputl(void*, ulong);
334   void              hnputs(void*, ushort);
335   vlong             nhgetv(void*);
336   ulong             nhgetl(void*);
337   ushort            nhgets(void*);
```

```
#include  "u.h"
#include  "../port/lib.h"
#include  "mem.h"
#include  "dat.h"
#include  "fns.h"
#include  "../port/error.h"

Ref       pidalloc;
Ref       noteidalloc;

struct
{
  Lock;
  Proc*   arena;
  Proc*   free;
}procalloc;

struct
{
  Lock;
  Waitq*  free;
}waitqalloc;

typedef struct
{
  Lock;
  Proc*   head;
  Proc*   tail;
  int     n;
} Schedq;

int       nrdy;
Schedq    runq[Nrq];

char *statename[] =
{                 /* BUG: generate automatically */
  "Dead",
  "Moribund",
  "Ready",
  "Scheding",
  "Running",
  "Queueing",
  "QueueingR",
  "QueueingW",
  "Wakeme",
  "Broken",
  "Stopped",
  "Rendez",
};

/*
 * Always splhi()'ed.
 */
void
schedinit(void)           /* never returns */
{
  setlabel(&m->sched);
  if(up) {
          m->proc = 0;
          switch(up->state) {
          case Running:
                  ready(up);
                  break;
          case Moribund:
                  up->state = Dead;
                  /*
                   * Holding locks from pexit:
                   *      procalloc
                   *      palloc
                   */
                  mmurelease(up);

                  up->qnext = procalloc.free;
                  procalloc.free = up;

                  unlock(&palloc);
                  unlock(&procalloc);
                  break;
          }
          up->mach = 0;
          up = 0;
  }
  sched();
}

/*
 *  If changing this routine, look also at sleep().  It
 *  contains a copy of the guts of sched().
 */
void
sched(void)
{
  if(up) {
          splhi();

          /* statistics */
          m->cs++;

          procsave(up);
          if(setlabel(&up->sched)) {
                  procrestore(up);
                  spllo();
                  return;
          }
          gotolabel(&m->sched);
  }
  up = runproc();
  up->state = Running;
  up->mach = MACHP(m->machno);
  m->proc = up;
  mmuswitch(up);
  gotolabel(&up->sched);
}

int
anyready(void)
{
  return nrdy;
}

int
anyhigher(void)
{
  Schedq *rq;

  if(nrdy == 0)
          return 0;

  for(rq = &runq[Nrq-1]; rq > &runq[up->priority]; rq--)
          if(rq->head != nil)
                  return 1;

  return 0;
}

enum
```

```
{
  Squantum = (HZ+Nrq-1)/Nrq,
};

void
ready(Proc *p)
{
  int s, pri;
  Schedq *rq;

  s = splhi();

  /* history counts */
  if(p->state == Running){
          p->rt++;
          pri = ((p->art + (p->rt<<1))>>2)/Squantum;
  } else {
          p->art = (p->art + (p->rt<<1))>>2;
          p->rt = 0;
          pri = p->art/Squantum;
  }
  pri = p->basepri - pri;
  if(pri < 0)
          pri = 0;

  /* the only intersection between the classes is at PriNormal */
  if(pri < PriNormal && p->basepri > PriNormal)
          pri = PriNormal;

  /* stick at low priority any process waiting for a lock */
  if(p->lockwait)
          pri = PriLock;

  p->priority = pri;
  rq = &runq[p->priority];

  lock(runq);
  p->rnext = 0;
  if(rq->tail)
          rq->tail->rnext = p;
  else
          rq->head = p;
  rq->tail = p;
  rq->n++;
  nrdy++;
  p->readytime = m->ticks;
  p->state = Ready;
  unlock(runq);
  splx(s);
}

Proc*
runproc(void)
{
  Schedq *rq, *xrq;
  Proc *p, *l;
  ulong rt;

loop:

  /*
   *  find a process that last ran on this processor (affinity),
   *  or one that hasn't moved in a while (load balancing).
   */
  spllo();
  for(;;){
          idlehands();
          if((++(m->fairness) & 0x3) == 0){
                  /*
                   *  once in a while, run process that's been waiting longest
                   *  regardless of movetime
                   */
                  rt = 0xffffffff;
                  xrq = nil;
                  for(rq = runq; rq < &runq[Nrq]; rq++){
                          p = rq->head;
                          if(p == 0)
                                  continue;
                          if(p->readytime < rt){
                                  xrq = rq;
                                  rt = p->readytime;
                          }
                  }
                  if(xrq != nil){
                          rq = xrq;
                          p = rq->head;
                          if(p != nil && p->wired == nil)
                                  p->movetime = 0;
                          goto found;
                  }
          } else {
                  /*
                   *  get highest priority process that this
                   *  processor can run given affinity constraints
                   */
                  for(rq = &runq[Nrq-1]; rq >= runq; rq--){
                          p = rq->head;
                          if(p == 0)
                                  continue;
                          for(; p; p = p->rnext){
                                  if(p->mp == MACHP(m->machno)
                                  || p->movetime < MACHP(0)->ticks)
                                          goto found;
                          }
                  }
          }
  }

found:
  splhi();
  if(!canlock(runq))
          goto loop;

  l = 0;
  for(p = rq->head; p; p = p->rnext){
          if(p->mp == MACHP(m->machno) || p->movetime < MACHP(0)->ticks)
                  break;
          l = p;
  }

  /*
   *  p->mach==0 only when process state is saved
   */
  if(p == 0 || p->mach){
          unlock(runq);
          goto loop;
  }
  if(p->rnext == 0)
          rq->tail = l;
  if(l)
          l->rnext = p->rnext;
  else
          rq->head = p->rnext;
  rq->n--;
  nrdy--;
  if(p->state != Ready)
```

```
273             print("runproc %s %lud %s\n", p->text, p->pid, statename[p->state]);
274     unlock(runq);
275
276     p->state = Scheding;
277     if(p->mp != MACHP(m->machno))
278             p->movetime = MACHP(0)->ticks + HZ/10;
279     p->mp = MACHP(m->machno);
280     return p;
281   }
282
283   int
284   canpage(Proc *p)
285   {
286     int ok = 0;
287
288     splhi();
289     lock(runq);
290     /* Only reliable way to see if we are Running */
291     if(p->mach == 0) {
292             p->newtlb = 1;
293             ok = 1;
294     }
295     unlock(runq);
296     spllo();
297
298     return ok;
299   }
300
301   Proc*
302   newproc(void)
303   {
304     Proc *p;
305
306     lock(&procalloc);
307     for(;;) {
308             if(p = procalloc.free)
309                     break;
310
311             unlock(&procalloc);
312             resrcwait("no procs");
313             lock(&procalloc);
314     }
315     procalloc.free = p->qnext;
316     unlock(&procalloc);
317
318     p->state = Scheding;
319     p->psstate = "New";
320     p->mach = 0;
321     p->qnext = 0;
322     p->nchild = 0;
323     p->nwait = 0;
324     p->waitq = 0;
325     p->pgrp = 0;
326     p->egrp = 0;
327     p->fgrp = 0;
328     p->rgrp = 0;
329     p->pdbg = 0;
330     p->fpstate = FPinit;
331     p->kp = 0;
332     p->procctl = 0;
333     p->notepending = 0;
334     p->mp = 0;
335     p->movetime = 0;
336     p->wired = 0;
337     p->ureg = 0;
338     p->error[0] = '\0';
339     memset(p->seg, 0, sizeof p->seg);
340     p->pid = incref(&pidalloc);
341     p->noteid = incref(&noteidalloc);
342     if(p->pid==0 || p->noteid==0)
343             panic("pidalloc");
344     if(p->kstack == 0)
345             p->kstack = smalloc(KSTACK);
346
347     return p;
348   }
349
350   /*
351    * wire this proc to a machine
352    */
353   void
354   procwired(Proc *p, int bm)
355   {
356     Proc *pp;
357     int i;
358     char nwired[MAXMACH];
359     Mach *wm;
360
361     if(bm < 0){
362             /* pick a machine to wire to */
363             memset(nwired, 0, sizeof(nwired));
364             p->wired = 0;
365             pp = proctab(0);
366             for(i=0; i<conf.nproc; i++, pp++){
367                     wm = pp->wired;
368                     if(wm && pp->pid)
369                             nwired[wm->machno]++;
370             }
371             bm = 0;
372             for(i=0; i<conf.nmach; i++)
373                     if(nwired[i] < nwired[bm])
374                             bm = i;
375     } else {
376             /* use the virtual machine requested */
377             bm = bm % conf.nmach;
378     }
379
380     p->wired = MACHP(bm);
381     p->movetime = 0xffffffff;
382     p->mp = p->wired;
383   }
384
385   void
386   procinit0(void)           /* bad planning - clashes with devproc.c */
387   {
388     Proc *p;
389     int i;
390
391     procalloc.free = xalloc(conf.nproc*sizeof(Proc));
392     if(procalloc.free == nil)
393             panic("cannot allocate %d procs\n", conf.nproc);
394     procalloc.arena = procalloc.free;
395
396     p = procalloc.free;
397     for(i=0; i<conf.nproc-1; i++,p++)
398             p->qnext = p+1;
399     p->qnext = 0;
400   }
401
402
403   /*
404    *  Sleep, postnote, and wakeup are complicated by the
405    *  fact that at least one of them must indirect
406    *  through an unlocked structure to find the synchronizing
407    *  lock structure.  This is because sleep()
408    *  and wakeup() share direct knowledge only of r while
```

```
409     *  sleep() and postnote() share knowledge only of p.  We've
410     *  chosen to put the synchronization lock in p, i.e.,
411     *  p->rlock.  Therefore the interaction between sleep()
412     *  and postnote() is completely synchronized by keeping
413     *  p->rlock locked in sleep until the process has
414     *  saved all the information it needs to become dormant.
415     *
416     *  However, wakeup() can only find what process is sleeping
417     *  by looking at r->p.  A wakeup looks like:
418     *
419     *        1) set condition that sleep checks with (*f)()
420     *        2) is p = r->p non zero
421     *        3) lock(p->rlock)
422     *        4) check r->p == p
423     *        5) ...
424     *
425     *  A sleep looks like
426     *
427     *        a) lock(p->rlock)
428     *        b) r->p = up
429     *        c) check condition
430     *        d) ...
431     *
432     *  On a multiprocessor, two processors
433     *  may not see writes occur in the same order.  The coherence()
434     *  instruction ensures that a processor has flushed all its
435     *  writes to memory so that those writes will be seen by other
436     *  processors and that the processor will see all writes flushed
437     *  by other processors.
438     *
439     *  To make the above sequence work on a multiprocessor, we need
440     *  to put a coherence() call between (1) and (2) and between
441     *  (b) and (c).  That way we're guaranteed that if (1) and
442     *  (2) occur after (c), the wakeup process will know
443     *  which process is about to sleep and will enter its
444     *  critical section.  If it doesn't, the sleep could proceed
445     *  while the waker returns without doing anything.
446     *  Similarly, if (b) and (c) occur after (2),
447     *  the sleeper needs coherence to see that the condition was
448     *  set.  Otherwise it could sleep even though the wakeup
449     *  had already decided there was nothing to do.
450     *
451     *        jmk & presotto
452     */
453    void
454    sleep(Rendez *r, int (*f)(void*), void *arg)
455    {
456      int s;
457
458      s = splhi();
459
460      lock(&up->rlock);
461      if(r->p){
462              print("double sleep %lud %lud\n", r->p->pid, up->pid);
463              dumpstack();
464      }
465
466      /*
467       *  Wakeup only knows there may be something to do by testing
468       *  r->p in order to get something to lock on.
469       *  Flush that information out to memory in case the sleep is
470       *  committed.
471       */
472      r->p = up;
473      coherence();
474
475      if((*f)(arg) || up->notepending){
476              /*
477               *  if condition happened or a note is pending
478               *  never mind
479               */
480              r->p = nil;
481              unlock(&up->rlock);
482      } else {
483              /*
484               *  now we are committed to
485               *  change state and call scheduler
486               */
487              up->state = Wakeme;
488              up->r = r;
489
490              /* statistics */
491              m->cs++;
492
493              procsave(up);
494              if(setlabel(&up->sched)) {
495                      /*
496                       *  here when the process is awakened
497                       */
498                      procrestore(up);
499                      spllo();
500              } else {
501                      /*
502                       *  here to go to sleep (i.e. stop Running)
503                       */
504                      unlock(&up->rlock);
505                      gotolabel(&m->sched);
506              }
507      }
508
509      if(up->notepending) {
510              up->notepending = 0;
511              splx(s);
512              error(Eintr);
513      }
514
515      splx(s);
516    }
517
518    int
519    tfn(void *arg)
520    {
521      return MACHP(0)->ticks >= up->twhen || up->tfn(arg);
522    }
523
524    void
525    tsleep(Rendez *r, int (*fn)(void*), void *arg, int ms)
526    {
527      ulong when;
528      Proc *f, **l;
529
530      /* avoid overflows at the cost of precision */
531      if(ms >= 1000000)
532              when = ms/(1000/HZ);
533      else
534              when = MS2TK(ms);
535      when += MACHP(0)->ticks;
536
537      lock(&talarm);
538      /* take out of list if checkalarm didn't */
539      if(up->trend) {
540              l = &talarm.list;
541              for(f = *l; f; f = f->tlink) {
542                      if(f == up) {
543                              *l = up->tlink;
544                              break;
```

```
545                     }
546                     l = &f->tlink;
547             }
548     }
549     /* insert in increasing time order */
550     l = &talarm.list;
551     for(f = *l; f; f = f->tlink) {
552             if(f->twhen >= when)
553                     break;
554             l = &f->tlink;
555     }
556     up->trend = r;
557     up->twhen = when;
558     up->tfn = fn;
559     up->tlink = *l;
560     *l = up;
561     unlock(&talarm);
562
563     if(waserror()){
564             up->twhen = 0;
565             nexterror();
566     }
567     sleep(r, tfn, arg);
568     up->twhen = 0;
569     poperror();
570   }
571
572   /*
573    * Expects that only one process can call wakeup for any given Rendez
574    */
575   int
576   wakeup(Rendez *r)
577   {
578     Proc *p;
579     int s, rv;
580
581     /*
582      *  this makes sure that the condition sleep checks
583      *  with its (*f)(void) is visible to it
584      */
585     coherence();
586
587     rv = 0;
588     p = r->p;
589     if(p == 0)
590             return rv;
591
592     s = splhi();
593     lock(&p->rlock);
594
595     if(r->p == p && p->r == r){
596             r->p = 0;
597             if(p->state != Wakeme)
598                     panic("wakeup: state");
599             p->r = 0;
600             ready(p);
601             rv = 1;
602     }
603
604     unlock(&p->rlock);
605     splx(s);
606
607     return rv;
608   }
609
610   int
611   postnote(Proc *p, int dolock, char *n, int flag)
612   {
613     int s, ret;
614     Rendez *r;
615     Proc *d, **l;
616
617     if(dolock)
618             qlock(&p->debug);
619
620   //        if(p->kp)
621   //                print("sending %s to kproc %lud %s\n", n, p->pid, p->text);
622
623     if(flag != NUser && (p->notify == 0 || p->notified))
624             p->nnote = 0;
625
626     ret = 0;
627     if(p->nnote < NNOTE) {
628             strcpy(p->note[p->nnote].msg, n);
629             p->note[p->nnote++].flag = flag;
630             ret = 1;
631     }
632     p->notepending = 1;
633     if(dolock)
634             qunlock(&p->debug);
635
636     s = splhi();
637     lock(&p->rlock);
638     r = p->r;
639     if(r){
640             if(r->p != p || p->r != r || p->state != Wakeme)
641                     panic("postnote: state %d %d %d", r->p != p, p->r != r, p->state);
642             r->p = 0;
643             p->r = 0;
644             ready(p);
645     }
646     unlock(&p->rlock);
647     splx(s);
648
649     if(p->state != Rendezvous)
650             return ret;
651
652     /* Try and pull out of a rendezvous */
653     lock(p->rgrp);
654     if(p->state == Rendezvous) {
655             p->rendval = ~0;
656             l = &REND(p->rgrp, p->rendtag);
657             for(d = *l; d; d = d->rendhash) {
658                     if(d == p) {
659                             *l = p->rendhash;
660                             break;
661                     }
662                     l = &d->rendhash;
663             }
664             ready(p);
665     }
666     unlock(p->rgrp);
667     return ret;
668   }
669
670   /*
671    * weird thing: keep at most NBROKEN around
672    */
673   #define   NBROKEN 4
674   struct
675   {
676     QLock;
677     int     n;
678     Proc    *p[NBROKEN];
679   }broken;
680
```

```
void
addbroken(Proc *p)
{
  qlock(&broken);
  if(broken.n == NBROKEN) {
          ready(broken.p[0]);
          memmove(&broken.p[0], &broken.p[1], sizeof(Proc*)*(NBROKEN-1));
          --broken.n;
  }
  broken.p[broken.n++] = p;
  qunlock(&broken);

  p->state = Broken;
  p->psstate = 0;
  sched();
}

void
unbreak(Proc *p)
{
  int b;

  qlock(&broken);
  for(b=0; b < broken.n; b++)
          if(broken.p[b] == p) {
                  broken.n--;
                  memmove(&broken.p[b], &broken.p[b+1],
                                  sizeof(Proc*)*(NBROKEN-(b+1)));
                  ready(p);
                  break;
          }
  qunlock(&broken);
}

int
freebroken(void)
{
  int i, n;

  qlock(&broken);
  n = broken.n;
  for(i=0; i<n; i++) {
          ready(broken.p[i]);
          broken.p[i] = 0;
  }
  broken.n = 0;
  qunlock(&broken);
  return n;
}

void
pexit(char *exitstr, int freemem)
{
  int n;
  Proc *p;
  Segment **s, **es;
  long utime, stime;
  Waitq *wq, *f, *next;
  Fgrp *fgrp;
  Egrp *egrp;
  Rgrp *rgrp;
  Pgrp *pgrp;
  Chan *dot;

  up->alarm = 0;

  /* nil out all the resources under lock (free later) */
  qlock(&up->debug);
  fgrp = up->fgrp;
  up->fgrp = nil;
  egrp = up->egrp;
  up->egrp = nil;
  rgrp = up->rgrp;
  up->rgrp = nil;
  pgrp = up->pgrp;
  up->pgrp = nil;
  dot = up->dot;
  up->dot = nil;
  qunlock(&up->debug);

  if(fgrp)
          closefgrp(fgrp);
  if(egrp)
          closeegrp(egrp);
  if(rgrp)
          closergrp(rgrp);
  if(dot)
          cclose(dot);
  if(pgrp)
          closepgrp(pgrp);

  /*
   * if not a kernel process and have a parent,
   * do some housekeeping.
   */
  if(up->kp == 0) {
          p = up->parent;
          if(p == 0) {
                  if(exitstr == 0)
                          exitstr = "unknown";
                  panic("boot process died: %s", exitstr);
          }

          while(waserror())
                  ;

          wq = smalloc(sizeof(Waitq));
          poperror();

          readnum(0, wq->w.pid, NUMSIZE, up->pid, NUMSIZE);
          utime = up->time[TUser] + up->time[TCUser];
          stime = up->time[TSys] + up->time[TCSys];
          readnum(0, &wq->w.time[TUser*12], NUMSIZE,
                  TK2MS(utime), NUMSIZE);
          readnum(0, &wq->w.time[TSys*12], NUMSIZE,
                  TK2MS(stime), NUMSIZE);
          readnum(0, &wq->w.time[TReal*12], NUMSIZE,
                  TK2MS(MACHP(0)->ticks - up->time[TReal]), NUMSIZE);
          if(exitstr && exitstr[0]){
                  n = sprint(wq->w.msg, "%s %lud:", up->text, up->pid);
                  strncpy(wq->w.msg+n, exitstr, ERRLEN-n);
                  wq->w.msg[ERRLEN-1] = 0;
          }
          else
                  wq->w.msg[0] = '\0';

          lock(&p->exl);
          /*
           *  If my parent is no longer alive, or if there would be more
           *  than 128 zombie child processes for my parent, then don't
           *  leave a wait record behind.  This helps prevent badly
           *  written daemon processes from accumulating lots of wait
           *  records.
           */
          if(p->pid == up->parentpid && p->state != Broken && p->nwait < 128) {
                  p->nchild--;
```

```
817                     p->time[TCUser] += utime;
818                     p->time[TCSys] += stime;
819
820                     wq->next = p->waitq;
821                     p->waitq = wq;
822                     p->nwait++;
823
824                     wakeup(&p->waitr);
825                     unlock(&p->exl);
826             }
827             else {
828                     unlock(&p->exl);
829                     free(wq);
830             }
831     }
832
833     if(!freemem)
834             addbroken(up);
835
836     qlock(&up->seglock);
837     es = &up->seg[NSEG];
838     for(s = up->seg; s < es; s++) {
839             if(*s) {
840                     putseg(*s);
841                     *s = 0;
842             }
843     }
844     qunlock(&up->seglock);
845
846     lock(&up->exl);         /* Prevent my children from leaving waits */
847     up->pid = 0;
848     wakeup(&up->waitr);
849     unlock(&up->exl);
850
851     for(f = up->waitq; f; f = next) {
852             next = f->next;
853             free(f);
854     }
855
856     /* release debuggers */
857     qlock(&up->debug);
858     if(up->pdbg) {
859             wakeup(&up->pdbg->sleep);
860             up->pdbg = 0;
861     }
862     qunlock(&up->debug);
863
864     /* Sched must not loop for these locks */
865     lock(&procalloc);
866     lock(&palloc);
867
868     up->state = Moribund;
869     sched();
870     panic("pexit");
871   }
872
873   int
874   haswaitq(void *x)
875   {
876     Proc *p;
877
878     p = (Proc *)x;
879     return p->waitq != 0;
880   }
881
882   ulong
883   pwait(Waitmsg *w)
884   {
885     ulong cpid;
886     Waitq *wq;
887
888     if(!canqlock(&up->qwaitr))
889             error(Einuse);
890
891     if(waserror()) {
892             qunlock(&up->qwaitr);
893             nexterror();
894     }
895
896     lock(&up->exl);
897     if(up->nchild == 0 && up->waitq == 0) {
898             unlock(&up->exl);
899             error(Enochild);
900     }
901     unlock(&up->exl);
902
903     sleep(&up->waitr, haswaitq, up);
904
905     lock(&up->exl);
906     wq = up->waitq;
907     up->waitq = wq->next;
908     up->nwait--;
909     unlock(&up->exl);
910
911     qunlock(&up->qwaitr);
912     poperror();
913
914     if(w)
915             memmove(w, &wq->w, sizeof(Waitmsg));
916     cpid = atoi(wq->w.pid);
917     free(wq);
918     return cpid;
919   }
920
921   Proc*
922   proctab(int i)
923   {
924     return &procalloc.arena[i];
925   }
926
927   void
928   dumpaproc(Proc *p)
929   {
930     ulong bss;
931     char *s;
932
933     if(p == 0)
934             return;
935
936     bss = 0;
937     if(p->seg[BSEG])
938             bss = p->seg[BSEG]->top;
939
940     s = p->psstate;
941     if(s == 0)
942             s = "kproc";
943     print("%3lud:%10s pc %8lux dbgpc %8lux  %8s (%s) ut %ld st %ld bss %lux qpc %lux\n",
944             p->pid, p->text, p->pc, dbgpc(p),  s, statename[p->state],
945             p->time[0], p->time[1], bss, p->qpc);
946   }
947
948   void
949   procdump(void)
950   {
951     int i;
952     Proc *p;
```

```
953
954     if(up)
955             print("up %lud\n", up->pid);
956     else
957             print("no current process\n");
958     for(i=0; i<conf.nproc; i++) {
959             p = &procalloc.arena[i];
960             if(p->state == Dead)
961                     continue;
962
963             dumpaproc(p);
964             delay(150);
965     }
966   }
967
968   /*
969    *  wait till all processes have flushed their mmu
970    *  state about segement s
971    */
972   void
973   procflushseg(Segment *s)
974   {
975     int i, ns, nm, nwait;
976     Proc *p;
977
978     /*
979      *  tell all processes with this
980      *  segment to flush their mmu's
981      */
982     nwait = 0;
983     for(i=0; i<conf.nproc; i++) {
984             p = &procalloc.arena[i];
985             if(p->state == Dead)
986                     continue;
987             for(ns = 0; ns < NSEG; ns++)
988                     if(p->seg[ns] == s){
989                             p->newtlb = 1;
990                             for(nm = 0; nm < conf.nmach; nm++){
991                                     if(MACHP(nm)->proc == p){
992                                             MACHP(nm)->flushmmu = 1;
993                                             nwait++;
994                                     }
995                             }
996                             break;
997                     }
998     }
999
1000    if(nwait == 0)
1001            return;
1002
1003    /*
1004     *  wait for all processors to take a clock interrupt
1005     *  and flush their mmu's
1006     */
1007    for(nm = 0; nm < conf.nmach; nm++)
1008            if(MACHP(nm) != m)
1009                    while(MACHP(nm)->flushmmu)
1010                            sched();
1011  }
1012
1013  void
1014  scheddump(void)
1015  {
1016    Proc *p;
1017    Schedq *rq;
1018
1019    for(rq = &runq[Nrq-1]; rq >= runq; rq--){
1020            if(rq->head == 0)
1021                    continue;
1022            print("rq%ld:", rq-runq);
1023            for(p = rq->head; p; p = p->rnext)
1024                    print(" %lud(%lud, %lud)", p->pid, m->ticks - p->readytime,
1025                            MACHP(0)->ticks - p->movetime);
1026            print("\n");
1027            delay(150);
1028    }
1029    print("nrdy %d\n", nrdy);
1030  }
1031
1032  void
1033  kproc(char *name, void (*func)(void *), void *arg)
1034  {
1035    Proc *p;
1036    static Pgrp *kpgrp;
1037
1038    p = newproc();
1039    p->psstate = 0;
1040    p->procmode = 0644;
1041    p->kp = 1;
1042
1043    p->fpsave = up->fpsave;
1044    p->scallnr = up->scallnr;
1045    p->s = up->s;
1046    p->nerrlab = 0;
1047    p->slash = up->slash;
1048    p->dot = up->dot;
1049    incref(p->dot);
1050
1051    memmove(p->note, up->note, sizeof(p->note));
1052    p->nnote = up->nnote;
1053    p->notified = 0;
1054    p->lastnote = up->lastnote;
1055    p->notify = up->notify;
1056    p->ureg = 0;
1057    p->dbgreg = 0;
1058
1059    p->basepri = PriKproc;
1060    p->priority = p->basepri;
1061
1062    kprocchild(p, func, arg);
1063
1064    strcpy(p->user, eve);
1065    if(kpgrp == 0)
1066            kpgrp = newpgrp();
1067    p->pgrp = kpgrp;
1068    incref(kpgrp);
1069
1070    strcpy(p->text, name);
1071
1072    p->nchild = 0;
1073    p->parent = 0;
1074    memset(p->time, 0, sizeof(p->time));
1075    p->time[TReal] = MACHP(0)->ticks;
1076    ready(p);
1077    /*
1078     *  since the bss/data segments are now shareable,
1079     *  any mmu info about this process is now stale
1080     *  and has to be discarded.
1081     */
1082    flushmmu();
1083  }
1084
1085  /*
1086   *  called splhi() by notify().  See comment in notify for the
1087   *  reasoning.
1088   */
```

```
void
procctl(Proc *p)
{
  char *state;
  ulong s;

  switch(p->procctl) {
  case Proc_exitbig:
          spllo();
          pexit("Killed: Insufficient physical memory", 1);

  case Proc_exitme:
          spllo();                /* pexit has locks in it */
          pexit("Killed", 1);

  case Proc_traceme:
          if(p->nnote == 0)
                  return;
          /* No break */

  case Proc_stopme:
          p->procctl = 0;
          state = p->psstate;
          p->psstate = "Stopped";
          /* free a waiting debugger */
          s = spllo();
          qlock(&p->debug);
          if(p->pdbg) {
                  wakeup(&p->pdbg->sleep);
                  p->pdbg = 0;
          }
          qunlock(&p->debug);
          splhi();
          p->state = Stopped;
          sched();
          p->psstate = state;
          splx(s);
          return;
  }
}

#include "errstr.h"

void
error(char *err)
{
  spllo();
  strncpy(up->error, err, ERRLEN);
  nexterror();
}

void
nexterror(void)
{
  gotolabel(&up->errlab[--up->nerrlab]);
}

void
exhausted(char *resource)
{
  char buf[ERRLEN];

  sprint(buf, "no free %s", resource);
  iprint("%s\n", buf);
  error(buf);
}

void
killbig(void)
{
  int i;
  Segment *s;
  ulong l, max;
  Proc *p, *ep, *kp;

  max = 0;
  kp = 0;
  ep = procalloc.arena+conf.nproc;
  for(p = procalloc.arena; p < ep; p++) {
          if(p->state == Dead || p->kp)
                  continue;
          l = 0;
          for(i=1; i<NSEG; i++) {
                  s = p->seg[i];
                  if(s != 0)
                          l += s->top - s->base;
          }
          if(l > max) {
                  kp = p;
                  max = l;
          }
  }
  kp->procctl = Proc_exitbig;
  for(i = 0; i < NSEG; i++) {
          s = kp->seg[i];
          if(s != 0 && canqlock(&s->lk)) {
                  mfreeseg(s, s->base, (s->top - s->base)/BY2PG);
                  qunlock(&s->lk);
          }
  }
  print("%lud: %s killed because no swap configured\n", kp->pid, kp->text);
}

/*
 *  change ownership to 'new' of all processes owned by 'old'.  Used when
 *  eve changes.
 */
void
renameuser(char *old, char *new)
{
  Proc *p, *ep;

  ep = procalloc.arena+conf.nproc;
  for(p = procalloc.arena; p < ep; p++)
          if(strcmp(old, p->user) == 0)
                  memmove(p->user, new, NAMELEN);
}

/*
 *  time accounting called by clock() splhi'd
 */
void
accounttime(void)
{
  Proc *p;
  int n;
  static int nrun;

  p = m->proc;
  if(p) {
          nrun++;
          p->time[p->insyscall]++;
  }

  /* only one processor gets to compute system load averages */
  if(m->machno != 0)
```

```
1225            return;
1226
1227    /* calculate decaying load average */
1228    n = nrun;
1229    nrun = 0;
1230
1231    n = (nrdy+n)*1000;
1232    m->load = (m->load*19+n)/20;
1233  }
```

```
#include  "u.h"
#include  "../port/lib.h"
#include  "mem.h"
#include  "dat.h"
#include  "fns.h"
#include  "../port/error.h"

static ulong padblockcnt;
static ulong concatblockcnt;
static ulong pullupblockcnt;
static ulong copyblockcnt;
static ulong consumecnt;
static ulong producecnt;
static ulong qcopycnt;

static int debugging;

#define QDEBUG    if(0)

/*
 *  IO queues
 */
typedef struct Queue      Queue;

struct Queue
{
  Lock;

  Block*  bfirst;         /* buffer */
  Block*  blast;

  int     len;            /* bytes allocated to queue */
  int     dlen;           /* data bytes in queue */
  int     limit;          /* max bytes in queue */
  int     inilim;         /* initial limit */
  int     state;
  int     noblock;        /* true if writes return immediately when q full */
  int     eof;            /* number of eofs read by user */

  void    (*kick)(void*); /* restart output */
  void*   arg;            /* argument to kick */

  QLock   rlock;          /* mutex for reading processes */
  Rendez  rr;             /* process waiting to read */
  QLock   wlock;          /* mutex for writing processes */
  Rendez  wr;             /* process waiting to write */

  char    err[ERRLEN];
};

enum
{
  /* Queue.state */
  Qstarve         = (1<<0),       /* consumer starved */
  Qmsg            = (1<<1),       /* message stream */
  Qclosed         = (1<<2),
  Qflow           = (1<<3),

  Maxatomic       = 32*1024,
};

void
ixsummary(void)
{
  debugging ^= 1;
  iallocsummary();
  print("pad %lud, concat %lud, pullup %lud, copy %lud\n",
          padblockcnt, concatblockcnt, pullupblockcnt, copyblockcnt);
  print("consume %lud, produce %lud, qcopy %lud\n",
          consumecnt, producecnt, qcopycnt);
}

/*
 *  free a list of blocks
 */
void
freeblist(Block *b)
{
  Block *next;

  for(; b != 0; b = next){
          next = b->next;
          b->next = 0;
          freeb(b);
  }
}

/*
 *  pad a block to the front (or the back if size is negative)
 */
Block*
padblock(Block *bp, int size)
{
  int n;
  Block *nbp;

  QDEBUG checkb(bp, "padblock 1");
  if(size >= 0){
          if(bp->rp - bp->base >= size){
                  bp->rp -= size;
                  return bp;
          }

          if(bp->next)
                  panic("padblock 0x%uX", getcallerpc(&bp));
          n = BLEN(bp);
          padblockcnt++;
          nbp = allocb(size+n);
          nbp->rp += size;
          nbp->wp = nbp->rp;
          memmove(nbp->wp, bp->rp, n);
          nbp->wp += n;
          freeb(bp);
          nbp->rp -= size;
  } else {
          size = -size;

          if(bp->next)
                  panic("padblock 0x%uX", getcallerpc(&bp));

          if(bp->lim - bp->wp >= size)
                  return bp;

          n = BLEN(bp);
          padblockcnt++;
          nbp = allocb(size+n);
          memmove(nbp->wp, bp->rp, n);
          nbp->wp += n;
          freeb(bp);
  }
  QDEBUG checkb(nbp, "padblock 1");
  return nbp;
}

/*
 *  return count of bytes in a string of blocks
```

```
137	 */
138	int
139	blocklen(Block *bp)
140	{
141	  int len;
142	
143	  len = 0;
144	  while(bp) {
145	          len += BLEN(bp);
146	          bp = bp->next;
147	  }
148	  return len;
149	}
150	
151	/*
152	 *  copy the  string of blocks into
153	 *  a single block and free the string
154	 */
155	Block*
156	concatblock(Block *bp)
157	{
158	  int len;
159	  Block *nb, *f;
160	
161	  if(bp->next == 0)
162	          return bp;
163	
164	  nb = allocb(blocklen(bp));
165	  for(f = bp; f; f = f->next) {
166	          len = BLEN(f);
167	          memmove(nb->wp, f->rp, len);
168	          nb->wp += len;
169	  }
170	  concatblockcnt += BLEN(nb);
171	  freeblist(bp);
172	  QDEBUG checkb(nb, "concatblock 1");
173	  return nb;
174	}
175	
176	/*
177	 *  make sure the first block has at least n bytes
178	 */
179	Block*
180	pullupblock(Block *bp, int n)
181	{
182	  int i;
183	  Block *nbp;
184	
185	  /*
186	   *  this should almost always be true, it's
187	   *  just to avoid every caller checking.
188	   */
189	  if(BLEN(bp) >= n)
190	          return bp;
191	
192	  /*
193	   *  if not enough room in the first block,
194	   *  add another to the front of the list.
195	   */
196	  if(bp->lim - bp->rp < n){
197	          nbp = allocb(n);
198	          nbp->next = bp;
199	          bp = nbp;
200	  }
201	
202	  /*
203	   *  copy bytes from the trailing blocks into the first
204	   */
205	  n -= BLEN(bp);
206	  while(nbp = bp->next){
207	          i = BLEN(nbp);
208	          if(i > n) {
209	                  memmove(bp->wp, nbp->rp, n);
210	                  pullupblockcnt++;
211	                  bp->wp += n;
212	                  nbp->rp += n;
213	                  QDEBUG checkb(bp, "pullupblock 1");
214	                  return bp;
215	          }
216	          else {
217	                  memmove(bp->wp, nbp->rp, i);
218	                  pullupblockcnt++;
219	                  bp->wp += i;
220	                  bp->next = nbp->next;
221	                  nbp->next = 0;
222	                  freeb(nbp);
223	                  n -= i;
224	                  if(n == 0){
225	                          QDEBUG checkb(bp, "pullupblock 2");
226	                          return bp;
227	                  }
228	          }
229	  }
230	  freeb(bp);
231	  return 0;
232	}
233	
234	/*
235	 *  trim to len bytes starting at offset
236	 */
237	Block *
238	trimblock(Block *bp, int offset, int len)
239	{
240	  ulong l;
241	  Block *nb, *startb;
242	
243	  QDEBUG checkb(bp, "trimblock 1");
244	  if(blocklen(bp) < offset+len) {
245	          freeblist(bp);
246	          return nil;
247	  }
248	
249	  while((l = BLEN(bp)) < offset) {
250	          offset -= l;
251	          nb = bp->next;
252	          bp->next = nil;
253	          freeb(bp);
254	          bp = nb;
255	  }
256	
257	  startb = bp;
258	  bp->rp += offset;
259	
260	  while((l = BLEN(bp)) < len) {
261	          len -= l;
262	          bp = bp->next;
263	  }
264	
265	  bp->wp -= (BLEN(bp) - len);
266	
267	  if(bp->next) {
268	          freeblist(bp->next);
269	          bp->next = nil;
270	  }
271	
272	  return startb;
```

```
}

/*
 *  copy 'count' bytes into a new block
 */
Block*
copyblock(Block *bp, int count)
{
  int l;
  Block *nbp;

  QDEBUG checkb(bp, "copyblock 0");
  nbp = allocb(count);
  for(; count > 0 && bp != 0; bp = bp->next){
          l = BLEN(bp);
          if(l > count)
                  l = count;
          memmove(nbp->wp, bp->rp, l);
          nbp->wp += l;
          count -= l;
  }
  if(count > 0){
          memset(nbp->wp, 0, count);
          nbp->wp += count;
  }
  copyblockcnt++;
  QDEBUG checkb(nbp, "copyblock 1");

  return nbp;
}

Block*
adjustblock(Block* bp, int len)
{
  int n;
  Block *nbp;

  if(len < 0){
          freeb(bp);
          return nil;
  }

  if(bp->rp+len > bp->lim){
          nbp = copyblock(bp, len);
          freeblist(bp);
          QDEBUG checkb(nbp, "adjustblock 1");

          return nbp;
  }

  n = BLEN(bp);
  if(len > n)
          memset(bp->wp, 0, len-n);
  bp->wp = bp->rp+len;
  QDEBUG checkb(bp, "adjustblock 2");

  return bp;
}


/*
 *  throw away up to count bytes from a
 *  list of blocks.  Return count of bytes
 *  thrown away.
 */
int
pullblock(Block **bph, int count)
{
  Block *bp;
  int n, bytes;

  bytes = 0;
  if(bph == nil)
          return 0;

  while(*bph != nil && count != 0) {
          bp = *bph;
          n = BLEN(bp);
          if(count < n)
                  n = count;
          bytes += n;
          count -= n;
          bp->rp += n;
          QDEBUG checkb(bp, "pullblock ");
          if(BLEN(bp) == 0) {
                  *bph = bp->next;
                  bp->next = nil;
                  freeb(bp);
          }
  }
  return bytes;
}

/*
 *  get next block from a queue, return null if nothing there
 */
Block*
qget(Queue *q)
{
  int dowakeup;
  Block *b;

  /* sync with qwrite */
  ilock(q);

  b = q->bfirst;
  if(b == 0){
          q->state |= Qstarve;
          iunlock(q);
          return 0;
  }
  q->bfirst = b->next;
  b->next = 0;
  q->len -= BALLOC(b);
  q->dlen -= BLEN(b);
  QDEBUG checkb(b, "qget");

  /* if writer flow controlled, restart */
  if((q->state & Qflow) && q->len < q->limit/2){
          q->state &= ~Qflow;
          dowakeup = 1;
  } else
          dowakeup = 0;

  iunlock(q);

  if(dowakeup)
          wakeup(&q->wr);

  return b;
}

/*
 *  throw away the next 'len' bytes in the queue
 */
void
```

```
409  qdiscard(Queue *q, int len)
410  {
411    Block *b;
412    int dowakeup, n, sofar;
413
414    ilock(q);
415    for(sofar = 0; sofar < len; sofar += n){
416            b = q->bfirst;
417            if(b == nil)
418                    break;
419            QDEBUG checkb(b, "qdiscard");
420            n = BLEN(b);
421            if(n <= len - sofar){
422                    q->bfirst = b->next;
423                    b->next = 0;
424                    q->len -= BALLOC(b);
425                    q->dlen -= BLEN(b);
426                    freeb(b);
427            } else {
428                    n = len - sofar;
429                    b->rp += n;
430                    q->dlen -= n;
431            }
432    }
433
434    /* if writer flow controlled, restart */
435    if((q->state & Qflow) && q->len < q->limit/2){
436            q->state &= ~Qflow;
437            dowakeup = 1;
438    } else
439            dowakeup = 0;
440
441    iunlock(q);
442
443    if(dowakeup)
444            wakeup(&q->wr);
445  }
446
447  /*
448   *  Interrupt level copy out of a queue, return # bytes copied.
449   */
450  int
451  qconsume(Queue *q, void *vp, int len)
452  {
453    Block *b;
454    int n, dowakeup;
455    uchar *p = vp;
456    Block *tofree = nil;
457
458    /* sync with qwrite */
459    ilock(q);
460
461    for(;;) {
462            b = q->bfirst;
463            if(b == 0){
464                    q->state |= Qstarve;
465                    iunlock(q);
466                    return -1;
467            }
468            QDEBUG checkb(b, "qconsume 1");
469
470            n = BLEN(b);
471            if(n > 0)
472                    break;
473            q->bfirst = b->next;
474            q->len -= BALLOC(b);
475
476            /* remember to free this */
477            b->next = tofree;
478            tofree = b;
479    };
480
481    if(n < len)
482            len = n;
483    memmove(p, b->rp, len);
484    consumecnt += n;
485    if((q->state & Qmsg) || len == n)
486            q->bfirst = b->next;
487    b->rp += len;
488    q->dlen -= len;
489
490    /* discard the block if we're done with it */
491    if((q->state & Qmsg) || len == n){
492            b->next = 0;
493            q->len -= BALLOC(b);
494            q->dlen -= BLEN(b);
495
496            /* remember to free this */
497            b->next = tofree;
498            tofree = b;
499    }
500
501    /* if writer flow controlled, restart */
502    if((q->state & Qflow) && q->len < q->limit/2){
503            q->state &= ~Qflow;
504            dowakeup = 1;
505    } else
506            dowakeup = 0;
507
508    iunlock(q);
509
510    if(dowakeup)
511            wakeup(&q->wr);
512
513    if(tofree != nil)
514            freeblist(tofree);
515
516    return len;
517  }
518
519  int
520  qpass(Queue *q, Block *b)
521  {
522    int dlen, len, dowakeup;
523
524    /* sync with qread */
525    dowakeup = 0;
526    ilock(q);
527    if(q->len >= q->limit){
528            freeblist(b);
529            iunlock(q);
530            return -1;
531    }
532
533    /* add buffer to queue */
534    if(q->bfirst)
535            q->blast->next = b;
536    else
537            q->bfirst = b;
538    len = BALLOC(b);
539    dlen = BLEN(b);
540    QDEBUG checkb(b, "qpass");
541    while(b->next){
542            b = b->next;
543            QDEBUG checkb(b, "qpass");
544            len += BALLOC(b);
```

```
545             dlen += BLEN(b);
546     }
547     q->blast = b;
548     q->len += len;
549     q->dlen += dlen;
550
551     if(q->len >= q->limit/2)
552             q->state |= Qflow;
553
554     if(q->state & Qstarve){
555             q->state &= ~Qstarve;
556             dowakeup = 1;
557     }
558     iunlock(q);
559
560     if(dowakeup)
561             wakeup(&q->rr);
562
563     return len;
564   }
565
566   int
567   qpassnolim(Queue *q, Block *b)
568   {
569     int dlen, len, dowakeup;
570
571     /* sync with qread */
572     dowakeup = 0;
573     ilock(q);
574
575     /* add buffer to queue */
576     if(q->bfirst)
577             q->blast->next = b;
578     else
579             q->bfirst = b;
580     len = BALLOC(b);
581     dlen = BLEN(b);
582     QDEBUG checkb(b, "qpass");
583     while(b->next){
584             b = b->next;
585             QDEBUG checkb(b, "qpass");
586             len += BALLOC(b);
587             dlen += BLEN(b);
588     }
589     q->blast = b;
590     q->len += len;
591     q->dlen += dlen;
592
593     if(q->len >= q->limit/2)
594             q->state |= Qflow;
595
596     if(q->state & Qstarve){
597             q->state &= ~Qstarve;
598             dowakeup = 1;
599     }
600     iunlock(q);
601
602     if(dowakeup)
603             wakeup(&q->rr);
604
605     return len;
606   }
607
608   /*
609    *  if the allocated space is way out of line with the used
610    *  space, reallocate to a smaller block
611    */
612   Block*
613   packblock(Block *bp)
614   {
615     Block **l, *nbp;
616     int n;
617
618     for(l = &bp; *l; l = &(*l)->next){
619             nbp = *l;
620             n = BLEN(nbp);
621             if((n<<2) < BALLOC(nbp)){
622                     *l = allocb(n);
623                     memmove((*l)->wp, nbp->rp, n);
624                     (*l)->wp += n;
625                     (*l)->next = nbp->next;
626                     freeb(nbp);
627             }
628     }
629
630     return bp;
631   }
632
633   int
634   qproduce(Queue *q, void *vp, int len)
635   {
636     Block *b;
637     int dowakeup;
638     uchar *p = vp;
639
640     /* sync with qread */
641     dowakeup = 0;
642     ilock(q);
643
644     /* no waiting receivers, room in buffer? */
645     if(q->len >= q->limit){
646             q->state |= Qflow;
647             iunlock(q);
648             return -1;
649     }
650
651     /* save in buffer */
652     b = iallocb(len);
653     if(b == 0){
654             iunlock(q);
655             return 0;
656     }
657     memmove(b->wp, p, len);
658     producecnt += len;
659     b->wp += len;
660     if(q->bfirst)
661             q->blast->next = b;
662     else
663             q->bfirst = b;
664     q->blast = b;
665     /* b->next = 0; done by iallocb() */
666     q->len += BALLOC(b);
667     q->dlen += BLEN(b);
668     QDEBUG checkb(b, "qproduce");
669
670     if(q->state & Qstarve){
671             q->state &= ~Qstarve;
672             dowakeup = 1;
673     }
674
675     if(q->len >= q->limit)
676             q->state |= Qflow;
677     iunlock(q);
678
679     if(dowakeup)
680             wakeup(&q->rr);
```

```
681
682       return len;
683     }
684
685     /*
686      *  copy from offset in the queue
687      */
688     Block*
689     qcopy(Queue *q, int len, ulong offset)
690     {
691       int sofar;
692       int n;
693       Block *b, *nb;
694       uchar *p;
695
696       nb = allocb(len);
697
698       ilock(q);
699
700       /* go to offset */
701       b = q->bfirst;
702       for(sofar = 0; ; sofar += n){
703               if(b == nil){
704                       iunlock(q);
705                       return nb;
706               }
707               n = BLEN(b);
708               if(sofar + n > offset){
709                       p = b->rp + offset - sofar;
710                       n -= offset - sofar;
711                       break;
712               }
713               QDEBUG checkb(b, "qcopy");
714               b = b->next;
715       }
716
717       /* copy bytes from there */
718       for(sofar = 0; sofar < len;){
719               if(n > len - sofar)
720                       n = len - sofar;
721               memmove(nb->wp, p, n);
722               qcopycnt += n;
723               sofar += n;
724               nb->wp += n;
725               b = b->next;
726               if(b == nil)
727                       break;
728               n = BLEN(b);
729               p = b->rp;
730       }
731       iunlock(q);
732
733       return nb;
734     }
735
736     /*
737      *  called by non-interrupt code
738      */
739     Queue*
740     qopen(int limit, int msg, void (*kick)(void*), void *arg)
741     {
742       Queue *q;
743
744       q = malloc(sizeof(Queue));
745       if(q == 0)
746               return 0;
747
748       ilock(q);
749       q->limit = q->inilim = limit;
750       q->kick = kick;
751       q->arg = arg;
752       q->state = msg ? Qmsg : 0;
753       q->state |= Qstarve;
754       q->eof = 0;
755       q->noblock = 0;
756       iunlock(q);
757
758       return q;
759     }
760
761     static int
762     notempty(void *a)
763     {
764       Queue *q = a;
765
766       return (q->state & Qclosed) || q->bfirst != 0;
767     }
768
769     /*
770      *  get next block from a queue (up to a limit)
771      */
772     Block*
773     qbread(Queue *q, int len)
774     {
775       Block *b, *nb;
776       int n, dowakeup;
777
778       qlock(&q->rlock);
779       if(waserror()){
780               qunlock(&q->rlock);
781               nexterror();
782       }
783
784       /* wait for data */
785       for(;;){
786               /* sync with qwrite/qproduce */
787               ilock(q);
788
789               b = q->bfirst;
790               if(b){
791                       QDEBUG checkb(b, "qbread 0");
792                       break;
793               }
794
795               if(q->state & Qclosed){
796                       iunlock(q);
797                       poperror();
798                       qunlock(&q->rlock);
799                       if(++q->eof > 3)
800                               error(q->err);
801                       return 0;
802               }
803
804               q->state |= Qstarve;    /* flag requesting producer to wake me */
805               iunlock(q);
806               sleep(&q->rr, notempty, q);
807       }
808
809       /* remove a buffered block */
810       q->bfirst = b->next;
811       b->next = 0;
812       n = BLEN(b);
813       q->dlen -= n;
814       q->len -= BALLOC(b);
815       QDEBUG checkb(b, "qbread 1");
816
```

```
817     /* split block if it's too big and this is not a message-oriented queue */
818     nb = b;
819     if(n > len){
820             if((q->state&Qmsg) == 0){
821                     iunlock(q);
822
823                     n -= len;
824                     b = allocb(n);
825                     memmove(b->wp, nb->rp+len, n);
826                     b->wp += n;
827
828                     ilock(q);
829                     b->next = q->bfirst;
830                     if(q->bfirst == 0)
831                             q->blast = b;
832                     q->bfirst = b;
833                     q->len += BALLOC(b);
834                     q->dlen += n;
835             }
836             nb->wp = nb->rp + len;
837     }
838
839     /* if writer flow controlled, restart */
840     if((q->state & Qflow) && q->len < q->limit/2){
841             q->state &= ~Qflow;
842             dowakeup = 1;
843     } else
844             dowakeup = 0;
845
846     iunlock(q);
847
848     /* wakeup flow controlled writers */
849     if(dowakeup){
850             if(q->kick)
851                     q->kick(q->arg);
852             wakeup(&q->wr);
853     }
854
855     poperror();
856     qunlock(&q->rlock);
857     return nb;
858   }
859
860   /*
861    *  read a queue.  if no data is queued, post a Block
862    *  and wait on its Rendez.
863    */
864   long
865   qread(Queue *q, void *vp, int len)
866   {
867     Block *b;
868
869     b = qbread(q, len);
870     if(b == 0)
871             return 0;
872
873     len = BLEN(b);
874     memmove(vp, b->rp, len);
875     freeb(b);
876     return len;
877   }
878
879   static int
880   qnotfull(void *a)
881   {
882     Queue *q = a;
883
884     return q->len < q->limit || (q->state & Qclosed);
885   }
886
887   /*
888    *  add a block to a queue obeying flow control
889    */
890   long
891   qbwrite(Queue *q, Block *b)
892   {
893     int n, dowakeup;
894
895     dowakeup = 0;
896     n = BLEN(b);
897     qlock(&q->wlock);
898     if(waserror()){
899             qunlock(&q->wlock);
900             nexterror();
901     }
902
903     /* flow control */
904     for(;;){
905             ilock(q);
906
907             if(q->state & Qclosed){
908                     iunlock(q);
909                     freeb(b);
910                     error(q->err);
911             }
912
913             if(q->len < q->limit)
914                     break;
915
916             if(q->noblock){
917                     iunlock(q);
918                     freeb(b);
919                     qunlock(&q->wlock);
920                     poperror();
921                     return n;
922             }
923
924             q->state |= Qflow;
925             iunlock(q);
926             sleep(&q->wr, qnotfull, q);
927     }
928
929     if(q->bfirst)
930             q->blast->next = b;
931     else
932             q->bfirst = b;
933     q->blast = b;
934     b->next = 0;
935     q->len += BALLOC(b);
936     q->dlen += n;
937     QDEBUG checkb(b, "qbwrite");
938
939     if(q->state & Qstarve){
940             q->state &= ~Qstarve;
941             dowakeup = 1;
942     }
943
944     iunlock(q);
945
946     if(dowakeup){
947             if(q->kick)
948                     q->kick(q->arg);
949             wakeup(&q->rr);
950     }
951
952     qunlock(&q->wlock);
```

```
953     poperror();
954     return n;
955   }
956
957   /*
958    *  write to a queue.  only Maxatomic bytes at a time is atomic.
959    */
960   int
961   qwrite(Queue *q, void *vp, int len)
962   {
963     int n, sofar;
964     Block *b;
965     uchar *p = vp;
966
967     QDEBUG if(islo() == 0)
968             print("qwrite hi %lux\n", getcallerpc(&q));
969
970     sofar = 0;
971     do {
972             n = len-sofar;
973             if(n > Maxatomic)
974                     n = Maxatomic;
975
976             b = allocb(n);
977             setmalloctag(b, (up->text[0]<<24)|(up->text[1]<<16)|(up->text[2]<<8)|up->text[3]);
978             if(waserror()){
979                     freeb(b);
980                     nexterror();
981             }
982             memmove(b->wp, p+sofar, n);
983             poperror();
984             b->wp += n;
985
986             qbwrite(q, b);
987
988             sofar += n;
989     } while(sofar < len && (q->state & Qmsg) == 0);
990
991     return len;
992   }
993
994   /*
995    *  used by print() to write to a queue.  Since we may be splhi or not in
996    *  a process, don't qlock.
997    */
998   int
999   qiwrite(Queue *q, void *vp, int len)
1000  {
1001    int n, sofar, dowakeup;
1002    Block *b;
1003    uchar *p = vp;
1004
1005    dowakeup = 0;
1006
1007    sofar = 0;
1008    do {
1009            n = len-sofar;
1010            if(n > Maxatomic)
1011                    n = Maxatomic;
1012
1013            b = allocb(n);
1014            memmove(b->wp, p+sofar, n);
1015            b->wp += n;
1016
1017            ilock(q);
1018
1019            QDEBUG checkb(b, "qiwrite");
1020            if(q->bfirst)
1021                    q->blast->next = b;
1022            else
1023                    q->bfirst = b;
1024            q->blast = b;
1025            q->len += BALLOC(b);
1026            q->dlen += n;
1027
1028            if(q->state & Qstarve){
1029                    q->state &= ~Qstarve;
1030                    dowakeup = 1;
1031            }
1032
1033            iunlock(q);
1034
1035            if(dowakeup){
1036                    if(q->kick)
1037                            q->kick(q->arg);
1038                    wakeup(&q->rr);
1039            }
1040
1041            sofar += n;
1042    } while(sofar < len && (q->state & Qmsg) == 0);
1043
1044    return len;
1045  }
1046
1047  /*
1048   *  be extremely careful when calling this,
1049   *  as there is no reference accounting
1050   */
1051  void
1052  qfree(Queue *q)
1053  {
1054    qclose(q);
1055    free(q);
1056  }
1057
1058  /*
1059   *  Mark a queue as closed.  No further IO is permitted.
1060   *  All blocks are released.
1061   */
1062  void
1063  qclose(Queue *q)
1064  {
1065    Block *bfirst;
1066
1067    if(q == nil)
1068            return;
1069
1070    /* mark it */
1071    ilock(q);
1072    q->state |= Qclosed;
1073    q->state &= ~(Qflow|Qstarve);
1074    strcpy(q->err, Ehungup);
1075    bfirst = q->bfirst;
1076    q->bfirst = 0;
1077    q->len = 0;
1078    q->dlen = 0;
1079    q->noblock = 0;
1080    iunlock(q);
1081
1082    /* free queued blocks */
1083    freeblist(bfirst);
1084
1085    /* wake up readers/writers */
1086    wakeup(&q->rr);
1087    wakeup(&q->wr);
1088  }
```

```
1089
1090  /*
1091   *  Mark a queue as closed.  Wakeup any readers.  Don't remove queued
1092   *  blocks.
1093   */
1094  void
1095  qhangup(Queue *q, char *msg)
1096  {
1097    /* mark it */
1098    ilock(q);
1099    q->state |= Qclosed;
1100    if(msg == 0 || *msg == 0)
1101            strcpy(q->err, Ehungup);
1102    else
1103            strncpy(q->err, msg, ERRLEN-1);
1104    iunlock(q);
1105
1106    /* wake up readers/writers */
1107    wakeup(&q->rr);
1108    wakeup(&q->wr);
1109  }
1110
1111  /*
1112   *  return non-zero if the q is hungup
1113   */
1114  int
1115  qisclosed(Queue *q)
1116  {
1117    return q->state & Qclosed;
1118  }
1119
1120  /*
1121   *  mark a queue as no longer hung up
1122   */
1123  void
1124  qreopen(Queue *q)
1125  {
1126    ilock(q);
1127    q->state &= ~Qclosed;
1128    q->state |= Qstarve;
1129    q->eof = 0;
1130    q->limit = q->inilim;
1131    iunlock(q);
1132  }
1133
1134  /*
1135   *  return bytes queued
1136   */
1137  int
1138  qlen(Queue *q)
1139  {
1140    return q->dlen;
1141  }
1142
1143  /*
1144   * return space remaining before flow control
1145   */
1146  int
1147  qwindow(Queue *q)
1148  {
1149    int l;
1150
1151    l = q->limit - q->len;
1152    if(l < 0)
1153            l = 0;
1154    return l;
1155  }
1156
1157  /*
1158   *  return true if we can read without blocking
1159   */
1160  int
1161  qcanread(Queue *q)
1162  {
1163    return q->bfirst!=0;
1164  }
1165
1166  /*
1167   *  change queue limit
1168   */
1169  void
1170  qsetlimit(Queue *q, int limit)
1171  {
1172    q->limit = limit;
1173  }
1174
1175  /*
1176   *  set blocking/nonblocking
1177   */
1178  void
1179  qnoblock(Queue *q, int onoff)
1180  {
1181    q->noblock = onoff;
1182  }
1183
1184  /*
1185   *  flush the output queue
1186   */
1187  void
1188  qflush(Queue *q)
1189  {
1190    Block *bfirst;
1191
1192    /* mark it */
1193    ilock(q);
1194    bfirst = q->bfirst;
1195    q->bfirst = 0;
1196    q->len = 0;
1197    q->dlen = 0;
1198    iunlock(q);
1199
1200    /* free queued blocks */
1201    freeblist(bfirst);
1202
1203    /* wake up readers/writers */
1204    wakeup(&q->wr);
1205  }
1206
1207  int
1208  qfull(Queue *q)
1209  {
1210    return q->state & Qflow;
1211  }
1212
1213  int
1214  qstate(Queue *q)
1215  {
1216    return q->state;
1217  }
```

```
#include "u.h"
#include "../port/lib.h"
#include "mem.h"
#include "dat.h"
#include "fns.h"

struct {
  ulong rlock;
  ulong rlockq;
  ulong wlock;
  ulong wlockq;
  ulong qlock;
  ulong qlockq;
} rwstats;

void
qlock(QLock *q)
{
  Proc *p, *mp;

  lock(&q->use);
rwstats.qlock++;
  if(!q->locked) {
          q->locked = 1;
          unlock(&q->use);
          return;
  }
rwstats.qlockq++;
  p = q->tail;
  mp = up;
  if(mp == nil)
          panic("qlock");
  if(p == 0)
          q->head = mp;
  else
          p->qnext = mp;
  q->tail = mp;
  mp->qnext = 0;
  mp->state = Queueing;
  up->qpc = getcallerpc(&q);
  unlock(&q->use);
  sched();
}

int
canqlock(QLock *q)
{
  if(!canlock(&q->use))
          return 0;
  if(q->locked){
          unlock(&q->use);
          return 0;
  }
  q->locked = 1;
  unlock(&q->use);
  return 1;
}

void
qunlock(QLock *q)
{
  Proc *p;

  lock(&q->use);
  p = q->head;
  if(p) {
          q->head = p->qnext;
          if(q->head == 0)
                  q->tail = 0;
          unlock(&q->use);
          ready(p);
          return;
  }
  q->locked = 0;
  unlock(&q->use);
}

void
rlock(RWlock *q)
{
  Proc *p, *mp;

  lock(&q->use);
rwstats.rlock++;
  if(q->writer == 0 && q->head == nil){
          /* no writer, go for it */
          q->readers++;
          unlock(&q->use);
          return;
  }

rwstats.rlockq++;
  p = q->tail;
  mp = up;
  if(mp == nil)
          panic("rlock");
  if(p == 0)
          q->head = mp;
  else
          p->qnext = mp;
  q->tail = mp;
  mp->qnext = 0;
  mp->state = QueueingR;
  unlock(&q->use);
  sched();
}

void
runlock(RWlock *q)
{
  Proc *p;

  lock(&q->use);
  p = q->head;
  if(--(q->readers) > 0 || p == nil){
          unlock(&q->use);
          return;
  }

  /* start waiting writer */
  if(p->state != QueueingW)
          panic("runlock");
  q->head = p->qnext;
  if(q->head == 0)
          q->tail = 0;
  q->writer = 1;
  unlock(&q->use);
  ready(p);
}

void
wlock(RWlock *q)
{
  Proc *p, *mp;

  lock(&q->use);
```

```
137   rwstats.wlock++;
138     if(q->readers == 0 && q->writer == 0){
139             /* noone waiting, go for it */
140             q->wpc = getcallerpc(&q);
141             q->wproc = up;
142             q->writer = 1;
143             unlock(&q->use);
144             return;
145     }
146
147     /* wait */
148   rwstats.wlockq++;
149     p = q->tail;
150     mp = up;
151     if(mp == nil)
152             panic("wlock");
153     if(p == nil)
154             q->head = mp;
155     else
156             p->qnext = mp;
157     q->tail = mp;
158     mp->qnext = 0;
159     mp->state = QueueingW;
160     unlock(&q->use);
161     sched();
162   }
163
164   void
165   wunlock(RWlock *q)
166   {
167     Proc *p;
168
169     lock(&q->use);
170     p = q->head;
171     if(p == nil){
172             q->writer = 0;
173             unlock(&q->use);
174             return;
175     }
176     if(p->state == QueueingW){
177             /* start waiting writer */
178             q->head = p->qnext;
179             if(q->head == nil)
180                     q->tail = nil;
181             unlock(&q->use);
182             ready(p);
183             return;
184     }
185
186     if(p->state != QueueingR)
187             panic("wunlock");
188
189     /* waken waiting readers */
190     while(q->head != nil && q->head->state == QueueingR){
191             p = q->head;
192             q->head = p->qnext;
193             q->readers++;
194             ready(p);
195     }
196     if(q->head == nil)
197             q->tail = nil;
198     q->writer = 0;
199     unlock(&q->use);
200   }
201
202   /* same as rlock but punts if there are any readers waiting */
203   int
204   canrlock(RWlock *q)
205   {
206     lock(&q->use);
207   rwstats.rlock++;
208     if(q->writer == 0 && q->head == nil){
209             /* no writer, go for it */
210             q->readers++;
211             unlock(&q->use);
212             return 1;
213     }
214     unlock(&q->use);
215     return 0;
216   }
```

```
#include  "u.h"
#include  "../port/lib.h"
#include  "mem.h"
#include  "dat.h"
#include  "fns.h"
#include  "../port/error.h"

static void       imagereclaim(void);
static void       imagechanreclaim(void);

/* System specific segattach devices */
#include "io.h"
#include "segment.h"

static Lock physseglock;

#define NFREECHAN 64
#define IHASHSIZE 64
#define ihash(s)  imagealloc.hash[s%IHASHSIZE]
static struct
{
  Lock;
  Image   *free;
  Image   *hash[IHASHSIZE];
  QLock   ireclaim;       /* mutex on reclaiming free images */

  Chan    **freechan;     /* free image channels */
  int     nfreechan;      /* number of free channels */
  int     szfreechan;     /* size of freechan array */
  QLock   fcreclaim;      /* mutex on reclaiming free channels */
}imagealloc;

void
initseg(void)
{
  Image *i, *ie;

  imagealloc.free = xalloc(conf.nimage*sizeof(Image));
  ie = &imagealloc.free[conf.nimage-1];
  for(i = imagealloc.free; i < ie; i++)
          i->next = i+1;
  i->next = 0;
  imagealloc.freechan = malloc(NFREECHAN * sizeof(Chan*));
  imagealloc.szfreechan = NFREECHAN;
}

Segment *
newseg(int type, ulong base, ulong size)
{
  Segment *s;
  int mapsize;

  if(size > (SEGMAPSIZE*PTEPERTAB))
          error(Enovmem);

  if(swapfull())
          error(Enoswap);

  s = smalloc(sizeof(Segment));
  s->ref = 1;
  s->type = type;
  s->base = base;
  s->top = base+(size*BY2PG);
  s->size = size;

  mapsize = ROUND(size, PTEPERTAB)/PTEPERTAB;
  if(mapsize > nelem(s->ssegmap)){
          mapsize *= 2;
          if(mapsize > (SEGMAPSIZE*PTEPERTAB))
                  mapsize = (SEGMAPSIZE*PTEPERTAB);
          s->map = smalloc(mapsize*sizeof(Pte*));
          s->mapsize = mapsize;
  }
  else{
          s->map = s->ssegmap;
          s->mapsize = nelem(s->ssegmap);
  }

  return s;
}

void
putseg(Segment *s)
{
  Pte **pp, **emap;
  Image *i;

  if(s == 0)
          return;

  i = s->image;
  if(i != 0) {
          lock(i);
          lock(s);
          if(i->s == s && s->ref == 1)
                  i->s = 0;
          unlock(i);
  }
  else
          lock(s);

  s->ref--;
  if(s->ref != 0) {
          unlock(s);
          return;
  }

  qlock(&s->lk);
  if(i)
          putimage(i);

  emap = &s->map[s->mapsize];
  for(pp = s->map; pp < emap; pp++)
          if(*pp)
                  freepte(s, *pp);

  qunlock(&s->lk);
  if(s->map != s->ssegmap)
          free(s->map);
  if(s->profile != 0)
          free(s->profile);
  free(s);
}

void
relocateseg(Segment *s, ulong offset)
{
  Page **pg, *x;
  Pte *pte, **p, **endpte;

  endpte = &s->map[s->mapsize];
  for(p = s->map; p < endpte; p++) {
          if(*p == 0)
                  continue;
          pte = *p;
          for(pg = pte->first; pg <= pte->last; pg++) {
```

```
137                     if(x = *pg)
138                             x->va += offset;
139             }
140       }
141     }
142
143     Segment*
144     dupseg(Segment **seg, int segno, int share)
145     {
146       int i, size;
147       Pte *pte;
148       Segment *n, *s;
149
150       SET(n);
151       s = seg[segno];
152
153       switch(s->type&SG_TYPE) {
154       case SG_TEXT:           /* New segment shares pte set */
155       case SG_SHARED:
156       case SG_PHYSICAL:
157       case SG_SHDATA:
158               incref(s);
159               return s;
160
161       case SG_STACK:
162               qlock(&s->lk);
163               if(waserror()){
164                       qunlock(&s->lk);
165                       nexterror();
166               }
167               n = newseg(s->type, s->base, s->size);
168               poperror();
169               break;
170
171       case SG_BSS:            /* Just copy on write */
172       case SG_MAP:
173               qlock(&s->lk);
174               if(share && s->ref == 1) {
175                       s->type = (s->type&~SG_TYPE)|SG_SHARED;
176                       incref(s);
177                       qunlock(&s->lk);
178                       return s;
179               }
180               if(waserror()){
181                       qunlock(&s->lk);
182                       nexterror();
183               }
184               n = newseg(s->type, s->base, s->size);
185               poperror();
186               break;
187
188       case SG_DATA:           /* Copy on write plus demand load info */
189               if(segno == TSEG)
190                       return data2txt(s);
191
192               qlock(&s->lk);
193               if(share && s->ref == 1) {
194                       s->type = (s->type&~SG_TYPE)|SG_SHDATA;
195                       incref(s);
196                       qunlock(&s->lk);
197                       return s;
198               }
199               if(waserror()){
200                       qunlock(&s->lk);
201                       nexterror();
202               }
203               n = newseg(s->type, s->base, s->size);
204               poperror();
205
206               incref(s->image);
207               n->image = s->image;
208               n->fstart = s->fstart;
209               n->flen = s->flen;
210               break;
211       }
212       size = s->mapsize;
213       for(i = 0; i < size; i++)
214               if(pte = s->map[i])
215                       n->map[i] = ptecpy(pte);
216
217       n->flushme = s->flushme;
218       qunlock(&s->lk);
219       return n;
220     }
221
222     void
223     segpage(Segment *s, Page *p)
224     {
225       Pte **pte;
226       ulong off;
227       Page **pg;
228
229       if(p->va < s->base || p->va >= s->top)
230               panic("segpage");
231
232       off = p->va - s->base;
233       pte = &s->map[off/PTEMAPMEM];
234       if(*pte == 0)
235               *pte = ptealloc();
236
237       pg = &(*pte)->pages[(off&(PTEMAPMEM-1))/BY2PG];
238       *pg = p;
239       if(pg < (*pte)->first)
240               (*pte)->first = pg;
241       if(pg > (*pte)->last)
242               (*pte)->last = pg;
243     }
244
245     Image*
246     attachimage(int type, Chan *c, ulong base, ulong len)
247     {
248       Image *i, **l;
249
250       /* reclaim any free channels from reclaimed segments */
251       if(imagealloc.nfreechan)
252               imagechanreclaim();
253
254       lock(&imagealloc);
255
256       /*
257        * Search the image cache for remains of the text from a previous
258        * or currently running incarnation
259        */
260       for(i = ihash(c->qid.path); i; i = i->hash) {
261               if(c->qid.path == i->qid.path) {
262                       lock(i);
263                       if(eqqid(c->qid, i->qid) &&
264                          eqqid(c->mqid, i->mqid) &&
265                          c->mchan == i->mchan &&
266                          c->type == i->type) {
267                               i->ref++;
268                               goto found;
269                       }
270                       unlock(i);
271               }
272       }
```

```
273
274     /*
275      * imagereclaim dumps pages from the free list which are cached by image
276      * structures. This should free some image structures.
277      */
278     while(!(i = imagealloc.free)) {
279             unlock(&imagealloc);
280             imagereclaim();
281             sched();
282             lock(&imagealloc);
283     }
284
285     imagealloc.free = i->next;
286
287     lock(i);
288     incref(c);
289     i->c = c;
290     i->type = c->type;
291     i->qid = c->qid;
292     i->mqid = c->mqid;
293     i->mchan = c->mchan;
294     i->ref = 1;
295     l = &ihash(c->qid.path);
296     i->hash = *l;
297     *l = i;
298   found:
299     unlock(&imagealloc);
300
301     if(i->s == 0) {
302             /* Disaster after commit in exec */
303             if(waserror()) {
304                     unlock(i);
305                     pexit(Enovmem, 1);
306             }
307             i->s = newseg(type, base, len);
308             i->s->image = i;
309             poperror();
310     }
311     else
312             incref(i->s);
313
314     return i;
315   }
316
317   static struct {
318     int     calls;                  /* times imagereclaim was called */
319     int     loops;                  /* times the main loop was run */
320     uvlong  ticks;                  /* total time in the main loop */
321     uvlong  maxt;                   /* longest time in main loop */
322   } irstats;
323
324   static void
325   imagereclaim(void)
326   {
327     Page *p;
328     uvlong ticks;
329
330     irstats.calls++;
331     /* Somebody is already cleaning the page cache */
332     if(!canqlock(&imagealloc.ireclaim))
333             return;
334
335     lock(&palloc);
336     ticks = fastticks(nil);
337     for(p = palloc.head; p; p = p->next) {
338             if(p->ref == 0 && p->image && canlock(p)) {
339                     if(p->ref == 0)
340                             uncachepage(p);
341                     unlock(p);
342             }
343     }
344     ticks = fastticks(nil) - ticks;
345     unlock(&palloc);
346     irstats.loops++;
347     irstats.ticks += ticks;
348     if(ticks > irstats.maxt)
349             irstats.maxt = ticks;
350     //print("T%llud+", ticks);
351     qunlock(&imagealloc.ireclaim);
352   }
353
354   /*
355    *  since close can block, this has to be called outside of
356    *  spin locks.
357    */
358   static void
359   imagechanreclaim(void)
360   {
361     Chan *c;
362
363     /* Somebody is already cleaning the image chans */
364     if(!canqlock(&imagealloc.fcreclaim))
365             return;
366
367     /*
368      * We don't have to recheck that nfreechan > 0 after we
369      * acquire the lock, because we're the only ones who decrement
370      * it (the other lock contender increments it), and there's only
371      * one of us thanks to the qlock above.
372      */
373     while(imagealloc.nfreechan > 0){
374             lock(&imagealloc);
375             imagealloc.nfreechan--;
376             c = imagealloc.freechan[imagealloc.nfreechan];
377             unlock(&imagealloc);
378             cclose(c);
379     }
380
381     qunlock(&imagealloc.fcreclaim);
382   }
383
384   void
385   putimage(Image *i)
386   {
387     Chan *c, **cp;
388     Image *f, **l;
389
390     if(i->notext)
391             return;
392
393     lock(i);
394     if(--i->ref == 0) {
395             l = &ihash(i->qid.path);
396             i->qid = (Qid){~0, ~0};
397             unlock(i);
398             c = i->c;
399
400             lock(&imagealloc);
401             for(f = *l; f; f = f->hash) {
402                     if(f == i) {
403                             *l = i->hash;
404                             break;
405                     }
406                     l = &f->hash;
407             }
408
```

```
409             i->next = imagealloc.free;
410             imagealloc.free = i;
411
412             /* defer freeing channel till we're out of spin lock's */
413             if(imagealloc.nfreechan == imagealloc.szfreechan){
414                     imagealloc.szfreechan += NFREECHAN;
415                     cp = malloc(imagealloc.szfreechan*sizeof(Chan*));
416                     if(cp == nil)
417                             panic("putimage");
418                     memmove(cp, imagealloc.freechan, imagealloc.nfreechan*sizeof(Chan*));
419                     free(imagealloc.freechan);
420                     imagealloc.freechan = cp;
421             }
422             imagealloc.freechan[imagealloc.nfreechan++] = c;
423             unlock(&imagealloc);
424
425             return;
426     }
427     unlock(i);
428   }
429
430   long
431   ibrk(ulong addr, int seg)
432   {
433     Segment *s, *ns;
434     ulong newtop, newsize;
435     int i, mapsize;
436     Pte **map;
437
438     s = up->seg[seg];
439     if(s == 0)
440             error(Ebadarg);
441
442     if(addr == 0)
443             return s->base;
444
445     qlock(&s->lk);
446
447     /* We may start with the bss overlapping the data */
448     if(addr < s->base) {
449             if(seg != BSEG || up->seg[DSEG] == 0 || addr < up->seg[DSEG]->base) {
450                     qunlock(&s->lk);
451                     error(Enovmem);
452             }
453             addr = s->base;
454     }
455
456     newtop = PGROUND(addr);
457     newsize = (newtop-s->base)/BY2PG;
458     if(newtop < s->top) {
459             mfreeseg(s, newtop, (s->top-newtop)/BY2PG);
460             qunlock(&s->lk);
461             flushmmu();
462             return 0;
463     }
464
465     if(swapfull()){
466             qunlock(&s->lk);
467             error(Enoswap);
468     }
469
470     for(i = 0; i < NSEG; i++) {
471             ns = up->seg[i];
472             if(ns == 0 || ns == s)
473                     continue;
474             if(newtop >= ns->base && newtop < ns->top) {
475                     qunlock(&s->lk);
476                     error(Esoverlap);
477             }
478     }
479
480     if(newsize > (SEGMAPSIZE*PTEPERTAB)) {
481             qunlock(&s->lk);
482             error(Enovmem);
483     }
484     mapsize = ROUND(newsize, PTEPERTAB)/PTEPERTAB;
485     if(mapsize > s->mapsize){
486             map = smalloc(mapsize*sizeof(Pte*));
487             memmove(map, s->map, s->mapsize*sizeof(Pte*));
488             if(s->map != s->ssegmap)
489                     free(s->map);
490             s->map = map;
491             s->mapsize = mapsize;
492     }
493
494     s->top = newtop;
495     s->size = newsize;
496     qunlock(&s->lk);
497     return 0;
498   }
499
500   /*
501    *  called with s->lk locked
502    */
503   void
504   mfreeseg(Segment *s, ulong start, int pages)
505   {
506     int i, j, size;
507     ulong soff;
508     Page *pg;
509     Page *list;
510
511     soff = start-s->base;
512     j = (soff&(PTEMAPMEM-1))/BY2PG;
513
514     size = s->mapsize;
515     list = nil;
516     for(i = soff/PTEMAPMEM; i < size; i++) {
517             if(pages <= 0)
518                     break;
519             if(s->map[i] == 0) {
520                     pages -= PTEPERTAB-j;
521                     j = 0;
522                     continue;
523             }
524             while(j < PTEPERTAB) {
525                     pg = s->map[i]->pages[j];
526                     if(pg){
527                             s->map[i]->pages[j] = 0;
528                             pg->next = list;
529                             list = pg;
530                     }
531                     if(--pages == 0)
532                             goto out;
533                     j++;
534             }
535             j = 0;
536     }
537   out:
538     /* flush this seg in all other processes */
539     i = s->type&SG_TYPE;
540     switch(i){
541     case SG_SHARED:
542     case SG_SHDATA:
543             procflushseg(s);
544             break;
```

```
545     }
546
547     /* free the pages */
548     for(pg = list; pg != nil; pg = list){
549             list = list->next;
550             putpage(pg);
551     }
552   }
553
554   Segment*
555   isoverlap(Proc *p, ulong va, int len)
556   {
557     int i;
558     Segment *ns;
559     ulong newtop;
560
561     newtop = va+len;
562     for(i = 0; i < NSEG; i++) {
563             ns = p->seg[i];
564             if(ns == 0)
565                     continue;
566             if((newtop > ns->base && newtop <= ns->top) ||
567                (va >= ns->base && va < ns->top))
568                     return ns;
569     }
570     return nil;
571   }
572
573   int
574   addphysseg(Physseg* new)
575   {
576     Physseg *ps;
577
578     /*
579      * Check not already entered and there is room
580      * for a new entry and the terminating null entry.
581      */
582     lock(&physseglock);
583     for(ps = physseg; ps->name; ps++){
584             if(strcmp(ps->name, new->name) == 0){
585                     unlock(&physseglock);
586                     return -1;
587             }
588     }
589     if(ps-physseg >= nelem(physseg)-2){
590             unlock(&physseglock);
591             return -1;
592     }
593
594     *ps = *new;
595     unlock(&physseglock);
596
597     return 0;
598   }
599
600   ulong
601   segattach(Proc *p, ulong attr, char *name, ulong va, ulong len)
602   {
603     int sno;
604     Segment *s, *os;
605     Physseg *ps;
606
607     if(va != 0 && (va&KZERO) == KZERO)      /* BUG: Only ok for now */
608             error(Ebadarg);
609
610     validaddr((ulong)name, 1, 0);
611     vmemchr(name, 0, ~0);
612
613     for(sno = 0; sno < NSEG; sno++)
614             if(p->seg[sno] == nil && sno != ESEG)
615                     break;
616
617     if(sno == NSEG)
618             error(Enovmem);
619
620     len = PGROUND(len);
621     if(len == 0)
622             error(Ebadarg);
623
624     /*
625      * Find a hole in the address space.
626      * Starting at the lowest possible stack address - len,
627      * check for an overlapping segment, and repeat at the
628      * base of that segment - len until either a hole is found
629      * or the address space is exhausted.
630      */
631     if(va == 0) {
632             va = p->seg[SSEG]->base - len;
633             for(;;) {
634                     os = isoverlap(p, va, len);
635                     if(os == nil)
636                             break;
637                     va = os->base;
638                     if(len > va)
639                             error(Enovmem);
640                     va -= len;
641             }
642     }
643
644     va = va&~(BY2PG-1);
645     if(isoverlap(p, va, len) != nil)
646             error(Esoverlap);
647
648     for(ps = physseg; ps->name; ps++)
649             if(strcmp(name, ps->name) == 0)
650                     goto found;
651
652     error(Ebadarg);
653   found:
654     if(len > ps->size)
655             error(Enovmem);
656
657     attr &= ~SG_TYPE;               /* Turn off what is not allowed */
658     attr |= ps->attr;               /* Copy in defaults */
659
660     s = newseg(attr, va, len/BY2PG);
661     s->pseg = ps;
662     p->seg[sno] = s;
663
664     return va;
665   }
666
667   void
668   pteflush(Pte *pte, int s, int e)
669   {
670     int i;
671     Page *p;
672
673     for(i = s; i < e; i++) {
674             p = pte->pages[i];
675             if(pagedout(p) == 0)
676                     memset(p->cachectl, PG_TXTFLUSH, sizeof(p->cachectl));
677     }
678   }
679
680   long
```

```
681  syssegflush(ulong *arg)
682  {
683    Segment *s;
684    ulong addr, l;
685    Pte *pte;
686    int chunk, ps, pe, len;
687
688    addr = arg[0];
689    len = arg[1];
690
691    while(len > 0) {
692            s = seg(up, addr, 1);
693            if(s == 0)
694                    error(Ebadarg);
695
696            s->flushme = 1;
697    more:
698            l = len;
699            if(addr+l > s->top)
700                    l = s->top - addr;
701
702            ps = addr-s->base;
703            pte = s->map[ps/PTEMAPMEM];
704            ps &= PTEMAPMEM-1;
705            pe = PTEMAPMEM;
706            if(pe-ps > l){
707                    pe = ps + l;
708                    pe = (pe+BY2PG-1)&~(BY2PG-1);
709            }
710            if(pe == ps) {
711                    qunlock(&s->lk);
712                    error(Ebadarg);
713            }
714
715            if(pte)
716                    pteflush(pte, ps/BY2PG, pe/BY2PG);
717
718            chunk = pe-ps;
719            len -= chunk;
720            addr += chunk;
721
722            if(len > 0 && addr < s->top)
723                    goto more;
724
725            qunlock(&s->lk);
726    }
727    flushmmu();
728    return 0;
729  }
730
731  void
732  segclock(ulong pc)
733  {
734    Segment *s;
735
736    s = up->seg[TSEG];
737    if(s == 0 || s->profile == 0)
738            return;
739
740    s->profile[0] += TK2MS(1);
741    if(pc >= s->base && pc < s->top) {
742            pc -= s->base;
743            s->profile[pc>>LRESPROF] += TK2MS(1);
744    }
745  }
```

```
#include  "u.h"
#include  "../port/lib.h"
#include  "mem.h"
#include  "dat.h"
#include  "fns.h"
#include  "../port/error.h"

static int        canflush(Proc*, Segment*);
static void       executeio(void);
static int        needpages(void*);
static void       pageout(Proc*, Segment*);
static void       pagepte(int, Page**);
static void       pager(void*);

  Image   swapimage;
static    int     swopen;
static    Page    **iolist;
static    int     ioptr;

void
swapinit(void)
{
  swapalloc.swmap = xalloc(conf.nswap);
  swapalloc.top = &swapalloc.swmap[conf.nswap];
  swapalloc.alloc = swapalloc.swmap;
  swapalloc.last = swapalloc.swmap;
  swapalloc.free = conf.nswap;
  iolist = xalloc(conf.nswppo*sizeof(Page*));
  if(swapalloc.swmap == 0 || iolist == 0)
          panic("swapinit: not enough memory");

  swapimage.notext = 1;
}

ulong
newswap(void)
{
  uchar *look;

  lock(&swapalloc);

  if(swapalloc.free == 0){
          unlock(&swapalloc);
          return ~0;
  }

  look = memchr(swapalloc.last, 0, swapalloc.top-swapalloc.last);
  if(look == 0)
          panic("inconsistent swap");

  *look = 1;
  swapalloc.last = look;
  swapalloc.free--;
  unlock(&swapalloc);
  return (look-swapalloc.swmap) * BY2PG;
}

void
putswap(Page *p)
{
  uchar *idx;

  lock(&swapalloc);
  idx = &swapalloc.swmap[((ulong)p)/BY2PG];
  if(--(*idx) == 0) {
          swapalloc.free++;
          if(idx < swapalloc.last)
                  swapalloc.last = idx;
  }
  if(*idx >= 254)
          panic("putswap %lux == %ud", p, *idx);
  unlock(&swapalloc);
}

void
dupswap(Page *p)
{
  lock(&swapalloc);
  if(++swapalloc.swmap[((ulong)p)/BY2PG] == 0)
          panic("dupswap");
  unlock(&swapalloc);
}

int
swapcount(ulong daddr)
{
  return swapalloc.swmap[daddr/BY2PG];
}

void
kickpager(void)
{
  static int started;

  if(started)
          wakeup(&swapalloc.r);
  else {
          kproc("pager", pager, 0);
          started = 1;
  }
}

static void
pager(void *junk)
{
  int i;
  Segment *s;
  Proc *p, *ep;

  if(waserror())
          panic("pager: os error\n");

  p = proctab(0);
  ep = &p[conf.nproc];

loop:
  up->psstate = "Idle";
  sleep(&swapalloc.r, needpages, 0);

  while(needpages(junk)) {

          if(swapimage.c) {
                  p++;
                  if(p >= ep)
                          p = proctab(0);

                  if(p->state == Dead || p->kp)
                          continue;

                  if(!canqlock(&p->seglock))
                          continue;               /* process changing its segments */

                  for(i = 0; i < NSEG; i++) {
                          if(!needpages(junk)){
                                  qunlock(&p->seglock);
                                  goto loop;
```

```
137                             }
138
139                             if(s = p->seg[i]) {
140                                     switch(s->type&SG_TYPE) {
141                                     default:
142                                             break;
143                                     case SG_TEXT:
144                                             pageout(p, s);
145                                             break;
146                                     case SG_DATA:
147                                     case SG_BSS:
148                                     case SG_STACK:
149                                     case SG_SHARED:
150                                     case SG_SHDATA:
151                                     case SG_MAP:
152                                             up->psstate = "Pageout";
153                                             pageout(p, s);
154                                             if(ioptr != 0) {
155                                                     up->psstate = "I/O";
156                                                     executeio();
157                                             }
158                                             break;
159                                     }
160                             }
161                     }
162                     qunlock(&p->seglock);
163             }
164             else {
165                     if(!cpuserver)
166                             freebroken();   /* can use the memory */
167                     else
168                             killbig();
169
170                     /* Emulate the old system if no swap channel */
171                     print("no physical memory\n");
172                     tsleep(&swapalloc.pause, return0, 0, 5000);
173                     wakeup(&palloc.r);
174             }
175     }
176     goto loop;
177   }
178
179   static void
180   pageout(Proc *p, Segment *s)
181   {
182     int type, i, size;
183     Pte *l;
184     Page **pg, *entry;
185
186     if(!canqlock(&s->lk))   /* We cannot afford to wait, we will surely deadlock */
187             return;
188
189     if(s->steal) {          /* Protected by /dev/proc */
190             qunlock(&s->lk);
191             return;
192     }
193
194     if(!canflush(p, s)) {   /* Able to invalidate all tlbs with references */
195             qunlock(&s->lk);
196             putseg(s);
197             return;
198     }
199
200     if(waserror()) {
201             qunlock(&s->lk);
202             putseg(s);
203             return;
204     }
205
206     /* Pass through the pte tables looking for memory pages to swap out */
207     type = s->type&SG_TYPE;
208     size = s->mapsize;
209     for(i = 0; i < size; i++) {
210             l = s->map[i];
211             if(l == 0)
212                     continue;
213             for(pg = l->first; pg < l->last; pg++) {
214                     entry = *pg;
215                     if(pagedout(entry))
216                             continue;
217
218                     if(entry->modref & PG_REF) {
219                             entry->modref &= ~PG_REF;
220                             continue;
221                     }
222
223                     pagepte(type, pg);
224
225                     if(ioptr >= conf.nswppo)
226                             goto out;
227             }
228     }
229   out:
230     poperror();
231     qunlock(&s->lk);
232     putseg(s);
233   }
234
235   static int
236   canflush(Proc *p, Segment *s)
237   {
238     int i;
239     Proc *ep;
240
241     lock(s);
242     if(s->ref == 1) {               /* Easy if we are the only user */
243             s->ref++;
244             unlock(s);
245             return canpage(p);
246     }
247     s->ref++;
248     unlock(s);
249
250     /* Now we must do hardwork to ensure all processes which have tlb
251      * entries for this segment will be flushed if we succeed in paging it out
252      */
253     p = proctab(0);
254     ep = &p[conf.nproc];
255     while(p < ep) {
256             if(p->state != Dead) {
257                     for(i = 0; i < NSEG; i++)
258                             if(p->seg[i] == s)
259                                     if(!canpage(p))
260                                             return 0;
261             }
262             p++;
263     }
264     return 1;
265   }
266
267   static void
268   pagepte(int type, Page **pg)
269   {
270     ulong daddr;
271     Page *outp;
272
```

```
273     outp = *pg;
274     switch(type) {
275     case SG_TEXT:                           /* Revert to demand load */
276             putpage(outp);
277             *pg = 0;
278             break;
279
280     case SG_DATA:
281     case SG_BSS:
282     case SG_STACK:
283     case SG_SHARED:
284     case SG_SHDATA:
285     case SG_MAP:
286             /*
287              *  get a new swap address and clear any pages
288              *  referring to it from the cache
289              */
290             daddr = newswap();
291             if(daddr == ~0)
292                     break;
293             cachedel(&swapimage, daddr);
294
295             lock(outp);
296
297             /* forget anything that it used to cache */
298             uncachepage(outp);
299
300             /*
301              *  incr the reference count to make sure it sticks around while
302              *  being written
303              */
304             outp->ref++;
305
306             /*
307              *  enter it into the cache so that a fault happening
308              *  during the write will grab the page from the cache
309              *  rather than one partially written to the disk
310              */
311             outp->daddr = daddr;
312             cachepage(outp, &swapimage);
313             *pg = (Page*)(daddr|PG_ONSWAP);
314             unlock(outp);
315
316             /* Add page to IO transaction list */
317             iolist[ioptr++] = outp;
318             break;
319     }
320   }
321
322   void
323   pagersummary(void)
324   {
325     print("%lud/%lud memory %lud/%lud swap %d iolist\n",
326             palloc.user-palloc.freecount,
327             palloc.user, conf.nswap-swapalloc.free, conf.nswap,
328             ioptr);
329   }
330
331   static void
332   executeio(void)
333   {
334     Page *out;
335     int i, n;
336     Chan *c;
337     char *kaddr;
338     KMap *k;
339
340     c = swapimage.c;
341
342     for(i = 0; i < ioptr; i++) {
343             if(ioptr > conf.nswppo)
344                     panic("executeio: ioptr %d > %d\n", ioptr, conf.nswppo);
345             out = iolist[i];
346             k = kmap(out);
347             kaddr = (char*)VA(k);
348
349             if(waserror())
350                     panic("executeio: page out I/O error");
351
352             n = devtab[c->type]->write(c, kaddr, BY2PG, out->daddr);
353             if(n != BY2PG)
354                     nexterror();
355
356             kunmap(k);
357             poperror();
358
359             /* Free up the page after I/O */
360             lock(out);
361             out->ref--;
362             unlock(out);
363             putpage(out);
364     }
365     ioptr = 0;
366   }
367
368   static int
369   needpages(void*)
370   {
371     return palloc.freecount < swapalloc.headroom;
372   }
373
374   void
375   setswapchan(Chan *c)
376   {
377     char dirbuf[DIRLEN];
378     Dir d;
379
380     if(swapimage.c) {
381             if(swapalloc.free != conf.nswap){
382                     cclose(c);
383                     error(Einuse);
384             }
385             cclose(swapimage.c);
386     }
387
388     /*
389      *  if this isn't a file, set the swap space
390      *  to be at most the size of the partition
391      */
392     if(devtab[c->type]->dc != L'M'){
393             devtab[c->type]->stat(c, dirbuf);
394             convM2D(dirbuf, &d);
395             if(d.length < conf.nswap*BY2PG){
396                     conf.nswap = d.length/BY2PG;
397                     swapalloc.top = &swapalloc.swmap[conf.nswap];
398                     swapalloc.free = conf.nswap;
399             }
400     }
401
402     swapimage.c = c;
403   }
404
405   int
406   swapfull(void)
407   {
408     return swapalloc.free < conf.nswap/10;
```

```
409   }
```

```
#include  "u.h"
#include  "../port/lib.h"
#include  "mem.h"
#include  "dat.h"
#include  "fns.h"
#include  "../port/error.h"

/*
 * The sys*() routines needn't poperror() as they return directly to syscall().
 */

int
growfd(Fgrp *f, int fd)   /* fd is always >= 0 */
{
  Chan **newfd, **oldfd;

  if(fd < f->nfd)
          return 0;
  if(fd >= f->nfd+DELTAFD)
          return -1;      /* out of range */
  if(fd % 100 == 0)
          pprint("warning: process exceeds %d file descriptors\n", fd);
  /*
   * Unbounded allocation is unwise; besides, there are only 16 bits
   * of fid in 9P
   */
  if(f->nfd >= 5000){
    Exhausted:
          exhausted("file descriptors");
          return -1;
  }
  newfd = malloc((f->nfd+DELTAFD)*sizeof(Chan*));
  if(newfd == 0)
          goto Exhausted;
  oldfd = f->fd;
  memmove(newfd, oldfd, f->nfd*sizeof(Chan*));
  f->fd = newfd;
  free(oldfd);
  f->nfd += DELTAFD;
  if(fd > f->maxfd)
          f->maxfd = fd;
  return 1;
}

int
newfd(Chan *c)
{
  int fd;
  Fgrp *f;

  f = up->fgrp;
  lock(f);
  for(fd=0; fd<f->nfd; fd++)
          if(f->fd[fd] == 0)
                  break;
  if(fd >= f->nfd && growfd(f, fd) < 0){
          unlock(f);
          return -1;
  }
  if(fd > f->maxfd)
          f->maxfd = fd;
  f->fd[fd] = c;
  unlock(f);
  return fd;
}

Chan*
fdtochan(int fd, int mode, int chkmnt, int iref)
{
  Chan *c;
  Fgrp *f;

  c = 0;
  f = up->fgrp;

  lock(f);
  if(fd<0 || f->nfd<=fd || (c = f->fd[fd])==0) {
          unlock(f);
          error(Ebadfd);
  }
  if(iref)
          incref(c);
  unlock(f);

  if(chkmnt && (c->flag&CMSG)) {
          if(iref)
                  cclose(c);
          error(Ebadusefd);
  }

  if(mode<0 || c->mode==ORDWR)
          return c;

  if((mode&OTRUNC) && c->mode==OREAD) {
          if(iref)
                  cclose(c);
          error(Ebadusefd);
  }

  if((mode&~OTRUNC) != c->mode) {
          if(iref)
                  cclose(c);
          error(Ebadusefd);
  }

  return c;
}

int
openmode(ulong o)
{
  o &= ~(OTRUNC|OCEXEC|ORCLOSE);
  if(o > OEXEC)
          error(Ebadarg);
  if(o == OEXEC)
          return OREAD;
  return o;
}

long
sysfd2path(ulong *arg)
{
  Chan *c;

  validaddr(arg[1], arg[2], 1);

  c = fdtochan(arg[0], -1, 0, 1);

  if(c->name == nil)
          snprint((char*)arg[1], arg[2], "<null>");
  else
          snprint((char*)arg[1], arg[2], "%s", c->name->s);
  cclose(c);
  return 0;
}
```

```
long
syspipe(ulong *arg)
{
  int fd[2];
  Chan *c[2];
  Dev *d;
  Fgrp *f = up->fgrp;

  validaddr(arg[0], 2*BY2WD, 1);
  evenaddr(arg[0]);
  d = devtab[devno('|', 0)];
  c[0] = namec("#|", Atodir, 0, 0);
  c[1] = 0;
  fd[0] = -1;
  fd[1] = -1;
  if(waserror()){
          cclose(c[0]);
          if(c[1])
                  cclose(c[1]);
          if(fd[0] >= 0)
                  f->fd[fd[0]]=0;
          if(fd[1] >= 0)
                  f->fd[fd[1]]=0;
          nexterror();
  }
  c[1] = cclone(c[0], 0);
  if(walk(&c[0], "data", 1) < 0)
          error(Egreg);
  if(walk(&c[1], "data1", 1) < 0)
          error(Egreg);
  c[0] = d->open(c[0], ORDWR);
  c[1] = d->open(c[1], ORDWR);
  fd[0] = newfd(c[0]);
  fd[1] = newfd(c[1]);
  if(fd[0] < 0 || fd[1] < 0)
          error(Enofd);
  ((long*)arg[0])[0] = fd[0];
  ((long*)arg[0])[1] = fd[1];
  poperror();
  return 0;
}

long
sysdup(ulong *arg)
{
  int fd;
  Chan *c, *oc;
  Fgrp *f = up->fgrp;

  /*
   * Close after dup'ing, so date > #d/1 works
   */
  c = fdtochan(arg[0], -1, 0, 1);
  fd = arg[1];
  if(fd != -1){
          lock(f);
          if(fd<0 || growfd(f, fd)<0) {
                  unlock(f);
                  cclose(c);
                  error(Ebadfd);
          }
          if(fd > f->maxfd)
                  f->maxfd = fd;

          oc = f->fd[fd];
          f->fd[fd] = c;
          unlock(f);
          if(oc)
                  cclose(oc);
  }else{
          if(waserror()) {
                  cclose(c);
                  nexterror();
          }
          fd = newfd(c);
          if(fd < 0)
                  error(Enofd);
          poperror();
  }

  return fd;
}

long
sysopen(ulong *arg)
{
  int fd;
  Chan *c = 0;

  openmode(arg[1]);       /* error check only */
  if(waserror()){
          if(c)
                  cclose(c);
          nexterror();
  }
  validaddr(arg[0], 1, 0);
  c = namec((char*)arg[0], Aopen, arg[1], 0);
  fd = newfd(c);
  if(fd < 0)
          error(Enofd);
  poperror();
  return fd;
}

void
fdclose(int fd, int flag)
{
  int i;
  Chan *c;
  Fgrp *f = up->fgrp;

  lock(f);
  c = f->fd[fd];
  if(c == 0){
          /* can happen for users with shared fd tables */
          unlock(f);
          return;
  }
  if(flag){
          if(c==0 || !(c->flag&flag)){
                  unlock(f);
                  return;
          }
  }
  f->fd[fd] = 0;
  if(fd == f->maxfd)
          for(i=fd; --i>=0 && f->fd[i]==0; )
                  f->maxfd = i;

  unlock(f);
  cclose(c);
}

long
sysclose(ulong *arg)
{
```

```
273     fdtochan(arg[0], -1, 0, 0);
274     fdclose(arg[0], 0);
275
276     return 0;
277   }
278
279   long
280   unionread(Chan *c, void *va, long n)
281   {
282     int i;
283     long nr;
284     Chan *nc;
285     Mhead *m;
286     Mount *mount;
287
288     m = c->mh;
289     rlock(&m->lock);
290     mount = m->mount;
291     for(i = 0; mount != nil && i < c->uri; i++)
292             mount = mount->next;
293
294     while(mount != nil) {
295             if(waserror()) {
296                     runlock(&m->lock);
297                     nexterror();
298             }
299             if(mount->to == nil)
300                     goto next;
301             nc = cclone(mount->to, 0);
302             poperror();
303
304             /* Error causes component of union to be skipped */
305             if(waserror()) {
306                     cclose(nc);
307                     goto next;
308             }
309
310             nc = devtab[nc->type]->open(nc, OREAD);
311             nc->offset = c->offset;
312             nr = devtab[nc->type]->read(nc, va, n, nc->offset);
313             /* devdirread e.g. changes it */
314             c->offset = nc->offset;
315             poperror();
316
317             cclose(nc);
318             if(nr > 0) {
319                     runlock(&m->lock);
320                     return nr;
321             }
322             /* Advance to next element */
323     next:
324             c->uri++;
325             mount = mount->next;
326             if(mount == nil)
327                     break;
328             c->offset = 0;
329     }
330     runlock(&m->lock);
331     return 0;
332   }
333
334   long
335   sysread9p(ulong *arg)
336   {
337     int dir;
338     long n;
339     Chan *c;
340
341     validaddr(arg[1], arg[2], 1);
342     c = fdtochan(arg[0], OREAD, 1, 1);
343     if(waserror()) {
344             cclose(c);
345             nexterror();
346     }
347
348     n = arg[2];
349     dir = c->qid.path&CHDIR;
350
351     if(dir) {
352             n -= n%DIRLEN;
353             if(c->offset%DIRLEN || n==0)
354                     error(Etoosmall);
355     }
356
357     if(dir && c->mh)
358             n = unionread(c, (void*)arg[1], n);
359     else if(devtab[c->type]->dc != L'M')
360             n = devtab[c->type]->read(c, (void*)arg[1], n, c->offset);
361     else
362             n = mntread9p(c, (void*)arg[1], n, c->offset);
363
364     lock(c);
365     c->offset += n;
366     unlock(c);
367
368     poperror();
369     cclose(c);
370
371     return n;
372   }
373
374   long
375   sysread(ulong *arg)
376   {
377     int dir;
378     long n;
379     Chan *c;
380
381     n = arg[2];
382     validaddr(arg[1], n, 1);
383     c = fdtochan(arg[0], OREAD, 1, 1);
384
385     if(waserror()) {
386             cclose(c);
387             nexterror();
388     }
389
390     dir = c->qid.path&CHDIR;
391
392     if(dir) {
393             n -= n%DIRLEN;
394             if(c->offset%DIRLEN || n==0)
395                     error(Etoosmall);
396     }
397
398     if(dir && c->mh)
399             n = unionread(c, (void*)arg[1], n);
400     else
401             n = devtab[c->type]->read(c, (void*)arg[1], n, c->offset);
402
403     lock(c);
404     c->offset += n;
405     unlock(c);
406
407     poperror();
408     cclose(c);
```

```
409
410       return n;
411     }
412
413     long
414     syswrite9p(ulong *arg)
415     {
416       Chan *c;
417       long n;
418
419       validaddr(arg[1], arg[2], 0);
420       c = fdtochan(arg[0], OWRITE, 1, 1);
421       if(waserror()) {
422               cclose(c);
423               nexterror();
424       }
425
426       if(c->qid.path & CHDIR)
427               error(Eisdir);
428
429       if(devtab[c->type]->dc != L'M')
430               n = devtab[c->type]->write(c, (void*)arg[1], arg[2], c->offset);
431       else
432               n = mntwrite9p(c, (void*)arg[1], arg[2], c->offset);
433       lock(c);
434       c->offset += n;
435       unlock(c);
436
437       poperror();
438       cclose(c);
439
440       return n;
441     }
442
443     long
444     syswrite(ulong *arg)
445     {
446       Chan *c;
447       long m, n;
448       uvlong oo;
449
450       validaddr(arg[1], arg[2], 0);
451       n = arg[2];
452       c = fdtochan(arg[0], OWRITE, 1, 1);
453       if(waserror()) {
454               cclose(c);
455               lock(c);
456               c->offset -= n;
457               unlock(c);
458               nexterror();
459       }
460
461       if(c->qid.path & CHDIR)
462               error(Eisdir);
463
464       lock(c);
465       oo = c->offset;
466       c->offset += n;
467       unlock(c);
468
469       m = devtab[c->type]->write(c, (void*)arg[1], n, oo);
470
471       if(m < n){
472               lock(c);
473               c->offset -= n - m;
474               unlock(c);
475       }
476
477       poperror();
478       cclose(c);
479
480       return m;
481     }
482
483     static void
484     sseek(ulong *arg)
485     {
486       Chan *c;
487       char buf[DIRLEN];
488       Dir dir;
489       vlong off;
490       union {
491               vlong v;
492               ulong u[2];
493       } o;
494
495       c = fdtochan(arg[1], -1, 1, 1);
496       if(waserror()){
497               cclose(c);
498               nexterror();
499       }
500       if(c->qid.path & CHDIR)
501               error(Eisdir);
502
503       if(devtab[c->type]->dc == '|')
504               error(Eisstream);
505
506       off = 0;
507       o.u[0] = arg[2];
508       o.u[1] = arg[3];
509       switch(arg[4]){
510       case 0:
511               off = o.v;
512               c->offset = off;
513               break;
514
515       case 1:
516               lock(c);        /* lock for read/write update */
517               off = o.v + c->offset;
518               c->offset = off;
519               unlock(c);
520               break;
521
522       case 2:
523               devtab[c->type]->stat(c, buf);
524               convM2D(buf, &dir);
525               off = dir.length + o.v;
526               c->offset = off;
527               break;
528       }
529       *(vlong*)arg[0] = off;
530       c->uri = 0;
531       cclose(c);
532       poperror();
533     }
534
535     long
536     sysseek(ulong *arg)
537     {
538       validaddr(arg[0], BY2V, 1);
539       sseek(arg);
540       return 0;
541     }
542
543     long
544     sysoseek(ulong *arg)
```

```
{
  union {
          vlong v;
          ulong u[2];
  } o;
  ulong a[5];

  o.v = arg[1];
  a[0] = (ulong)&o.v;
  a[1] = arg[0];
  a[2] = o.u[0];
  a[3] = o.u[1];
  a[4] = arg[2];
  sseek(a);
  return o.v;
}

long
sysfstat(ulong *arg)
{
  Chan *c;

  validaddr(arg[1], DIRLEN, 1);
  evenaddr(arg[1]);
  c = fdtochan(arg[0], -1, 0, 1);
  if(waserror()) {
          cclose(c);
          nexterror();
  }
  devtab[c->type]->stat(c, (char*)arg[1]);
  poperror();
  cclose(c);
  return 0;
}

long
sysstat(ulong *arg)
{
  Chan *c;

  validaddr(arg[1], DIRLEN, 1);
  evenaddr(arg[1]);
  validaddr(arg[0], 1, 0);
  c = namec((char*)arg[0], Aaccess, 0, 0);
  if(waserror()){
          cclose(c);
          nexterror();
  }
  devtab[c->type]->stat(c, (char*)arg[1]);
  poperror();
  cclose(c);
  return 0;
}

long
syschdir(ulong *arg)
{
  Chan *c;

  validaddr(arg[0], 1, 0);

  c = namec((char*)arg[0], Atodir, 0, 0);
  cclose(up->dot);
  up->dot = c;
  return 0;
}

long
bindmount(ulong *arg, int ismount)
{
  ulong flag;
  int fd, ret;
  Chan *c0, *c1, *bc;
  struct{
          Chan    *chan;
          char    *spec;
          int     flags;
  }bogus;

  flag = arg[2];
  fd = arg[0];
  if(flag>MMASK || (flag&MORDER)==(MBEFORE|MAFTER))
          error(Ebadarg);

  bogus.flags = flag & MCACHE;

  if(ismount){
          if(up->pgrp->noattach)
                  error(Enoattach);

          bc = fdtochan(fd, ORDWR, 0, 1);
          if(waserror()) {
                  cclose(bc);
                  nexterror();
          }
          bogus.chan = bc;

          validaddr(arg[3], 1, 0);
          if(vmemchr((char*)arg[3], '\0', NAMELEN) == 0)
                  error(Ebadarg);

          bogus.spec = (char*)arg[3];
          if(waserror())
                  error(Ebadspec);
          nameok(bogus.spec, 1);
          poperror();

          ret = devno('M', 0);
          c0 = devtab[ret]->attach((char*)&bogus);

          poperror();
          cclose(bc);
  }
  else {
          bogus.spec = 0;
          validaddr(arg[0], 1, 0);
          c0 = namec((char*)arg[0], Aaccess, 0, 0);
  }

  if(waserror()){
          cclose(c0);
          nexterror();
  }

  validaddr(arg[1], 1, 0);
  c1 = namec((char*)arg[1], Amount, 0, 0);
  if(waserror()){
          cclose(c1);
          nexterror();
  }

  ret = cmount(c0, c1, flag, bogus.spec);

  poperror();
  cclose(c1);
  poperror();
```

```
681     cclose(c0);
682     if(ismount)
683             fdclose(fd, 0);
684     return ret;
685   }
686
687   long
688   sysbind(ulong *arg)
689   {
690     return bindmount(arg, 0);
691   }
692
693   long
694   sysmount(ulong *arg)
695   {
696     return bindmount(arg, 1);
697   }
698
699   long
700   sysunmount(ulong *arg)
701   {
702     Chan *cmount, *cmounted;
703
704     cmounted = 0;
705
706     validaddr(arg[1], 1, 0);
707     cmount = namec((char *)arg[1], Amount, 0, 0);
708
709     if(arg[0]) {
710             if(waserror()) {
711                     cclose(cmount);
712                     nexterror();
713             }
714             validaddr(arg[0], 1, 0);
715             cmounted = namec((char*)arg[0], Aopen, OREAD, 0);
716             poperror();
717     }
718
719     if(waserror()) {
720             cclose(cmount);
721             if(cmounted)
722                     cclose(cmounted);
723             nexterror();
724     }
725
726     cunmount(cmount, cmounted);
727     cclose(cmount);
728     if(cmounted)
729             cclose(cmounted);
730     poperror();
731     return 0;
732   }
733
734   long
735   syscreate(ulong *arg)
736   {
737     int fd;
738     Chan *c = 0;
739
740     openmode(arg[1]);       /* error check only */
741     if(waserror()) {
742             if(c)
743                     cclose(c);
744             nexterror();
745     }
746     validaddr(arg[0], 1, 0);
747     c = namec((char*)arg[0], Acreate, arg[1], arg[2]);
748     fd = newfd(c);
749     if(fd < 0)
750             error(Enofd);
751     poperror();
752     return fd;
753   }
754
755   long
756   sysremove(ulong *arg)
757   {
758     Chan *c;
759
760     validaddr(arg[0], 1, 0);
761     c = namec((char*)arg[0], Aaccess, 0, 0);
762     if(waserror()){
763             c->type = 0;    /* see below */
764             cclose(c);
765             nexterror();
766     }
767     devtab[c->type]->remove(c);
768     /*
769      * Remove clunks the fid, but we need to recover the Chan
770      * so fake it up.  rootclose() is known to be a nop.
771      */
772     c->type = 0;
773     poperror();
774     cclose(c);
775     return 0;
776   }
777
778   long
779   syswstat(ulong *arg)
780   {
781     Chan *c;
782
783     validaddr(arg[1], DIRLEN, 0);
784     nameok((char*)arg[1], 0);
785     validaddr(arg[0], 1, 0);
786     c = namec((char*)arg[0], Aaccess, 0, 0);
787     if(waserror()){
788             cclose(c);
789             nexterror();
790     }
791     devtab[c->type]->wstat(c, (char*)arg[1]);
792     poperror();
793     cclose(c);
794     return 0;
795   }
796
797   long
798   sysfwstat(ulong *arg)
799   {
800     Chan *c;
801
802     validaddr(arg[1], DIRLEN, 0);
803     nameok((char*)arg[1], 0);
804     c = fdtochan(arg[0], -1, 1, 1);
805     if(waserror()) {
806             cclose(c);
807             nexterror();
808     }
809     devtab[c->type]->wstat(c, (char*)arg[1]);
810     poperror();
811     cclose(c);
812     return 0;
813   }
```

```
#include  "u.h"
#include  "../port/lib.h"
#include  "mem.h"
#include  "dat.h"
#include  "fns.h"
#include  "../port/error.h"

#include  <a.out.h>

int       shargs(char*, int, char**);

long
sysr1(ulong *arg)
{
  print("[%s %s %lud] r1 = %lud\n", up->user, up->text, up->pid, arg[0]);
  return 0;
}

long
sysrfork(ulong *arg)
{
  Proc *p;
  int n, i;
  Fgrp *ofg;
  Pgrp *opg;
  Rgrp *org;
  Egrp *oeg;
  ulong pid, flag;
  Mach *wm;

  flag = arg[0];
  /* Check flags before we commit */
  if((flag & (RFFDG|RFCFDG)) == (RFFDG|RFCFDG))
          error(Ebadarg);
  if((flag & (RFNAMEG|RFCNAMEG)) == (RFNAMEG|RFCNAMEG))
          error(Ebadarg);
  if((flag & (RFENVG|RFCENVG)) == (RFENVG|RFCENVG))
          error(Ebadarg);

  if((flag&RFPROC) == 0) {
          if(flag & (RFMEM|RFNOWAIT))
                  error(Ebadarg);
          if(flag & (RFFDG|RFCFDG)) {
                  ofg = up->fgrp;
                  if(flag & RFFDG)
                          up->fgrp = dupfgrp(ofg);
                  else
                          up->fgrp = dupfgrp(nil);
                  closefgrp(ofg);
          }
          if(flag & (RFNAMEG|RFCNAMEG)) {
                  opg = up->pgrp;
                  up->pgrp = newpgrp();
                  if(flag & RFNAMEG)
                          pgrpcpy(up->pgrp, opg);
                  /* inherit noattach */
                  up->pgrp->noattach = opg->noattach;
                  closepgrp(opg);
          }
          if(flag & RFNOMNT)
                  up->pgrp->noattach = 1;
          if(flag & RFREND) {
                  org = up->rgrp;
                  up->rgrp = newrgrp();
                  closergrp(org);
          }
          if(flag & (RFENVG|RFCENVG)) {
                  oeg = up->egrp;
                  up->egrp = smalloc(sizeof(Egrp));
                  up->egrp->ref = 1;
                  if(flag & RFENVG)
                          envcpy(up->egrp, oeg);
                  closeegrp(oeg);
          }
          if(flag & RFNOTEG)
                  up->noteid = incref(&noteidalloc);
          return 0;
  }

  p = newproc();

  p->fpsave = up->fpsave;
  p->scallnr = up->scallnr;
  p->s = up->s;
  p->nerrlab = 0;
  p->slash = up->slash;
  p->dot = up->dot;
  incref(p->dot);

  memmove(p->note, up->note, sizeof(p->note));
  p->nnote = up->nnote;
  p->notified = 0;
  p->lastnote = up->lastnote;
  p->notify = up->notify;
  p->ureg = up->ureg;
  p->dbgreg = 0;

  /* Make a new set of memory segments */
  n = flag & RFMEM;
  qlock(&p->seglock);
  if(waserror()){
          qunlock(&p->seglock);
          nexterror();
  }
  for(i = 0; i < NSEG; i++)
          if(up->seg[i])
                  p->seg[i] = dupseg(up->seg, i, n);
  qunlock(&p->seglock);
  poperror();

  /* File descriptors */
  if(flag & (RFFDG|RFCFDG)) {
          if(flag & RFFDG)
                  p->fgrp = dupfgrp(up->fgrp);
          else
                  p->fgrp = dupfgrp(nil);
  }
  else {
          p->fgrp = up->fgrp;
          incref(p->fgrp);
  }

  /* Process groups */
  if(flag & (RFNAMEG|RFCNAMEG)) {
          p->pgrp = newpgrp();
          if(flag & RFNAMEG)
                  pgrpcpy(p->pgrp, up->pgrp);
          /* inherit noattach */
          p->pgrp->noattach = up->pgrp->noattach;
  }
  else {
          p->pgrp = up->pgrp;
          incref(p->pgrp);
  }
  if(flag & RFNOMNT)
          up->pgrp->noattach = 1;
```

```
137
138     if(flag & RFREND)
139             p->rgrp = newrgrp();
140     else {
141             incref(up->rgrp);
142             p->rgrp = up->rgrp;
143     }
144
145     /* Environment group */
146     if(flag & (RFENVG|RFCENVG)) {
147             p->egrp = smalloc(sizeof(Egrp));
148             p->egrp->ref = 1;
149             if(flag & RFENVG)
150                     envcpy(p->egrp, up->egrp);
151     }
152     else {
153             p->egrp = up->egrp;
154             incref(p->egrp);
155     }
156     p->hang = up->hang;
157     p->procmode = up->procmode;
158
159     /* Craft a return frame which will cause the child to pop out of
160      * the scheduler in user mode with the return register zero
161      */
162     forkchild(p, up->dbgreg);
163
164     p->parent = up;
165     p->parentpid = up->pid;
166     if(flag&RFNOWAIT)
167             p->parentpid = 1;
168     else {
169             lock(&up->exl);
170             up->nchild++;
171             unlock(&up->exl);
172     }
173     if((flag&RFNOTEG) == 0)
174             p->noteid = up->noteid;
175
176     p->fpstate = up->fpstate;
177     pid = p->pid;
178     memset(p->time, 0, sizeof(p->time));
179     p->time[TReal] = MACHP(0)->ticks;
180     memmove(p->text, up->text, NAMELEN);
181     memmove(p->user, up->user, NAMELEN);
182     /*
183      *  since the bss/data segments are now shareable,
184      *  any mmu info about this process is now stale
185      *  (i.e. has bad properties) and has to be discarded.
186      */
187     flushmmu();
188     p->priority = up->priority;
189     p->basepri = up->basepri;
190     p->mp = up->mp;
191     wm = up->wired;
192     if(wm)
193             procwired(p, wm->machno);
194     ready(p);
195     sched();
196     return pid;
197   }
198
199   static ulong
200   l2be(long l)
201   {
202     uchar *cp;
203
204     cp = (uchar*)&l;
205     return (cp[0]<<24) | (cp[1]<<16) | (cp[2]<<8) | cp[3];
206   }
207
208   long
209   sysexec(ulong *arg)
210   {
211     Segment *s, *ts;
212     ulong t, d, b;
213     int i;
214     Chan *tc;
215     char **argv, **argp;
216     char *a, *charp, *file;
217     char *progarg[sizeof(Exec)/2+1], elem[NAMELEN];
218     ulong ssize, spage, nargs, nbytes, n, bssend;
219     int indir;
220     Exec exec;
221     char line[sizeof(Exec)];
222     Fgrp *f;
223     Image *img;
224     ulong magic, text, entry, data, bss;
225
226     validaddr(arg[0], 1, 0);
227     file = (char*)arg[0];
228     indir = 0;
229     for(;;){
230             tc = namec(file, Aopen, OEXEC, 0);
231             if(waserror()){
232                     cclose(tc);
233                     nexterror();
234             }
235             if(!indir)
236                     strcpy(elem, up->elem);
237
238             n = devtab[tc->type]->read(tc, &exec, sizeof(Exec), 0);
239             if(n < 2)
240                     error(Ebadexec);
241             magic = l2be(exec.magic);
242             text = l2be(exec.text);
243             entry = l2be(exec.entry);
244             if(n==sizeof(Exec) && (magic == AOUT_MAGIC)){
245                     if((text&KZERO)
246                     || entry < UTZERO+sizeof(Exec)
247                     || entry >= UTZERO+sizeof(Exec)+text)
248                             error(Ebadexec);
249                     break; /* for binary */
250             }
251
252             /*
253              * Process #! /bin/sh args ...
254              */
255             memmove(line, &exec, sizeof(Exec));
256             if(indir || line[0]!='#' || line[1]!='!')
257                     error(Ebadexec);
258             n = shargs(line, n, progarg);
259             if(n == 0)
260                     error(Ebadexec);
261             indir = 1;
262             /*
263              * First arg becomes complete file name
264              */
265             progarg[n++] = file;
266             progarg[n] = 0;
267             validaddr(arg[1], BY2WD, 1);
268             arg[1] += BY2WD;
269             file = progarg[0];
270             progarg[0] = elem;
271             poperror();
272             cclose(tc);
```

```
273     }
274
275     data = l2be(exec.data);
276     bss = l2be(exec.bss);
277     t = (UTZERO+sizeof(Exec)+text+(BY2PG-1)) & ~(BY2PG-1);
278     d = (t + data + (BY2PG-1)) & ~(BY2PG-1);
279     bssend = t + data + bss;
280     b = (bssend + (BY2PG-1)) & ~(BY2PG-1);
281     if((t|d|b) & KZERO)
282             error(Ebadexec);
283
284     /*
285      * Args: pass 1: count
286      */
287     nbytes = BY2WD;         /* hole for profiling clock at top of stack */
288     nargs = 0;
289     if(indir){
290             argp = progarg;
291             while(*argp){
292                     a = *argp++;
293                     nbytes += strlen(a) + 1;
294                     nargs++;
295             }
296     }
297     evenaddr(arg[1]);
298     argp = (char**)arg[1];
299     validaddr((ulong)argp, BY2WD, 0);
300     while(*argp){
301             a = *argp++;
302             if(((ulong)argp&(BY2PG-1)) < BY2WD)
303                     validaddr((ulong)argp, BY2WD, 0);
304             validaddr((ulong)a, 1, 0);
305             nbytes += (vmemchr(a, 0, 0x7FFFFFFF) - a) + 1;
306             nargs++;
307     }
308     ssize = BY2WD*(nargs+1) + ((nbytes+(BY2WD-1)) & ~(BY2WD-1));
309
310     /*
311      * 8-byte align SP for those (e.g. sparc) that need it.
312      * execregs() will subtract another 4 bytes for argc.
313      */
314     if((ssize+4) & 7)
315             ssize += 4;
316     spage = (ssize+(BY2PG-1)) >> PGSHIFT;
317
318     /*
319      * Build the stack segment, putting it in kernel virtual for the moment
320      */
321     if(spage > TSTKSIZ)
322             error(Enovmem);
323
324     qlock(&up->seglock);
325     if(waserror()){
326             qunlock(&up->seglock);
327             nexterror();
328     }
329     up->seg[ESEG] = newseg(SG_STACK, TSTKTOP-USTKSIZE, USTKSIZE/BY2PG);
330
331     /*
332      * Args: pass 2: assemble; the pages will be faulted in
333      */
334     argv = (char**)(TSTKTOP - ssize);
335     charp = (char*)(TSTKTOP - nbytes);
336     if(indir)
337             argp = progarg;
338     else
339             argp = (char**)arg[1];
340
341     for(i=0; i<nargs; i++){
342             if(indir && *argp==0) {
343                     indir = 0;
344                     argp = (char**)arg[1];
345             }
346             *argv++ = charp + (USTKTOP-TSTKTOP);
347             n = strlen(*argp) + 1;
348             memmove(charp, *argp++, n);
349             charp += n;
350     }
351
352     memmove(up->text, elem, NAMELEN);
353
354     /*
355      * Committed.
356      * Free old memory.
357      * Special segments are maintained across exec
358      */
359     for(i = SSEG; i <= BSEG; i++) {
360             putseg(up->seg[i]);
361             /* prevent a second free if we have an error */
362             up->seg[i] = 0;
363     }
364     for(i = BSEG+1; i < NSEG; i++) {
365             s = up->seg[i];
366             if(s != 0 && (s->type&SG_CEXEC)) {
367                     putseg(s);
368                     up->seg[i] = 0;
369             }
370     }
371
372     /*
373      * Close on exec
374      */
375     f = up->fgrp;
376     for(i=0; i<=f->maxfd; i++)
377             fdclose(i, CCEXEC);
378
379     /* Text.  Shared. Attaches to cache image if possible */
380     /* attachimage returns a locked cache image */
381     img = attachimage(SG_TEXT|SG_RONLY, tc, UTZERO, (t-UTZERO)>>PGSHIFT);
382     ts = img->s;
383     up->seg[TSEG] = ts;
384     ts->flushme = 1;
385     ts->fstart = 0;
386     ts->flen = sizeof(Exec)+text;
387     unlock(img);
388
389     /* Data. Shared. */
390     s = newseg(SG_DATA, t, (d-t)>>PGSHIFT);
391     up->seg[DSEG] = s;
392
393     /* Attached by hand */
394     incref(img);
395     s->image = img;
396     s->fstart = ts->fstart+ts->flen;
397     s->flen = data;
398
399     /* BSS. Zero fill on demand */
400     up->seg[BSEG] = newseg(SG_BSS, d, (b-d)>>PGSHIFT);
401
402     /*
403      * Move the stack
404      */
405     s = up->seg[ESEG];
406     up->seg[ESEG] = 0;
407     up->seg[SSEG] = s;
408     qunlock(&up->seglock);
```

```
409     poperror();
410     s->base = USTKTOP-USTKSIZE;
411     s->top = USTKTOP;
412     relocateseg(s, USTKTOP-TSTKTOP);
413
414     /*
415      *  '/' processes are higher priority (hack to make /ip more responsive).
416      */
417     if(devtab[tc->type]->dc == L'/')
418             up->basepri = PriRoot;
419     up->priority = up->basepri;
420     poperror();
421     cclose(tc);
422
423     /*
424      *  At this point, the mmu contains info about the old address
425      *  space and needs to be flushed
426      */
427     flushmmu();
428     qlock(&up->debug);
429     up->nnote = 0;
430     up->notify = 0;
431     up->notified = 0;
432     procsetup(up);
433     qunlock(&up->debug);
434     if(up->hang)
435             up->procctl = Proc_stopme;
436
437     return execregs(entry, ssize, nargs);
438   }
439
440   int
441   shargs(char *s, int n, char **ap)
442   {
443     int i;
444
445     s += 2;
446     n -= 2;         /* skip #! */
447     for(i=0; s[i]!='\n'; i++)
448             if(i == n-1)
449                     return 0;
450     s[i] = 0;
451     *ap = 0;
452     i = 0;
453     for(;;) {
454             while(*s==' ' || *s=='\t')
455                     s++;
456             if(*s == 0)
457                     break;
458             i++;
459             *ap++ = s;
460             *ap = 0;
461             while(*s && *s!=' ' && *s!='\t')
462                     s++;
463             if(*s == 0)
464                     break;
465             else
466                     *s++ = 0;
467     }
468     return i;
469   }
470
471   int
472   return0(void*)
473   {
474     return 0;
475   }
476
```

```
477   long
478   syssleep(ulong *arg)
479   {
480
481     int n;
482
483     n = arg[0];
484     if(n <= 0) {
485             up->priority = 0;
486             sched();
487             return 0;
488     }
489     if(n < TK2MS(1))
490             n = TK2MS(1);
491     tsleep(&up->sleep, return0, 0, n);
492     return 0;
493   }
494
495   long
496   sysalarm(ulong *arg)
497   {
498     return procalarm(arg[0]);
499   }
500
501   long
502   sysexits(ulong *arg)
503   {
504     char *status;
505     char *inval = "invalid exit string";
506     char buf[ERRLEN];
507
508     status = (char*)arg[0];
509     if(status){
510             if(waserror())
511                     status = inval;
512             else{
513                     validaddr((ulong)status, 1, 0);
514                     if(vmemchr(status, 0, ERRLEN) == 0){
515                             memmove(buf, status, ERRLEN);
516                             buf[ERRLEN-1] = 0;
517                             status = buf;
518                     }
519             }
520             poperror();
521
522     }
523     pexit(status, 1);
524     return 0;               /* not reached */
525   }
526
527   long
528   syswait(ulong *arg)
529   {
530     if(arg[0]){
531             validaddr(arg[0], sizeof(Waitmsg), 1);
532             evenaddr(arg[0]);
533     }
534     return pwait((Waitmsg*)arg[0]);
535   }
536
537   long
538   sysdeath(ulong*)
539   {
540     pprint("deprecated system call\n");
541     pexit("Suicide", 0);
542     return 0;       /* not reached */
543   }
544
```

```
void
werrstr(char *fmt, ...)
{
  va_list va;

  if(up == nil)
          return;

  va_start(va, fmt);
  doprint(up->error, up->error+ERRLEN, fmt, va);
  va_end(va);
}

long
syserrstr(ulong *arg)
{
  char *e, tmp[ERRLEN];

  validaddr(arg[0], ERRLEN, 1);
  e = (char*)arg[0];
  memmove(tmp, e, ERRLEN);
  memmove(e, up->error, ERRLEN);
  memmove(up->error, tmp, ERRLEN);
  return 0;
}

long
sysnotify(ulong *arg)
{
  if(arg[0] != 0)
          validaddr(arg[0], sizeof(ulong), 0);
  up->notify = (int(*)(void*, char*))(arg[0]);
  return 0;
}

long
sysnoted(ulong *arg)
{
  if(arg[0]!=NRSTR && !up->notified)
          error(Egreg);
  return 0;
}

long
syssegbrk(ulong *arg)
{
  int i;
  ulong addr;
  Segment *s;

  addr = arg[0];
  for(i = 0; i < NSEG; i++) {
          s = up->seg[i];
          if(s == 0 || addr < s->base || addr >= s->top)
                  continue;
          switch(s->type&SG_TYPE) {
          case SG_TEXT:
          case SG_DATA:
          case SG_STACK:
                  error(Ebadarg);
          default:
                  return ibrk(arg[1], i);
          }
  }

  error(Ebadarg);
  return 0;               /* not reached */
}

long
syssegattach(ulong *arg)
{
  return segattach(up, arg[0], (char*)arg[1], arg[2], arg[3]);
}

long
syssegdetach(ulong *arg)
{
  int i;
  ulong addr;
  Segment *s;

  qlock(&up->seglock);
  if(waserror()){
          qunlock(&up->seglock);
          nexterror();
  }

  s = 0;
  addr = arg[0];
  for(i = 0; i < NSEG; i++)
          if(s = up->seg[i]) {
                  qlock(&s->lk);
                  if((addr >= s->base && addr < s->top) ||
                     (s->top == s->base && addr == s->base))
                          goto found;
                  qunlock(&s->lk);
          }

  error(Ebadarg);

found:
  /* Check we are not detaching the current stack segment */
  if((ulong)arg >= s->base && (ulong)arg < s->top) {
          qunlock(&s->lk);
          error(Ebadarg);
  }
  up->seg[i] = 0;
  qunlock(&s->lk);
  putseg(s);
  qunlock(&up->seglock);
  poperror();

  /* Ensure we flush any entries from the lost segment */
  flushmmu();
  return 0;
}

long
syssegfree(ulong *arg)
{
  Segment *s;
  ulong from, pages;

  from = PGROUND(arg[0]);
  s = seg(up, from, 1);
  if(s == 0)
          error(Ebadarg);

  pages = (arg[1]+BY2PG-1)/BY2PG;

  if(from+pages*BY2PG > s->top) {
          qunlock(&s->lk);
          error(Ebadarg);
  }
```

```
681     mfreeseg(s, from, pages);
682     qunlock(&s->lk);
683     flushmmu();
684
685     return 0;
686   }
687
688   /* For binary compatibility */
689   long
690   sysbrk_(ulong *arg)
691   {
692     return ibrk(arg[0], BSEG);
693   }
694
695   long
696   sysrendezvous(ulong *arg)
697   {
698     ulong tag;
699     ulong val;
700     Proc *p, **l;
701
702     tag = arg[0];
703     l = &REND(up->rgrp, tag);
704
705     lock(up->rgrp);
706     for(p = *l; p; p = p->rendhash) {
707             if(p->rendtag == tag) {
708                     *l = p->rendhash;
709                     val = p->rendval;
710                     p->rendval = arg[1];
711
712                     while(p->mach != 0)
713                             ;
714                     ready(p);
715                     unlock(up->rgrp);
716                     return val;
717             }
718             l = &p->rendhash;
719     }
720
721     /* Going to sleep here */
722     up->rendtag = tag;
723     up->rendval = arg[1];
724     up->rendhash = *l;
725     *l = up;
726     up->state = Rendezvous;
727     unlock(up->rgrp);
728
729     sched();
730
731     return up->rendval;
732   }
```

```
#include "u.h"
#include "../port/lib.h"
#include "mem.h"
#include "dat.h"
#include "fns.h"
#include "../port/error.h"

struct {
  ulong   locks;
  ulong   glare;
  ulong   inglare;
} lockstats;

void
lockloop(Lock *l, ulong pc)
{
  Proc *p;

  p = l->p;
  print("lock loop key 0x%lux pc 0x%lux held by pc 0x%lux proc %lud\n",
          l->key, pc, l->pc, p ? p->pid : 0);
  dumpaproc(up);
  if(p != nil)
          dumpaproc(p);

  if(up && up->state == Running && islo())
          sched();
}

void
lock(Lock *l)
{
  int i, cansched;
  ulong pc, oldpri;

  pc = getcallerpc(&l);

  lockstats.locks++;
  if(tas(&l->key) == 0){
          l->pc = pc;
          l->p = up;
          l->isilock = 0;
          return;
  }

  lockstats.glare++;
  cansched = up != nil && up->state == Running;
  if(cansched){
          oldpri = up->priority;
          up->lockwait = 1;
          up->priority = PriLock;
  } else
          oldpri = 0;

  for(;;){
          lockstats.inglare++;
          i = 0;
          while(l->key){
                  if(conf.nmach < 2 && cansched){
                          if(i++ > 1000){
                                  i = 0;
                                  lockloop(l, pc);
                          }
                          sched();
                  } else {
                          if(i++ > 100000000){
                                  i = 0;
                                  lockloop(l, pc);
                          }
                  }
          }
          if(tas(&l->key) == 0){
                  l->pc = pc;
                  l->p = up;
                  l->isilock = 0;
                  if(cansched){
                          up->lockwait = 0;
                          up->priority = oldpri;
                  }
                  return;
          }
  }
}

void
ilock(Lock *l)
{
  ulong x;
  ulong pc, oldpri;
  int cansched;

  pc = getcallerpc(&l);
  lockstats.locks++;

  x = splhi();
  if(tas(&l->key) == 0){
          l->sr = x;
          l->pc = pc;
          l->p = up;
          l->isilock = 1;
          return;
  }

  lockstats.glare++;
  cansched = up != nil && up->state == Running;
  if(cansched){
          oldpri = up->priority;
          up->lockwait = 1;
          up->priority = PriLock;
  } else
          oldpri = 0;
  if(conf.nmach < 2)
          panic("ilock: no way out: pc %uX\n", pc);

  for(;;){
          lockstats.inglare++;
          splx(x);
          while(l->key)
                  ;
          x = splhi();
          if(tas(&l->key) == 0){
                  l->sr = x;
                  l->pc = pc;
                  l->p = up;
                  l->isilock = 1;
                  if(cansched){
                          up->lockwait = 0;
                          up->priority = oldpri;
                  }
                  return;
          }
  }
}

int
canlock(Lock *l)
```

```
137  {
138    if(tas(&l->key))
139            return 0;
140
141    l->pc = getcallerpc(&l);
142    l->p = up;
143    l->isilock = 0;
144    return 1;
145  }
146
147  void
148  unlock(Lock *l)
149  {
150
151    if(l->key == 0)
152            print("unlock: not locked: pc %luX\n", getcallerpc(&l));
153    if(l->isilock)
154            print("unlock of ilock: pc %lux, held by %lux\n", getcallerpc(&l), l->pc);
155    l->pc = 0;
156    l->key = 0;
157    coherence();
158  }
159
160  void
161  iunlock(Lock *l)
162  {
163    ulong sr;
164
165    if(l->key == 0)
166            print("iunlock: not locked: pc %luX\n", getcallerpc(&l));
167    if(!l->isilock)
168            print("iunlock of lock: pc %lux, held by %lux\n", getcallerpc(&l), l->pc);
169
170    sr = l->sr;
171    l->pc = 0;
172    l->key = 0;
173    coherence();
174
175    m->splpc = getcallerpc(&l);
176    splxpc(sr);
177  }
```

```
#include  "u.h"
#include  "../port/lib.h"
#include  "mem.h"
#include  "dat.h"
#include  "fns.h"
#include  "../port/error.h"

// compute nanosecond epoch time from the fastest ticking clock
// on the system.  converting the time to nanoseconds requires
// the following formula
//
//        t = (((1000000000<<31)/f)*ticks)>>31
//
//  where
//
//        'f'             is the clock frequency
//        'ticks'         are clock ticks
//
//  to avoid too much calculation in gettod(), we calculate
//
//        mult = (1000000000<<31)/f
//
//  each time f is set.  f is normally set by a user level
//  program writing to /dev/fastclock.


// frequency of the tod clock
#define TODFREQ   1000000000LL

struct {
  Lock;
  vlong   multiplier;     // t = off + (multiplier*ticks)>>31
  vlong   hz;             // frequency of fast clock
  vlong   last;           // last reading of fast clock
  vlong   off;            // offset from epoch to last
  vlong   lasttime;       // last return value from gettod
  vlong   delta;          // add 'delta' each slow clock tick from sstart to send
  ulong   sstart;         // ...
  ulong   send;           // ...
} tod;

void
todinit(void)
{
  fastticks((uvlong*)&tod.hz);
  todsetfreq(tod.hz);
  addclock0link(todfix);
}

//
//  calculate multiplier
//
void
todsetfreq(vlong f)
{
  ilock(&tod);
  tod.hz = f;
  tod.multiplier = (TODFREQ<<31)/f;
  iunlock(&tod);
}

//
//  Set the time of day struct
//
void
todset(vlong t, vlong delta, int n)
{
  ilock(&tod);
  if(t >= 0){
        tod.off = t;
        tod.last = fastticks(nil);
        tod.lasttime = 0;
        tod.delta = 0;
        tod.sstart = tod.send;
  } else {
        if(n <= 0)
                n = 1;
        n *= HZ;
        if(delta < 0 && n > -delta)
                n = -delta;
        if(delta > 0 && n > delta)
                n = delta;
        delta = delta/n;
        tod.sstart = MACHP(0)->ticks;
        tod.send = tod.sstart + n;
        tod.delta = delta;
  }
  iunlock(&tod);
}

//
//  get time of day
//
vlong
todget(vlong *ticksp)
{
  uvlong x;
  vlong ticks, diff;
  ulong t;

  if(tod.hz == 0)
        ticks = fastticks((uvlong*)&tod.hz);
  else
        ticks = fastticks(nil);
  diff = ticks - tod.last;

  // convert to epoch
  x = (diff * tod.multiplier) >> 31;
  x = x + tod.off;

  if(m->machno == 0){
        ilock(&tod);

        // add in correction
        if(tod.sstart != tod.send){
                t = MACHP(0)->ticks;
                if(t >= tod.send)
                        t = tod.send;
                tod.off = tod.off + tod.delta*(t - tod.sstart);
                tod.sstart = t;
        }

        // protect against overflows
        if(diff > tod.hz){
                tod.last = ticks;
                tod.off = x;
        }
        iunlock(&tod);
  }

  // time can't go backwards
  if(x < tod.lasttime)
        x = tod.lasttime;
  else
        tod.lasttime = x;

```

```
137     if(ticksp != nil)
138             *ticksp = ticks;
139
140     return x;
141   }
142
143   //
144   //  called every clock tick
145   //
146   void
147   todfix(void)
148   {
149     static ulong last;
150
151     // once a second, make sure we don't overflow
152     if(MACHP(0)->ticks - last >= HZ){
153             last = MACHP(0)->ticks;
154             todget(nil);
155     }
156   }
157
158   long
159   seconds(void)
160   {
161     vlong x;
162     int i;
163
164     x = todget(nil);
165     x = x/TODFREQ;
166     i = x;
167     return i;
168   }
```

```
#include "u.h"
#include "../port/lib.h"
#include "mem.h"
#include "dat.h"
#include "fns.h"

#define datoff            ((ulong)((Xhdr*)0)->data)

enum
{
  Chunk           = 64*1024,
  Nhole           = 128,
  Magichole       = 0x484F4C45,                   /* HOLE */
};

typedef struct Hole Hole;
typedef struct Xalloc Xalloc;
typedef struct Xhdr Xhdr;

struct Hole
{
  ulong   addr;
  ulong   size;
  ulong   top;
  Hole*   link;
};

struct Xhdr
{
  ulong   size;
  ulong   magix;
  char    data[1];
};

struct Xalloc
{
  Lock;
  Hole    hole[Nhole];
  Hole*   flist;
  Hole*   table;
};

static Xalloc     xlists;

void
xinit(void)
{
  Hole *h, *eh;
  int upages, np0, np1;

  eh = &xlists.hole[Nhole-1];
  for(h = xlists.hole; h < eh; h++)
          h->link = h+1;

  xlists.flist = xlists.hole;

  upages = conf.upages;
  np1 = upages;
  if(np1 > conf.npage1)
          np1 = conf.npage1;

  palloc.p1 = conf.base1 + (conf.npage1 - np1)*BY2PG;
  conf.npage1 -= np1;
  xhole(conf.base1, conf.npage1*BY2PG);
  conf.npage1 = conf.base1+(conf.npage1*BY2PG);
  upages -= np1;

  np0 = upages;
  if(np0 > conf.npage0)
          np0 = conf.npage0;

  palloc.p0 = conf.base0 + (conf.npage0 - np0)*BY2PG;
  conf.npage0 -= np0;
  xhole(conf.base0, conf.npage0*BY2PG);
  conf.npage0 = conf.base0+(conf.npage0*BY2PG);

  palloc.np0 = np0;
  palloc.np1 = np1;
  /* Save the bounds of kernel alloc memory for kernel mmu mapping */
  conf.base0 = (ulong)KADDR(conf.base0);
  conf.base1 = (ulong)KADDR(conf.base1);
  conf.npage0 = (ulong)KADDR(conf.npage0);
  conf.npage1 = (ulong)KADDR(conf.npage1);
}

void*
xspanalloc(ulong size, int align, ulong span)
{
  ulong a, v, t;

  a = (ulong)xalloc(size+align+span);
  if(a == 0)
          panic("xspanalloc: %d %d %lux\n", size, align, span);

  if(span > 2) {
          v = (a + span) & ~(span-1);
          t = v - a;
          if(t > 0)
                  xhole(PADDR(a), t);
          t = a + span - v;
          if(t > 0)
                  xhole(PADDR(v+size+align), t);
  }
  else
          v = a;

  if(align > 1)
          v = (v + align) & ~(align-1);

  return (void*)v;
}

void*
xallocz(ulong size, int zero)
{
  Xhdr *p;
  Hole *h, **l;

  size += BY2V + sizeof(Xhdr);
  size &= ~(BY2V-1);

  ilock(&xlists);
  l = &xlists.table;
  for(h = *l; h; h = h->link) {
          if(h->size >= size) {
                  p = (Xhdr*)h->addr;
                  h->addr += size;
                  h->size -= size;
                  if(h->size == 0) {
                          *l = h->link;
                          h->link = xlists.flist;
                          xlists.flist = h;
                  }
                  iunlock(&xlists);
                  p = KADDR(p);
                  p->magix = Magichole;
```

```
137                     p->size = size;
138                     if(zero)
139                             memset(p->data, 0, size);
140                     return p->data;
141             }
142             l = &h->link;
143     }
144     iunlock(&xlists);
145     return nil;
146   }
147
148   void*
149   xalloc(ulong size)
150   {
151     return xallocz(size, 1);
152   }
153
154   void
155   xfree(void *p)
156   {
157     Xhdr *x;
158
159     x = (Xhdr*)((ulong)p - datoff);
160     if(x->magix != Magichole) {
161             xsummary();
162             panic("xfree(0x%lux) 0x%lux!=0x%lux", p, Magichole, x->magix);
163     }
164     xhole(PADDR(x), x->size);
165   }
166
167   int
168   xmerge(void *vp, void *vq)
169   {
170     Xhdr *p, *q;
171
172     p = vp;
173     if((uchar*)vp+p->size == (uchar*)vq) {
174             q = vq;
175             p->size += q->size;
176             return 1;
177     }
178     return 0;
179   }
180
181   void
182   xhole(ulong addr, ulong size)
183   {
184     ulong top;
185     Hole *h, *c, **l;
186
187     if(size == 0)
188             return;
189
190     top = addr + size;
191     ilock(&xlists);
192     l = &xlists.table;
193     for(h = *l; h; h = h->link) {
194             if(h->top == addr) {
195                     h->size += size;
196                     h->top = h->addr+h->size;
197                     c = h->link;
198                     if(c && h->top == c->addr) {
199                             h->top += c->size;
200                             h->size += c->size;
201                             h->link = c->link;
202                             c->link = xlists.flist;
203                             xlists.flist = c;
204                     }
205                     iunlock(&xlists);
206                     return;
207             }
208             if(h->addr > addr)
209                     break;
210             l = &h->link;
211     }
212     if(h && top == h->addr) {
213             h->addr -= size;
214             h->size += size;
215             iunlock(&xlists);
216             return;
217     }
218
219     if(xlists.flist == nil) {
220             iunlock(&xlists);
221             print("xfree: no free holes, leaked %lud bytes\n", size);
222             return;
223     }
224
225     h = xlists.flist;
226     xlists.flist = h->link;
227     h->addr = addr;
228     h->top = top;
229     h->size = size;
230     h->link = *l;
231     *l = h;
232     iunlock(&xlists);
233   }
234
235   void
236   xsummary(void)
237   {
238     int i;
239     Hole *h;
240
241     i = 0;
242     for(h = xlists.flist; h; h = h->link)
243             i++;
244
245     print("%d holes free\n", i);
246     i = 0;
247     for(h = xlists.table; h; h = h->link) {
248             print("%.8lux %.8lux %lud\n", h->addr, h->top, h->size);
249             i += h->size;
250     }
251     print("%d bytes free\n", i);
252   }
```

```
#include "u.h"
#include "lib.h"
#include "mem.h"
#include "dat.h"
#include "fns.h"
#include "io.h"

static int
addbytes(char **dbuf, char *edbuf, char **sbuf, char *esbuf)
{
  int n;

  n = edbuf - *dbuf;
  if(n <= 0)
          return 0;
  if(n > esbuf - *sbuf)
          n = esbuf - *sbuf;
  if(n <= 0)
          return -1;

  memmove(*dbuf, *sbuf, n);
  *sbuf += n;
  *dbuf += n;
  return edbuf - *dbuf;
}

int
bootpass(Boot *b, void *vbuf, int nbuf)
{
  char *buf, *ebuf;
  Exec *ep;
  ulong entry, data, text, bss;

  if(b->state == FAILED)
          return FAIL;

  if(nbuf == 0)
          goto Endofinput;

  buf = vbuf;
  ebuf = buf+nbuf;
  while(addbytes(&b->wp, b->ep, &buf, ebuf) == 0) {
          switch(b->state) {
          case INITKERNEL:
                  b->state = READEXEC;
                  b->bp = (char*)&b->exec;
                  b->wp = b->bp;
                  b->ep = b->bp+sizeof(Exec);
                  break;
          case READEXEC:
                  ep = &b->exec;
                  if(GLLONG(ep->magic) == I_MAGIC) {
                          b->state = READTEXT;
                          b->bp = (char*)PADDR(GLLONG(ep->entry));
                          b->wp = b->bp;
                          b->ep = b->wp+GLLONG(ep->text);
                          print("%ld", GLLONG(ep->text));
                          break;
                  }

                  /* check for gzipped kernel */
                  if(b->bp[0] == 0x1F && (uchar)b->bp[1] == 0x8B && b->bp[2] == 0x08) {
                          b->state = READGZIP;
                          b->bp = (char*)malloc(1024*1024);
                          b->wp = b->bp;
                          b->ep = b->wp + 1024*1024;
                          memmove(b->bp, &b->exec, sizeof(Exec));
                          b->wp += sizeof(Exec);
                          print("gz...");
                          break;
                  }

                  print("bad kernel format\n");
                  b->state = FAILED;
                  return FAIL;

          case READTEXT:
                  ep = &b->exec;
                  b->state = READDATA;
                  b->bp = (char*)PGROUND(GLLONG(ep->entry)+GLLONG(ep->text));
                  b->wp = b->bp;
                  b->ep = b->wp + GLLONG(ep->data);
                  print("+%ld", GLLONG(ep->data));
                  break;

          case READDATA:
                  ep = &b->exec;
                  bss = GLLONG(ep->bss);
                  print("+%ld=%ld\n",
                          bss, GLLONG(ep->text)+GLLONG(ep->data)+bss);
                  b->state = TRYBOOT;
                  return ENOUGH;

          case TRYBOOT:
          case READGZIP:
                  return ENOUGH;

          case READ9LOAD:
          case INIT9LOAD:
                  panic("9load");

          default:
                  panic("bootstate");
          }
  }
  return MORE;


Endofinput:
  /* end of input */
  switch(b->state) {
  case INITKERNEL:
  case READEXEC:
  case READDATA:
  case READTEXT:
          print("premature EOF\n");
          b->state = FAILED;
          return FAIL;

  case TRYBOOT:
          entry = GLLONG(b->exec.entry);
          print("entry: 0x%lux\n", entry);
          warp9(PADDR(entry));
          b->state = FAILED;
          return FAIL;

  case READGZIP:
          ep = &b->exec;
          if(b->bp[0] != 0x1F || (uchar)b->bp[1] != 0x8B || b->bp[2] != 0x08)
                  print("lost magic\n");

          print("%ld => ", b->wp - b->bp);
          if(gunzip((uchar*)ep, sizeof(*ep), (uchar*)b->bp, b->wp - b->bp) < sizeof(*ep)) {
                  print("badly compressed kernel\n");
                  return FAIL;
          }
```

```
137             entry = GLLONG(ep->entry);
138             text = GLLONG(ep->text);
139             data = GLLONG(ep->data);
140             bss = GLLONG(ep->bss);
141             print("%lud+%lud+%lud=%lud\n", text, data, bss, text+data+bss);
142
143             if(gunzip((uchar*)PADDR(entry)-sizeof(Exec), sizeof(Exec)+text+data,
144                  (uchar*)b->bp, b->wp-b->bp) < sizeof(Exec)+text+data) {
145                    print("error uncompressing kernel\n");
146                    return FAIL;
147             }
148
149             /* relocate data to start at page boundary */
150             memmove((void*)PGROUND(entry+text), (void*)PADDR(entry+text), data);
151
152             print("entry: %lux\n", entry);
153             warp9(PADDR(entry));
154             b->state = FAILED;
155             return FAIL;
156
157     case INIT9LOAD:
158     case READ9LOAD:
159             panic("end 9load");
160
161     default:
162             panic("bootdone");
163     }
164     b->state = FAILED;
165     return FAIL;
166   }
```

```
#include  "u.h"
#include  "lib.h"
#include  "mem.h"
#include  "dat.h"
#include  "fns.h"
#include  "io.h"
#include  "ureg.h"

/*
 *  8253 timer
 */
enum
{
  T0cntr= 0x40,           /* counter ports */
  T1cntr= 0x41,           /* ... */
  T2cntr= 0x42,           /* ... */
  Tmode=  0x43,           /* mode port */

  /* commands */
  Latch0= 0x00,           /* latch counter 0's value */
  Load0=  0x30,           /* load counter 0 with 2 bytes */

  /* modes */
  Square= 0x36,           /* perioic square wave */

  Freq=   1193182,        /* Real clock frequency */
};

static int cpufreq = 66000000;
static int cpumhz = 66;
static int loopconst = 100;
static int cpuidax, cpuiddx;

static void
clockintr(Ureg*, void*)
{
  m->ticks++;
  checkalarms();
}

#define STEPPING(x)       ((x)&0xf)
#define X86MODEL(x)       (((x)>>4)&0xf)
#define X86FAMILY(x)      (((x)>>8)&0xf)

enum
{
  /* flags */
  CpuidFPU        = 0x001,        /* on-chip floating point unit */
  CpuidMCE        = 0x080,        /* machine check exception */
  CpuidCX8        = 0x100,        /* CMPXCHG8B instruction */
};

typedef struct
{
  int family;
  int model;
  int aalcycles;
  char *name;
} X86type;

X86type x86intel[] =
{
  { 4,    0,      22,     "486DX", },     /* known chips */
  { 4,    1,      22,     "486DX50", },
  { 4,    2,      22,     "486SX", },
  { 4,    3,      22,     "486DX2", },
  { 4,    4,      22,     "486SL", },
  { 4,    5,      22,     "486SX2", },
  { 4,    7,      22,     "DX2WB", },     /* P24D */
  { 4,    8,      22,     "DX4", },       /* P24C */
  { 4,    9,      22,     "DX4WB", },     /* P24CT */
  { 5,    0,      23,     "P5", },
  { 5,    1,      23,     "P5", },
  { 5,    2,      23,     "P54C", },
  { 5,    3,      23,     "P24T", },
  { 5,    4,      23,     "P55C MMX", },
  { 5,    7,      23,     "P54C VRT", },
  { 6,    1,      16,     "PentiumPro", },/* trial and error */
  { 6,    3,      16,     "PentiumII", },
  { 6,    5,      16,     "PentiumII/Xeon", },
  { 6,    6,      16,     "Celeron", },
  { 6,    7,      16,     "PentiumIII/Xeon", },
  { 6,    8,      16,     "PentiumIII/Xeon", },

  { 3,    -1,     32,     "386", },       /* family defaults */
  { 4,    -1,     22,     "486", },
  { 5,    -1,     23,     "Pentium", },
  { 6,    -1,     16,     "PentiumPro", },

  { -1,   -1,     23,     "unknown", },   /* total default */
};


/*
 * The AMD processors all implement the CPUID instruction.
 * The later ones also return the processor name via functions
 * 0x80000002, 0x80000003 and 0x80000004 in registers AX, BX, CX
 * and DX:
 *        K5      "AMD-K5(tm) Processor"
 *        K6      "AMD-K6tm w/ multimedia extensions"
 *        K6 3D   "AMD-K6(tm) 3D processor"
 *        K6 3D+  ?
 */
static X86type x86amd[] =
{
  { 5,    0,      23,     "AMD-K5", },    /* guesswork */
  { 5,    1,      23,     "AMD-K5", },    /* guesswork */
  { 5,    2,      23,     "AMD-K5", },    /* guesswork */
  { 5,    3,      23,     "AMD-K5", },    /* guesswork */
  { 5,    6,      11,     "AMD-K6", },    /* trial and error */
  { 5,    7,      11,     "AMD-K6", },    /* trial and error */
  { 5,    8,      11,     "AMD-K6-2", },  /* trial and error */
  { 5,    9,      11,     "AMD-K6-III", },/* trial and error */

  { 6,    1,      11,     "AMD-Athlon", },/* trial and error */
  { 6,    2,      11,     "AMD-Athlon", },/* trial and error */

  { 4,    -1,     22,     "Am486", },     /* guesswork */
  { 5,    -1,     23,     "AMD-K5/K6", }, /* guesswork */
  { 6,    -1,     11,     "AMD-Athlon", },/* guesswork */

  { -1,   -1,     23,     "unknown", },   /* total default */
};

static X86type    *cputype;

/*
 *  delay for l milliseconds more or less.  delayloop is set by
 *  clockinit() to match the actual CPU speed.
 */
void
delay(int l)
{
  l *= loopconst;
  if(l <= 0)
          l = 1;
```

```
137     aamloop(l);
138   }
139
140   void
141   microdelay(int l)
142   {
143     l *= loopconst;
144     l /= 1000;
145     if(l <= 0)
146             l = 1;
147     aamloop(l);
148   }
149
150   extern void cpuid(char*, int*, int*);
151
152   X86type*
153   cpuidentify(void)
154   {
155     int family, model;
156     X86type *t;
157     char cpuidid[16];
158     int cpuidax, cpuiddx;
159
160     cpuid(cpuidid, &cpuidax, &cpuiddx);
161     if(strncmp(cpuidid, "AuthenticAMD", 12) == 0)
162             t = x86amd;
163     else
164             t = x86intel;
165     family = X86FAMILY(cpuidax);
166     model = X86MODEL(cpuidax);
167     while(t->name){
168             if((t->family == family && t->model == model)
169             || (t->family == family && t->model == -1)
170             || (t->family == -1))
171                     break;
172             t++;
173     }
174     if(t->name == nil)
175             panic("cpuidentify");
176
177     return t;
178   }
179
180   void
181   clockinit(void)
182   {
183     int x, y;       /* change in counter */
184     int loops, incr;
185     X86type *t;
186
187     /*
188      *  set vector for clock interrupts
189      */
190     setvec(VectorCLOCK, clockintr, 0);
191
192     t = cpuidentify();
193
194     /*
195      *  set clock for 1/HZ seconds
196      */
197     outb(Tmode, Load0|Square);
198     outb(T0cntr, (Freq/HZ));        /* low byte */
199     outb(T0cntr, (Freq/HZ)>>8);     /* high byte */
200
201     /*
202      * Introduce a little delay to make sure the count is
203      * latched and the timer is counting down; with a fast
204      * enough processor this may not be the case.
205      * The i8254 (which this probably is) has a read-back
206      * command which can be used to make sure the counting
207      * register has been written into the counting element.
208      */
209     x = (Freq/HZ);
210     for(loops = 0; loops < 100000 && x >= (Freq/HZ); loops++){
211             outb(Tmode, Latch0);
212             x = inb(T0cntr);
213             x |= inb(T0cntr)<<8;
214     }
215
216     /* find biggest loop that doesn't wrap */
217     incr = 16000000/(t->aalcycles*HZ*2);
218     x = 2000;
219     for(loops = incr; loops < 64*1024; loops += incr) {
220
221             /*
222              *  measure time for the loop
223              *
224              *                      MOVL    loops,CX
225              *      aaml1:          AAM
226              *                      LOOP    aaml1
227              *
228              *  the time for the loop should be independent of external
229              *  cache and memory system since it fits in the execution
230              *  prefetch buffer.
231              *
232              */
233             outb(Tmode, Latch0);
234             x = inb(T0cntr);
235             x |= inb(T0cntr)<<8;
236             aamloop(loops);
237             outb(Tmode, Latch0);
238             y = inb(T0cntr);
239             y |= inb(T0cntr)<<8;
240             x -= y;
241
242             if(x < 0)
243                     x += Freq/HZ;
244
245             if(x > Freq/(3*HZ))
246                     break;
247     }
248
249     /*
250      *  counter  goes at twice the frequency, once per transition,
251      *  i.e., twice per square wave
252      */
253     x >>= 1;
254
255     /*
256      *  figure out clock frequency and a loop multiplier for delay().
257      */
258     cpufreq = loops*((t->aalcycles*Freq)/x);
259     loopconst = (cpufreq/1000)/t->aalcycles;        /* AAM+LOOP's for 1 ms */
260
261     /*
262      *  add in possible .2% error and convert to MHz
263      */
264     cpumhz = (cpufreq + cpufreq/500)/1000000;
265   //        print("%dMHz %s loop %d\n", cpumhz, t->name, loopconst);
266   }
267
```

```
#include "u.h"
#include "lib.h"
#include "mem.h"
#include "dat.h"
#include "fns.h"
#include "io.h"

#include "dosfs.h"

/*
 * Where configuration info is left for the loaded programme.
 * This will turn into a structure as more is done by the boot loader
 * (e.g. why parse the .ini file twice?).
 * There are XXX bytes available at CONFADDR.
 */
#define BOOTLINELEN       64
#define BOOTARGS  ((char*)(CONFADDR+BOOTLINELEN))
#define   BOOTARGSLEN     (4096-0x200-BOOTLINELEN)
#define   MAXCONF         32

static char *confname[MAXCONF];
static char *confval[MAXCONF];
static int nconf;

extern char **ini;

char*
getconf(char *name)
{
  int i, n, nmatch;
  char buf[20];

  nmatch = 0;
  for(i = 0; i < nconf; i++)
        if(cistrcmp(confname[i], name) == 0)
                nmatch++;

  switch(nmatch) {
  default:
        print("\n");
        nmatch = 0;
        for(i = 0; i < nconf; i++)
                if(cistrcmp(confname[i], name) == 0)
                        print("%d. %s\n", ++nmatch, confval[i]);
        print("%d. none of the above\n", ++nmatch);
        do {
                getstr(name, buf, sizeof(buf), nil, 0);
                n = atoi(buf);
        } while(n < 1 || n > nmatch);

        for(i = 0; i < nconf; i++)
                if(cistrcmp(confname[i], name) == 0)
                        if(--n == 0)
                                return confval[i];
        break;

  case 1:
        for(i = 0; i < nconf; i++)
                if(cistrcmp(confname[i], name) == 0)
                        return confval[i];
        break;

  case 0:
        break;
  }
  return nil;
}

void
addconf(char *fmt, ...)
{
  donprint(BOOTARGS+strlen(BOOTARGS), BOOTARGS+BOOTARGSLEN, fmt, (&fmt+1));
}

/*
 *  read configuration file
 */
static char inibuf[BOOTARGSLEN];
static char id[8] = "ZORT 0\r\n";

int
dotini(Dos *dos)
{
  Dosfile rc;
  int i, n;
  char *cp, *p, *q, *line[MAXCONF];

  if(dosstat(dos, *ini, &rc) <= 0)
        return -1;

  cp = inibuf;
  *cp = 0;
  n = dosread(&rc, cp, BOOTARGSLEN-1);
  if(n <= 0)
        return -1;

  cp[n] = 0;

  /*
   * Keep a pristine copy.
   * We could change this to pass the parsed strings
   * to the booted programme instead of the raw
   * string, then it only gets done once.
   */
  if(strncmp(cp, id, sizeof(id))){
        memmove(BOOTARGS, id, sizeof(id));
        if(n+1+sizeof(id) >= BOOTARGSLEN)
                n -= sizeof(id);
        memmove(BOOTARGS+sizeof(id), cp, n+1);
  }
  else
        memmove(BOOTARGS, cp, n+1);

  /*
   * Strip out '\r', change '\t' -> ' '.
   */
  p = cp;
  for(q = cp; *q; q++){
        if(*q == '\r')
                continue;
        if(*q == '\t')
                *q = ' ';
        *p++ = *q;
  }
  *p = 0;
  n = getfields(cp, line, MAXCONF, '\n');
  for(i = 0; i < n; i++){
        cp = strchr(line[i], '=');
        if(cp == 0)
                continue;
        *cp++ = 0;
        if(cp - line[i] >= NAMELEN+1)
                *(line[i]+NAMELEN-1) = 0;
        confname[nconf] = line[i];
        confval[nconf] = cp;
        nconf++;
```

```
    }
    return 0;
}

static int
parseether(uchar *to, char *from)
{
    char nip[4];
    char *p;
    int i;

    p = from;
    while(*p == ' ')
            ++p;
    for(i = 0; i < 6; i++){
            if(*p == 0)
                    return -1;
            nip[0] = *p++;
            if(*p == 0)
                    return -1;
            nip[1] = *p++;
            nip[2] = 0;
            to[i] = strtoul(nip, 0, 16);
            if(*p == ':')
                    p++;
    }
    return 0;
}

int
isaconfig(char *class, int ctlrno, ISAConf *isa)
{
    char cc[NAMELEN], *p, *q, *r;
    int n;

    sprint(cc, "%s%d", class, ctlrno);
    for(n = 0; n < nconf; n++){
            if(cistrncmp(confname[n], cc, NAMELEN))
                    continue;
            isa->nopt = 0;
            p = confval[n];
            while(*p){
                    while(*p == ' ' || *p == '\t')
                            p++;
                    if(*p == '\0')
                            break;
                    if(cistrncmp(p, "type=", 5) == 0){
                            p += 5;
                            for(q = isa->type; q < &isa->type[NAMELEN-1]; q++){
                                    if(*p == '\0' || *p == ' ' || *p == '\t')
                                            break;
                                    *q = *p++;
                            }
                            *q = '\0';
                    }
                    else if(cistrncmp(p, "port=", 5) == 0)
                            isa->port = strtoul(p+5, &p, 0);
                    else if(cistrncmp(p, "irq=", 4) == 0)
                            isa->irq = strtoul(p+4, &p, 0);
                    else if(cistrncmp(p, "mem=", 4) == 0)
                            isa->mem = strtoul(p+4, &p, 0);
                    else if(cistrncmp(p, "size=", 5) == 0)
                            isa->size = strtoul(p+5, &p, 0);
                    else if(cistrncmp(p, "ea=", 3) == 0){
                            if(parseether(isa->ea, p+3) == -1)
                                    memset(isa->ea, 0, 6);
                    }
                    else if(isa->nopt < NISAOPT){
                            r = isa->opt[isa->nopt];
                            while(*p && *p != ' ' && *p != '\t'){
                                    *r++ = *p++;
                                    if(r-isa->opt[isa->nopt] >= ISAOPTLEN-1)
                                            break;
                            }
                            *r = '\0';
                            isa->nopt++;
                    }
                    while(*p && *p != ' ' && *p != '\t')
                            p++;
            }
            return 1;
    }
    return 0;
}
```

```
#include "u.h"
#include "lib.h"
#include "mem.h"
#include "dat.h"
#include "fns.h"
#include "io.h"

static IOQ consiq;
static IOQ consoq;

static int useuart;

int       debug = 0;

void
consinit(void)
{
  char *p;
  int baud, port;

  qinit(&consiq);
  kbdinit();

  if((p = getconf("console")) == 0 || cistrcmp(p, "cga") == 0)
          return;

  qinit(&consoq);

  port = strtoul(p, 0, 0);
  baud = 0;
  if(p = getconf("baud"))
          baud = strtoul(p, 0, 0);
  if(baud == 0)
          baud = 9600;
  uartspecial(port, kbdchar, conschar, baud);
  useuart = 1;
  uartputs(&consoq, "\n", 1);
}

void
consdrain(void)
{
  if(useuart)
          uartdrain();
}

void
kbdchar(int c)
{
  c &= 0x7F;
  if(c == 0x10)
          panic("^p");
  if(c == 0x12)
          debug = !debug;
  consiq.putc(&consiq, c);
}

int
conschar(void)
{
  return consoq.getc(&consoq);
}

void
consputs(char* s, int n)
{
  cgascreenputs(s, n);
  if(useuart)
          uartputs(&consoq, s, n);
}

static int
getline(char *buf, int size, int dotimeout)
{
  int c, i=0;
  ulong start;
  char echo;

  for (;;) {
          start = m->ticks;
          do{
                  if(dotimeout && ((m->ticks - start) > 15*HZ))   /* 15 seconds to first char
                          return -2;
                  c = consiq.getc(&consiq);
          }while(c == -1);
          if(c == '\r')
                  c = '\n';               /* turn carriage return into newline */
          if(c == '\177')
                  c = '\010';             /* turn delete into backspace */
          if(c == '\025')
                  echo = '\n';            /* echo ^U as a newline */
          else
                  echo = c;
          consputs(&echo, 1);

          if(c == '\010'){
                  if(i > 0)
                          i--; /* bs deletes last character */
                  continue;
          }
          /* a newline ends a line */
          if (c == '\n')
                  break;
          /* ^U wipes out the line */
          if (c =='\025')
                  return -1;
          if(i == size)
                  return size;
          buf[i++] = c;
  }
  buf[i] = 0;
  return i;
}

int
getstr(char *prompt, char *buf, int size, char *def, int dotimeout)
{
  int len, isdefault;

  buf[0] = 0;
  isdefault = (def && *def);
  if(isdefault == 0)
          dotimeout = 0;
  for (;;) {
          if(isdefault)
                  print("%s[default==%s]: ", prompt, def);
          else
                  print("%s: ", prompt);
          len = getline(buf, size, dotimeout);
          switch(len){
          case -1:
                  /* ^U typed */
                  continue;
          case -2:
                  /* timeout, use default */
                  consputs("\n", 1);
```

```
137                     len = 0;
138                     break;
139             default:
140                     break;
141             }
142             if(len >= size){
143                     print("line too long\n");
144                     continue;
145             }
146             break;
147     }
148     if(len == 0 && isdefault)
149             strcpy(buf, def);
150     return 0;
151   }
152
153   int
154   sprint(char *s, char *fmt, ...)
155   {
156     return donprint(s, s+PRINTSIZE, fmt, (&fmt+1)) - s;
157   }
158
159   int
160   snprint(char *s, int n, char *fmt, ...)
161   {
162     return donprint(s, s+n, fmt, (&fmt+1)) - s;
163   }
164
165   int
166   print(char *fmt, ...)
167   {
168     char buf[PRINTSIZE];
169     int n;
170
171     n = donprint(buf, buf+sizeof(buf), fmt, (&fmt+1)) - buf;
172     consputs(buf, n);
173     return n;
174   }
175
176   void
177   panic(char *fmt, ...)
178   {
179     int n;
180     char buf[PRINTSIZE];
181
182     consputs("panic: ", 7);
183     n = donprint(buf, buf+sizeof(buf), fmt, (&fmt+1)) - buf;
184     consputs(buf, n);
185     consputs("\n", 1);
186   for(;;) splhi();
187     for(n=0; n<20; n++)
188             microdelay(500);
189     spllo();
190     consdrain();
191     i8042reset();
192     print("Takes a licking and keeps on ticking...\n");
193     for(;;)
194             idle();
195   }
```

```
typedef struct List {
  void    *next;
} List;

typedef struct Alarm {
  List;
  int     busy;
  long    dt;
  void    (*f)(void*);
  void    *arg;
} Alarm;

typedef struct Block Block;
struct Block {
  Block*  next;
  uchar*  rp;                   /* first unconsumed byte */
  uchar*  wp;                   /* first empty byte */
  uchar*  lim;                  /* 1 past the end of the buffer */
  uchar*  base;                 /* start of the buffer */
  ulong   flag;
};
#define BLEN(s)   ((s)->wp - (s)->rp)

typedef struct IOQ IOQ;
typedef struct IOQ {
  uchar   buf[4096];
  uchar   *in;
  uchar   *out;
  int     state;
  int     (*getc)(IOQ*);
  int     (*putc)(IOQ*, int);
  void    *ptr;
};

enum {
  Eaddrlen        = 6,
  ETHERMINTU      = 60,         /* minimum transmit size */
  ETHERMAXTU      = 1514,       /* maximum transmit size */
  ETHERHDRSIZE    = 14,         /* size of an ethernet header */

  MaxEther        = 4,
};

typedef struct {
  uchar   d[Eaddrlen];
  uchar   s[Eaddrlen];
  uchar   type[2];
  uchar   data[1500];
  uchar   crc[4];
} Etherpkt;

extern uchar broadcast[Eaddrlen];

typedef struct Ureg Ureg;

typedef struct Segdesc {
  ulong   d0;
  ulong   d1;
} Segdesc;

typedef struct Mach {
  ulong   ticks;                /* of the clock since boot time */
  void    *alarm;               /* alarms bound to this clock */
} Mach;

extern Mach *m;

#define I_MAGIC           ((((4*11)+0)*11)+7)

typedef struct Exec Exec;
struct    Exec
{
  uchar   magic[4];             /* magic number */
  uchar   text[4];              /* size of text segment */
  uchar   data[4];              /* size of initialized data */
  uchar   bss[4];               /* size of uninitialized data */
  uchar   syms[4];              /* size of symbol table */
  uchar   entry[4];             /* entry point */
  uchar   spsz[4];              /* size of sp/pc offset table */
  uchar   pcsz[4];              /* size of pc/line number table */
};

/*
 *  a parsed .ini line
 */
#define ISAOPTLEN 16
#define NISAOPT           8

typedef struct  ISAConf {
  char    type[NAMELEN];
  ulong   port;
  ulong   irq;
  ulong   mem;
  ulong   size;
  uchar   ea[6];

  int     nopt;
  char    opt[NISAOPT][ISAOPTLEN];
} ISAConf;

typedef struct Pcidev Pcidev;
typedef struct PCMmap PCMmap;

#define BOOTLINE  ((char*)CONFADDR)

enum {
  MB =            (1024*1024),
};
#define ROUND(s, sz)      (((s)+((sz)-1))&~((sz)-1))


typedef struct Type Type;
typedef struct Medium Medium;
typedef struct Boot Boot;

enum {                                  /* type */
  Tnil            = 0x00,

  Tfloppy         = 0x01,
  Tsd             = 0x02,
  Tether          = 0x03,

  Tany            = -1,
};

enum {                                  /* name and flag */
  Fnone           = 0x00,

  Ndos            = 0x00,
  Fdos            = (1<<Ndos),
  Nboot           = 0x01,
  Fboot           = (1<<Nboot),
  Nbootp          = 0x02,
  Fbootp          = (1<<Nbootp),
  NName           = 3,
```

```
137     Fany            = Fbootp|Fboot|Fdos,
138
139     Fini            = 0x10,
140     Fprobe          = 0x80,
141   };
142
143   typedef struct Type {
144     int     type;
145     int     flag;
146     int     (*init)(void);
147     void    (*initdev)(int, char*);
148     void* (*getdospart)(int, char*);        /* actually returns Dos* */
149     void    (*addconf)(int);
150     int     (*boot)(int, char*, Boot*);
151
152     int     mask;
153     Medium* media;
154   } Type;
155
156   extern void (*etherdetach)(void);
157
158   typedef struct Lock {     /* for ilock, iunlock */
159     int locked;
160     int spl;
161   } Lock;
162
163   enum {    /* returned by bootpass */
164     MORE, ENOUGH, FAIL
165   };
166   enum {
167     INITKERNEL,
168     READEXEC,
169     READTEXT,
170     READDATA,
171     READGZIP,
172     TRYBOOT,
173     INIT9LOAD,
174     READ9LOAD,
175     FAILED
176   };
177
178   struct Boot {
179     int state;
180
181     Exec exec;
182     char *bp;       /* base ptr */
183     char *wp;       /* write ptr */
184     char *ep;       /* end ptr */
185   };
186
187   extern int        debug;
```

```
#include <u.h>

#include  "lib.h"
#include  "mem.h"
#include  "dat.h"
#include  "fns.h"
#include  "io.h"
#include  "ureg.h"

#include  "dosfs.h"
#include  "devfloppy.h"


/* Intel 82077A (8272A compatible) floppy controller */

/* This module expects the following functions to be defined
 * elsewhere:
 *
 * inb()
 * outb()
 * floppyexec()
 * floppyeject()
 * floppysetup0()
 * floppysetup1()
 * dmainit()
 * dmasetup()
 * dmaend()
 *
 * On DMA systems, floppyexec() should be an empty function;
 * on non-DMA systems, dmaend() should be an empty function;
 * dmasetup() may enforce maximum transfer sizes.
 */

enum {
  /* file types */
  Qdir=           0,
  Qdata=          (1<<2),
  Qctl=           (2<<2),
  Qmask=          (3<<2),

  DMAchan=        2,      /* floppy dma channel */
};

#define DPRINT if(0)print

FType floppytype[] =
{
 { "3½HD",         T1440kb, 512, 18, 2, 1, 80, 0x1B, 0x54, 0, },
 { "3½DD",         T1440kb, 512,  9, 2, 1, 80, 0x1B, 0x54, 2, },
 { "3½DD",         T720kb,  512,  9, 2, 1, 80, 0x1B, 0x54, 2, },
 { "5¼HD",         T1200kb, 512, 15, 2, 1, 80, 0x2A, 0x50, 0, },
 { "5¼DD",         T1200kb, 512,  9, 2, 2, 40, 0x2A, 0x50, 1, },
 { "ATT3B1",      T1200kb, 512,  8, 2, 2, 48, 0x2A, 0x50, 1, },
 { "5¼DD",         T360kb,  512,  9, 2, 1, 40, 0x2A, 0x50, 2, },
};

/*
 *  bytes per sector encoding for the controller.
 *  - index for b2c is is (bytes per sector/128).
 *  - index for c2b is code from b2c
 */
static int b2c[] =
{
[1]       0,
[2]       1,
[4]       2,
[8]       3,
};
static int c2b[] =
{
  128,
  256,
  512,
  1024,
};

FController       fl;

#define MOTORBIT(i)       (1<<((i)+4))

/*
 *  predeclared
 */
static int        cmddone(void*);
static void       floppyformat(FDrive*, char*);
static void       floppykproc(void*);
static void       floppypos(FDrive*,long);
static int        floppyrecal(FDrive*);
static int        floppyresult(void);
static void       floppyrevive(void);
static long       pcfloppyseek(FDrive*, long);
static int        floppysense(void);
static void       floppywait(void);
static long       floppyxfer(FDrive*, int, void*, long, long);

static void
fldump(void)
{
  DPRINT("sra %ux srb %ux dor %ux msr %ux dir %ux\n", inb(Psra), inb(Psrb),
          inb(Pdor), inb(Pmsr), inb(Pdir));
}

static void
floppyalarm(Alarm* a)
{
  FDrive *dp;

  for(dp = fl.d; dp < &fl.d[fl.ndrive]; dp++){
          if((fl.motor&MOTORBIT(dp->dev)) && TK2SEC(m->ticks - dp->lasttouched) > 5)
                  floppyoff(dp);
  }

  alarm(5*1000, floppyalarm, 0);
  cancel(a);
}

/*
 *  set floppy drive to its default type
 */
static void
floppysetdef(FDrive *dp)
{
  FType *t;

  for(t = floppytype; t < &floppytype[nelem(floppytype)]; t++)
          if(dp->dt == t->dt){
                  dp->t = t;
                  break;
          }
}

int
floppyinit(void)
{
  FDrive *dp;
  FType *t;
```

```
137     ulong maxtsize;
138     int mask;
139
140     dmainit(DMAchan);
141
142     floppysetup0(&fl);
143
144     /*
145      *  init dependent parameters
146      */
147     maxtsize = 0;
148     for(t = floppytype; t < &floppytype[nelem(floppytype)]; t++){
149             t->cap = t->bytes * t->heads * t->sectors * t->tracks;
150             t->bcode = b2c[t->bytes/128];
151             t->tsize = t->bytes * t->sectors;
152             if(maxtsize < t->tsize)
153                     maxtsize = t->tsize;
154     }
155
156     fl.selected = fl.d;
157
158     /*
159      *  stop the motors
160      */
161     fl.motor = 0;
162     delay(10);
163     outb(Pdor, fl.motor | Fintena | Fena);
164     delay(10);
165
166     /*
167      *  init drives
168      */
169     mask = 0;
170     for(dp = fl.d; dp < &fl.d[fl.ndrive]; dp++){
171             dp->dev = dp - fl.d;
172             if(dp->dt == Tnone)
173                     continue;
174             mask |= 1<<dp->dev;
175             floppysetdef(dp);
176             dp->cyl = -1;                   /* because we don't know */
177             //dp->cache = (uchar*)xspanalloc(maxtsize, BY2PG, 64*1024);
178             dp->cache = (uchar*)ialloc(maxtsize, BY2PG);
179             dp->ccyl = -1;
180             dp->vers = 0;
181             dp->maxtries = 5;
182     }
183
184     /*
185      *  first operation will recalibrate
186      */
187     fl.confused = 1;
188
189     floppysetup1(&fl);
190
191     /* to turn the motor off when inactive */
192     alarm(5*1000, floppyalarm, 0);
193
194     return mask;
195   }
196
197   void
198   floppyinitdev(int i, char *name)
199   {
200     sprint(name, "fd%d", i);
201   }
202
203   int
204   floppyboot(int dev, char *file, Boot *b)
205   {
206     Dos *dos;
207
208     if(strncmp(file, "dos!", 4) == 0)
209             file += 4;
210     else if(strchr(file, '!') || strcmp(file, "")==0) {
211             print("syntax is fd0!file\n");
212             return -1;
213     }
214
215     dos = floppygetdospart(dev, "dos");
216     if(dos == nil) {
217             print("no such FAT partition fd%d!dos\n", dev);
218             return -1;
219     }
220
221     return dosboot(dos, file, b);
222   }
223
224   ///*
225   // *  check if the floppy has been replaced under foot.  cause
226   // *  an error if it has.
227   // *
228   // *  a seek and a read clears the condition.  this was determined
229   // *  experimentally, there has to be a better way.
230   // *
231   // *  if the read fails, cycle through the possible floppy
232   // *  density till one works or we've cycled through all
233   // *  possibilities for this drive.
234   // */
235   //static int
236   //changed(FDrive *dp)
237   //{
238   //        FType *start;
239   //
240   //        /*
241   //         *  if floppy has changed or first time through
242   //         */
243   //        if((inb(Pdir)&Fchange) || dp->vers == 0){
244   //                DPRINT("changed\n");
245   //                fldump();
246   //                dp->vers++;
247   //                floppysetdef(dp);
248   //                start = dp->t;
249   //                dp->confused = 1;       /* make floppyon recal */
250   //                floppyon(dp);
251   //                pcfloppyseek(dp, dp->t->heads*dp->t->tsize);
252   //
253   //                while(floppyxfer(dp, Fread, dp->cache, 0, dp->t->tsize) <= 0){
254   //                        while(++dp->t){
255   //                                if(dp->t == &floppytype[nelem(floppytype)])
256   //                                        dp->t = floppytype;
257   //                                if(dp->dt == dp->t->dt)
258   //                                        break;
259   //                        }
260   //                        floppyon(dp);
261   //                        DPRINT("changed: trying %s\n", dp->t->name);
262   //                        fldump();
263   //                        if(dp->t == start){
264   //                                return -1;
265   //                        }
266   //                }
267   //        }
268   //
269   //        return 0;
270   //}
271
272   static int
```

```
readtrack(FDrive *dp, int cyl, int head)
{
  int i, nn, sofar;
  ulong pos;

  nn = dp->t->tsize;
  if(dp->ccyl==cyl && dp->chead==head)
          return nn;
  pos = (cyl*dp->t->heads+head) * nn;
  for(sofar = 0; sofar < nn; sofar += i){
          dp->ccyl = -1;
          i = floppyxfer(dp, Fread, dp->cache + sofar, pos + sofar, nn - sofar);
          if(i <= 0)
                  return -1;
  }
  dp->ccyl = cyl;
  dp->chead = head;
  return nn;
}

long
floppyread(Dos *dos, void *a, long n)
{
  FDrive *dp;
  long rv, offset;
  int sec, head, cyl;
  long len;
  uchar *aa;

  aa = a;
  dp = &fl.d[dos->dev];

  offset = dp->offset;
  floppyon(dp);
//        if(changed(dp))
//                return -1;

  for(rv = 0; rv < n; rv += len){
          /*
           *  all xfers come out of the track cache
           */
          dp->len = n - rv;
          floppypos(dp, offset+rv);
          cyl = dp->tcyl;
          head = dp->thead;
          len = dp->len;
          sec = dp->tsec;
          if(readtrack(dp, cyl, head) < 0)
                  break;
          memmove(aa+rv, dp->cache + (sec-1)*dp->t->bytes, len);
  }
  dp->offset = offset+rv;

  return rv;
}

void*
floppygetdospart(int i, char *name)
{
  static Dos dos;

  if(strcmp(name, "dos") != 0)
          return nil;

  dos.dev = i;
  dos.read = floppyread;
  dos.seek = floppyseek;
  dos.start = 0;

  /* sometimes we get spurious errors and doing it again works */
  if(dosinit(&dos) < 0 && dosinit(&dos) < 0)
          return nil;
  return &dos;
}

static int
return0(void*)
{
  return 0;
}

static void
timedsleep(int (*f)(void*), void* arg, int ms)
{
  int s;
  ulong end;

  end = m->ticks + 1 + MS2TK(ms);
  while(m->ticks < end && !(*f)(arg)){
          s = spllo();
          delay(10);
          splx(s);
  }
}

/*
 *  start a floppy drive's motor.
 */
static void
floppyon(FDrive *dp)
{
  int alreadyon;
  int tries;

  if(fl.confused)
          floppyrevive();

  /* start motor and select drive */
  dp->lasttouched = m->ticks;
  alreadyon = fl.motor & MOTORBIT(dp->dev);
  if(!alreadyon){
          fl.motor |= MOTORBIT(dp->dev);
          outb(Pdor, fl.motor | Fintena | Fena | dp->dev);
          /* wait for drive to spin up */
          timedsleep(return0, 0, 750);

          /* clear any pending interrupts */
          floppysense();
  }

  /* set transfer rate */
  if(fl.rate != dp->t->rate){
          fl.rate = dp->t->rate;
          outb(Pdsr, fl.rate);
  }

  /* get drive to a known cylinder */
  if(dp->confused)
          for(tries = 0; tries < 4; tries++)
                  if(floppyrecal(dp) >= 0)
                          break;
  dp->lasttouched = m->ticks;
  fl.selected = dp;
}

/*
```

```
409     *  stop the floppy if it hasn't been used in 5 seconds
410     */
411   static void
412   floppyoff(FDrive *dp)
413   {
414     fl.motor &= ~MOTORBIT(dp->dev);
415     outb(Pdor, fl.motor | Fintena | Fena | dp->dev);
416   }
417
418   /*
419    *  send a command to the floppy
420    */
421   static int
422   floppycmd(void)
423   {
424     int i;
425     int tries;
426
427     fl.nstat = 0;
428     for(i = 0; i < fl.ncmd; i++){
429             for(tries = 0; ; tries++){
430                     if((inb(Pmsr)&(Ffrom|Fready)) == Fready)
431                             break;
432                     if(tries > 1000){
433                             DPRINT("cmd %ux can't be sent (%d)\n", fl.cmd[0], i);
434                             fldump();
435
436                             /* empty fifo, might have been a bad command */
437                             floppyresult();
438                             return -1;
439                     }
440                     microdelay(1);
441             }
442             outb(Pfdata, fl.cmd[i]);
443     }
444     return 0;
445   }
446
447   /*
448    *  get a command result from the floppy
449    *
450    *  when the controller goes ready waiting for a command
451    *  (instead of sending results), we're done
452    *
453    */
454   static int
455   floppyresult(void)
456   {
457     int i, s;
458     int tries;
459
460     /* get the result of the operation */
461     for(i = 0; i < sizeof(fl.stat); i++){
462             /* wait for status byte */
463             for(tries = 0; ; tries++){
464                     s = inb(Pmsr)&(Ffrom|Fready);
465                     if(s == Fready){
466                             fl.nstat = i;
467                             return fl.nstat;
468                     }
469                     if(s == (Ffrom|Fready))
470                             break;
471                     if(tries > 1000){
472                             DPRINT("floppyresult: %d stats\n", i);
473                             fldump();
474                             fl.confused = 1;
475                             return -1;
476                     }
477                     microdelay(1);
478             }
479             fl.stat[i] = inb(Pfdata);
480     }
481     fl.nstat = sizeof(fl.stat);
482     return fl.nstat;
483   }
484
485   /*
486    *  calculate physical address of a logical byte offset into the disk
487    *
488    *  truncate dp->length if it crosses a track boundary
489    */
490   static void
491   floppypos(FDrive *dp, long off)
492   {
493     int lsec;
494     int ltrack;
495     int end;
496
497     lsec = off/dp->t->bytes;
498     ltrack = lsec/dp->t->sectors;
499     dp->tcyl = ltrack/dp->t->heads;
500     dp->tsec = (lsec % dp->t->sectors) + 1;
501     dp->thead = (lsec/dp->t->sectors) % dp->t->heads;
502
503     /*
504      *  can't read across track boundaries.
505      *  if so, decrement the bytes to be read.
506      */
507     end = (ltrack+1)*dp->t->sectors*dp->t->bytes;
508     if(off+dp->len > end)
509             dp->len = end - off;
510   }
511
512   /*
513    *  get the interrupt cause from the floppy.
514    */
515   static int
516   floppysense(void)
517   {
518     fl.ncmd = 0;
519     fl.cmd[fl.ncmd++] = Fsense;
520     if(floppycmd() < 0)
521             return -1;
522     if(floppyresult() < 2){
523             DPRINT("can't read sense response\n");
524             fldump();
525             fl.confused = 1;
526             return -1;
527     }
528     return 0;
529   }
530
531   static int
532   cmddone(void *a)
533   {
534     USED(a);
535     return fl.ncmd == 0;
536   }
537
538   /*
539    *  Wait for a floppy interrupt.  If none occurs in 5 seconds, we
540    *  may have missed one.  This only happens on some portables which
541    *  do power management behind our backs.  Call the interrupt
542    *  routine to try to clear any conditions.
543    */
544   static void
```

```
545  floppywait(void)
546  {
547    timedsleep(cmddone, 0, 5000);
548    if(!cmddone(0)){
549            floppyintr(0);
550            fl.confused = 1;
551    }
552  }
553
554  /*
555   *  we've lost the floppy position, go to cylinder 0.
556   */
557  static int
558  floppyrecal(FDrive *dp)
559  {
560    dp->ccyl = -1;
561    dp->cyl = -1;
562
563    fl.ncmd = 0;
564    fl.cmd[fl.ncmd++] = Frecal;
565    fl.cmd[fl.ncmd++] = dp->dev;
566    if(floppycmd() < 0)
567            return -1;
568    floppywait();
569    if(fl.nstat < 2){
570            DPRINT("recalibrate: confused %ux\n", inb(Pmsr));
571            fl.confused = 1;
572            return -1;
573    }
574    if((fl.stat[0] & (Codemask|Seekend)) != Seekend){
575            DPRINT("recalibrate: failed\n");
576            dp->confused = 1;
577            return -1;
578    }
579    dp->cyl = fl.stat[1];
580    if(dp->cyl != 0){
581            DPRINT("recalibrate: wrong cylinder %d\n", dp->cyl);
582            dp->cyl = -1;
583            dp->confused = 1;
584            return -1;
585    }
586
587    dp->confused = 0;
588    return 0;
589  }
590
591  /*
592   *  if the controller or a specific drive is in a confused state,
593   *  reset it and get back to a kown state
594   */
595  static void
596  floppyrevive(void)
597  {
598    FDrive *dp;
599
600    /*
601     *  reset the controller if it's confused
602     */
603    if(fl.confused){
604            DPRINT("floppyrevive in\n");
605            fldump();
606
607            /* reset controller and turn all motors off */
608            splhi();
609            fl.ncmd = 1;
610            fl.cmd[0] = 0;
611            outb(Pdor, 0);
612            delay(10);
613            outb(Pdor, Fintena|Fena);
614            delay(10);
615            spllo();
616            fl.motor = 0;
617            fl.confused = 0;
618            floppywait();
619
620            /* mark all drives in an unknown state */
621            for(dp = fl.d; dp < &fl.d[fl.ndrive]; dp++)
622                    dp->confused = 1;
623
624            /* set rate to a known value */
625            outb(Pdsr, 0);
626            fl.rate = 0;
627
628            DPRINT("floppyrevive out\n");
629            fldump();
630    }
631  }
632
633  /*
634   *  seek to the target cylinder
635   *
636   *          interrupt, no results
637   */
638  static long
639  pcfloppyseek(FDrive *dp, long off)
640  {
641    floppypos(dp, off);
642    if(dp->cyl == dp->tcyl){
643            dp->offset = off;
644            return off;
645    }
646    dp->cyl = -1;
647
648    fl.ncmd = 0;
649    fl.cmd[fl.ncmd++] = Fseek;
650    fl.cmd[fl.ncmd++] = (dp->thead<<2) | dp->dev;
651    fl.cmd[fl.ncmd++] = dp->tcyl * dp->t->steps;
652    if(floppycmd() < 0)
653            return -1;
654    floppywait();
655    if(fl.nstat < 2){
656            DPRINT("seek: confused\n");
657            fl.confused = 1;
658            return -1;
659    }
660    if((fl.stat[0] & (Codemask|Seekend)) != Seekend){
661            DPRINT("seek: failed\n");
662            dp->confused = 1;
663            return -1;
664    }
665
666    dp->cyl = dp->tcyl;
667    dp->offset = off;
668    DPRINT("seek to %d succeeded\n", dp->offset);
669
670    return dp->offset;
671  }
672
673  /*
674   *  read or write to floppy.  try up to three times.
675   */
676  static long
677  floppyxfer(FDrive *dp, int cmd, void *a, long off, long n)
678  {
679    long offset;
680    int tries;
```

```
681
682     if(off >= dp->t->cap)
683             return 0;
684     if(off + n > dp->t->cap)
685             n = dp->t->cap - off;
686
687     /* retry on error (until it gets ridiculous) */
688     for(tries = 0; tries < dp->maxtries; tries++){
689
690             dp->len = n;
691             if(pcfloppyseek(dp, off) < 0){
692                     DPRINT("xfer: seek failed\n");
693                     dp->confused = 1;
694                     continue;
695             }
696
697             /*
698              *  set up the dma (dp->len may be trimmed)
699              */
700             dp->len = dmasetup(DMAchan, a, dp->len, cmd==Fread);
701             if(dp->len < 0){
702     buggery:
703                     dmaend(DMAchan);
704                     continue;
705             }
706
707             /*
708              *  start operation
709              */
710             fl.ncmd = 0;
711             fl.cmd[fl.ncmd++] = cmd | (dp->t->heads > 1 ? Fmulti : 0);
712             fl.cmd[fl.ncmd++] = (dp->thead<<2) | dp->dev;
713             fl.cmd[fl.ncmd++] = dp->tcyl;
714             fl.cmd[fl.ncmd++] = dp->thead;
715             fl.cmd[fl.ncmd++] = dp->tsec;
716             fl.cmd[fl.ncmd++] = dp->t->bcode;
717             fl.cmd[fl.ncmd++] = dp->t->sectors;
718             fl.cmd[fl.ncmd++] = dp->t->gpl;
719             fl.cmd[fl.ncmd++] = 0xFF;
720             if(floppycmd() < 0)
721                     goto buggery;
722
723             /*
724              *  give bus to DMA, floppyintr() will read result
725              */
726             floppywait();
727             dmaend(DMAchan);
728
729             /*
730              *  check for errors
731              */
732             if(fl.nstat < 7){
733                     DPRINT("xfer: confused\n");
734                     fl.confused = 1;
735                     continue;
736             }
737             if((fl.stat[0] & Codemask)!=0 || fl.stat[1] || fl.stat[2]){
738                     DPRINT("xfer: failed %lux %lux %lux\n", fl.stat[0],
739                             fl.stat[1], fl.stat[2]);
740                     DPRINT("offset %lud len %d\n", off, dp->len);
741                     if((fl.stat[0]&Codemask)==Cmdexec && fl.stat[1]==Overrun){
742                             DPRINT("DMA overrun: retry\n");
743                     } else
744                             dp->confused = 1;
745                     continue;
746             }
747
748             /*
749              *  check for correct cylinder
750              */
751             offset = fl.stat[3] * dp->t->heads + fl.stat[4];
752             offset = offset*dp->t->sectors + fl.stat[5] - 1;
753             offset = offset * c2b[fl.stat[6]];
754             if(offset != off+dp->len){
755                     DPRINT("xfer: ends on wrong cyl\n");
756                     dp->confused = 1;
757                     continue;
758             }
759
760             dp->lasttouched = m->ticks;
761             dp->maxtries = 20;
762             return dp->len;
763     }
764
765     return -1;
766   }
767
768   static void
769   floppyintr(Ureg *ur)
770   {
771     USED(ur);
772     switch(fl.cmd[0]&~Fmulti){
773     case Fread:
774     case Fwrite:
775     case Fformat:
776     case Fdumpreg:
777             floppyresult();
778             break;
779     case Fseek:
780     case Frecal:
781     default:
782             floppysense();  /* to clear interrupt */
783             break;
784     }
785     fl.ncmd = 0;
786   }
```

```
#include  "u.h"
#include  "lib.h"
#include  "mem.h"
#include  "dat.h"
#include  "fns.h"
#include  "dosfs.h"

/*
 *  predeclared
 */
static void       bootdump(Dosboot*);
static void       setname(Dosfile*, char*);

/*
 *  debugging
 */
#define chatty    0
#define chat      if(chatty)print

/*
 *  block io buffers
 */
enum
{
  Nbio=   16,
};
typedef struct    Clustbuf        Clustbuf;
struct Clustbuf
{
  int     age;
  long    sector;
  uchar   *iobuf;
  Dos     *dos;
  int     size;
};
Clustbuf  bio[Nbio];

/*
 *  get an io block from an io buffer
 */
Clustbuf*
getclust(Dos *dos, long sector)
{
  Clustbuf *p, *oldest;
  int size;

  chat("getclust @ %d\n", sector);

  /*
   *  if we have it, just return it
   */
  for(p = bio; p < &bio[Nbio]; p++){
          if(sector == p->sector && dos == p->dos){
                  p->age = m->ticks;
                  chat("getclust %d in cache\n", sector);
                  return p;
          }
  }

  /*
   *  otherwise, reuse the oldest entry
   */
  oldest = bio;
  for(p = &bio[1]; p < &bio[Nbio]; p++){
          if(p->age <= oldest->age)
                  oldest = p;
  }
  p = oldest;

  /*
   *  make sure the buffer is big enough
   */
  size = dos->clustsize*dos->sectsize;
  if(p->iobuf==0 || p->size < size)
          p->iobuf = ialloc(size, 0);
  p->size = size;

  /*
   *  read in the cluster
   */
  chat("getclust addr %d\n", (sector+dos->start)*dos->sectsize);
  if((*dos->seek)(dos, (sector+dos->start)*dos->sectsize) < 0){
          chat("can't seek block\n");
          return 0;
  }
  if((*dos->read)(dos, p->iobuf, size) != size){
          chat("can't read block\n");
          return 0;
  }

  p->age = m->ticks;
  p->dos = dos;
  p->sector = sector;
  chat("getclust %d read\n", sector);
  return p;
}

/*
 *  walk the fat one level ( n is a current cluster number ).
 *  return the new cluster number or -1 if no more.
 */
static long
fatwalk(Dos *dos, int n)
{
  ulong k, sect;
  Clustbuf *p;
  int o;

  chat("fatwalk %d\n", n);

  if(n < 2 || n >= dos->fatclusters)
          return -1;

  switch(dos->fatbits){
  case 12:
          k = (3*n)/2; break;
  case 16:
          k = 2*n; break;
  case 32:
          k = 4*n; break;
  default:
          return -1;
  }
  if(k >= dos->fatsize*dos->sectsize)
          panic("getfat");

  sect = (k/(dos->sectsize*dos->clustsize))*dos->clustsize + dos->fataddr;
  o = k%(dos->sectsize*dos->clustsize);
  p = getclust(dos, sect);
  k = p->iobuf[o++];
  if(o >= dos->sectsize*dos->clustsize){
          p = getclust(dos, sect+dos->clustsize);
          o = 0;
  }
  k |= p->iobuf[o++]<<8;
  if(dos->fatbits == 12){
```

```
137             if(n&1)
138                     k >>= 4;
139             else
140                     k &= 0xfff;
141             if(k >= 0xff8)
142                     k = -1;
143     }
144     else if (dos->fatbits == 32){
145             if(o >= dos->sectsize*dos->clustsize){
146                     p = getclust(dos, sect+dos->clustsize);
147                     o = 0;
148             }
149             k |= p->iobuf[o++]<<16;
150             k |= p->iobuf[o]<<24;
151             if (k >= 0xfffffff8)
152                     k = -1;
153     }
154     else
155             k = k < 0xfff8 ? k : -1;
156     chat("fatwalk %d -> %d\n", n, k);
157     return k;
158   }
159
160   /*
161    *  map a file's logical cluster address to a physical sector address
162    */
163   static long
164   fileaddr(Dosfile *fp, long ltarget)
165   {
166     Dos *dos = fp->dos;
167     long l;
168     long p;
169
170     chat("fileaddr %8.8s %d\n", fp->name, ltarget);
171     /*
172      *  root directory is contiguous and easy (unless FAT32)
173      */
174     if(fp->pstart == 0 && dos->rootsize != 0) {
175             if(ltarget*dos->sectsize*dos->clustsize >= dos->rootsize*sizeof(Dosdir))
176                     return -1;
177             l = dos->rootaddr + ltarget*dos->clustsize;
178             chat("fileaddr %d -> %d\n", ltarget, l);
179             return l;
180     }
181
182     /*
183      *  anything else requires a walk through the fat
184      */
185     if(ltarget >= fp->lcurrent && fp->pcurrent){
186             /* start at the currrent point */
187             l = fp->lcurrent;
188             p = fp->pcurrent;
189     } else {
190             /* go back to the beginning */
191             l = 0;
192             p = fp->pstart;
193     }
194     while(l != ltarget){
195             /* walk the fat */
196             p = fatwalk(dos, p);
197             if(p < 0)
198                     return -1;
199             l++;
200     }
201     fp->lcurrent = l;
202     fp->pcurrent = p;
203
204     /*
205      *  clusters start at 2 instead of 0 (why? - presotto)
206      */
207     l =  dos->dataaddr + (p-2)*dos->clustsize;
208     chat("fileaddr %d -> %d\n", ltarget, l);
209     return l;
210   }
211
212   /*
213    *  read from a dos file
214    */
215   long
216   dosread(Dosfile *fp, void *a, long n)
217   {
218     long addr;
219     long rv;
220     int i;
221     int off;
222     Clustbuf *p;
223     uchar *from, *to;
224
225     if((fp->attr & DDIR) == 0){
226             if(fp->offset >= fp->length)
227                     return 0;
228             if(fp->offset+n > fp->length)
229                     n = fp->length - fp->offset;
230     }
231
232     to = a;
233     for(rv = 0; rv < n; rv+=i){
234             /*
235              *  read the cluster
236              */
237             addr = fileaddr(fp, fp->offset/fp->dos->clustbytes);
238             if(addr < 0)
239                     return -1;
240             p = getclust(fp->dos, addr);
241             if(p == 0)
242                     return -1;
243
244             /*
245              *  copy the bytes we need
246              */
247             off = fp->offset % fp->dos->clustbytes;
248             from = &p->iobuf[off];
249             i = n - rv;
250             if(i > fp->dos->clustbytes - off)
251                     i = fp->dos->clustbytes - off;
252             memmove(to, from, i);
253             to += i;
254             fp->offset += i;
255     }
256
257     return rv;
258   }
259
260   /*
261    *  walk a directory returns
262    *        -1 if something went wrong
263    *         0 if not found
264    *         1 if found
265    */
266   int
267   doswalk(Dosfile *file, char *name)
268   {
269     Dosdir d;
270     long n;
271
272     if((file->attr & DDIR) == 0){
```

```
273             chat("walking non-directory!\n");
274             return -1;
275     }
276
277     setname(file, name);
278
279     file->offset = 0;       /* start at the beginning */
280     while((n = dosread(file, &d, sizeof(d))) == sizeof(d)){
281             chat("comparing to %8.8s.%3.3s\n", d.name, d.ext);
282             if(memcmp(file->name, d.name, sizeof(d.name)) != 0)
283                     continue;
284             if(memcmp(file->ext, d.ext, sizeof(d.ext)) != 0)
285                     continue;
286             if(d.attr & DVLABEL){
287                     chat("%8.8s.%3.3s is a LABEL\n", d.name, d.ext);
288                     continue;
289             }
290             file->attr = d.attr;
291             file->pstart = GSHORT(d.start);
292             if (file->dos->fatbits == 32)
293                     file->pstart |= GSHORT(d.highstart) << 16;
294             file->length = GLONG(d.length);
295             file->pcurrent = 0;
296             file->lcurrent = 0;
297             file->offset = 0;
298             return 1;
299     }
300     return n >= 0 ? 0 : -1;
301   }
302
303   /*
304    *  instructions that boot blocks can start with
305    */
306   #define   JMPSHORT        0xeb
307   #define JMPNEAR           0xe9
308
309   int
310   dosinit(Dos *dos)
311   {
312     Clustbuf *p;
313     Dosboot *b;
314     int i;
315
316     /* defaults till we know better */
317     dos->sectsize = 512;
318     dos->clustsize = 1;
319
320     /* get first sector */
321     p = getclust(dos, 0);
322     if(p == 0){
323             chat("can't read boot block\n");
324             return -1;
325     }
326
327     p->dos = 0;
328     b = (Dosboot *)p->iobuf;
329     if(b->magic[0] != JMPNEAR && (b->magic[0] != JMPSHORT || b->magic[2] != 0x90)){
330             chat("no dos file system %x %x %x %x\n",
331                     b->magic[0], b->magic[1], b->magic[2], b->magic[3]);
332             return -1;
333     }
334
335     if(chatty)
336             bootdump(b);
337
338     /*
339      *  determine the systems' wondersous properties
340      */
341     dos->fatbits = 0;
342     dos->sectsize = GSHORT(b->sectsize);
343     dos->clustsize = b->clustsize;
344     dos->clustbytes = dos->sectsize*dos->clustsize;
345     dos->nresrv = GSHORT(b->nresrv);
346     dos->nfats = b->nfats;
347     dos->fatsize = GSHORT(b->fatsize);
348     dos->rootsize = GSHORT(b->rootsize);
349     dos->volsize = GSHORT(b->volsize);
350     if(dos->volsize == 0)
351             dos->volsize = GLONG(b->bigvolsize);
352     dos->mediadesc = b->mediadesc;
353     if(dos->fatsize == 0) {
354             chat("fat32\n");
355             dos->rootsize = 0;
356             dos->fatsize = GLONG(b->bigfatsize);
357             dos->fatbits = 32;
358     }
359     dos->fataddr = dos->nresrv;
360     if (dos->rootsize == 0) {
361             dos->rootaddr = 0;
362             dos->rootclust = GLONG(b->rootdirstartclust);
363             dos->dataaddr = dos->fataddr + dos->nfats*dos->fatsize;
364     } else {
365             dos->rootaddr = dos->fataddr + dos->nfats*dos->fatsize;
366             i = dos->rootsize*sizeof(Dosdir) + dos->sectsize - 1;
367             i = i/dos->sectsize;
368             dos->dataaddr = dos->rootaddr + i;
369     }
370     dos->fatclusters = 2+(dos->volsize - dos->dataaddr)/dos->clustsize;
371     if(dos->fatbits != 32) {
372             if(dos->fatclusters < 4087)
373                     dos->fatbits = 12;
374             else
375                     dos->fatbits = 16;
376     }
377     dos->freeptr = 2;
378
379     /*
380      *  set up the root
381      */
382     dos->root.dos = dos;
383     dos->root.pstart = dos->rootsize == 0 ? dos->rootclust : 0;
384     dos->root.pcurrent = dos->root.lcurrent = 0;
385     dos->root.offset = 0;
386     dos->root.attr = DDIR;
387     dos->root.length = dos->rootsize*sizeof(Dosdir);
388
389     return 0;
390   }
391
392   static void
393   bootdump(Dosboot *b)
394   {
395     if(chatty == 0)
396             return;
397     print("magic: 0x%2.2x 0x%2.2x 0x%2.2x\n",
398             b->magic[0], b->magic[1], b->magic[2]);
399     print("version: \"%8.8s\"\n", b->version);
400     print("sectsize: %d\n", GSHORT(b->sectsize));
401     print("allocsize: %d\n", b->clustsize);
402     print("nresrv: %d\n", GSHORT(b->nresrv));
403     print("nfats: %d\n", b->nfats);
404     print("rootsize: %d\n", GSHORT(b->rootsize));
405     print("volsize: %d\n", GSHORT(b->volsize));
406     print("mediadesc: 0x%2.2x\n", b->mediadesc);
407     print("fatsize: %d\n", GSHORT(b->fatsize));
408     print("trksize: %d\n", GSHORT(b->trksize));
```

```
409     print("nheads: %d\n", GSHORT(b->nheads));
410     print("nhidden: %d\n", GLONG(b->nhidden));
411     print("bigvolsize: %d\n", GLONG(b->bigvolsize));
412   /*
413     print("driveno: %d\n", b->driveno);
414     print("reserved0: 0x%2.2x\n", b->reserved0);
415     print("bootsig: 0x%2.2x\n", b->bootsig);
416     print("volid: 0x%8.8x\n", GLONG(b->volid));
417     print("label: \"%11.11s\"\n", b->label);
418   */
419   }
420
421   /*
422    *  grab next element from a path, return the pointer to unprocessed portion of
423    *  path.
424    */
425   static char *
426   nextelem(char *path, char *elem)
427   {
428     int i;
429
430     while(*path == '/')
431             path++;
432     if(*path==0 || *path==' ')
433             return 0;
434     for(i=0; *path!='\0' && *path!='/' && *path!=' '; i++){
435             if(i==28){
436                     print("name component too long\n");
437                     return 0;
438             }
439             *elem++ = *path++;
440     }
441     *elem = '\0';
442     return path;
443   }
444
445   int
446   dosstat(Dos *dos, char *path, Dosfile *f)
447   {
448     char element[NAMELEN];
449
450     *f = dos->root;
451     while(path = nextelem(path, element)){
452             switch(doswalk(f, element)){
453             case -1:
454                     return -1;
455             case 0:
456                     return 0;
457             }
458     }
459     return 1;
460   }
461
462   /*
463    *  read in a segment
464    */
465   long
466   dosreadseg(Dosfile *fp, void *va, long len)
467   {
468     char *a;
469     long n, sofar;
470
471     a = va;
472     for(sofar = 0; sofar < len; sofar += n){
473             n = 8*1024;
474             if(len - sofar < n)
475                     n = len - sofar;
476             n = dosread(fp, a + sofar, n);
477             if(n <= 0)
478                     break;
479             print(".");
480     }
481     return sofar;
482   }
483
484   /*
485    *  boot
486    */
487   int
488   dosboot(Dos *dos, char *path, Boot *b)
489   {
490     Dosfile file;
491     long n;
492     static char buf[8192];
493
494     switch(dosstat(dos, path, &file)){
495     case -1:
496             print("error walking to %s\n", path);
497             return -1;
498     case 0:
499             print("%s not found\n", path);
500             return -1;
501     case 1:
502             print("found %8.8s.%3.3s attr 0x%ux start 0x%lux len %d\n", file.name,
503                     file.ext, file.attr, file.pstart, file.length);
504             break;
505     }
506
507     while((n = dosreadseg(&file, buf, sizeof buf)) > 0) {
508             if(bootpass(b, buf, n) != MORE)
509                     break;
510     }
511
512     bootpass(b, nil, 0);    /* tries boot */
513     return -1;
514   }
515
516   /*
517    *  set up a dos file name
518    */
519   static void
520   setname(Dosfile *fp, char *from)
521   {
522     char *to;
523
524     to = fp->name;
525     for(; *from && to-fp->name < 8; from++, to++){
526             if(*from == '.'){
527                     from++;
528                     break;
529             }
530             if(*from >= 'a' && *from <= 'z')
531                     *to = *from + 'A' - 'a';
532             else
533                     *to = *from;
534     }
535     while(to - fp->name < 8)
536             *to++ = ' ';
537
538     to = fp->ext;
539     for(; *from && to-fp->ext < 3; from++, to++){
540             if(*from >= 'a' && *from <= 'z')
541                     *to = *from + 'A' - 'a';
542             else
543                     *to = *from;
544     }
```

```
545     while(to-fp->ext < 3)
546             *to++ = ' ';
547
548     chat("name is %8.8s %3.3s\n", fp->name, fp->ext);
549   }
```

```
#include "u.h"
#include "lib.h"
#include "mem.h"
#include "dat.h"
#include "fns.h"
#include "io.h"

#include "etherif.h"
#include "ether8390.h"

/*
 * Western Digital/Standard Microsystems Corporation cards (WD80[01]3).
 * Also handles 8216 cards (Elite Ultra).
 * Configuration code based on that provided by SMC a long time ago.
 */
enum {                                  /* 83C584 Bus Interface Controller */
  Msr             = 0x00,         /* Memory Select Register */
  Icr             = 0x01,         /* Interface Configuration Register */
  Iar             = 0x02,         /* I/O Address Register */
  Bio             = 0x03,         /* BIOS ROM Address Register */
  Ear             = 0x03,         /* EEROM Address Register (shared with Bio) */
  Irr             = 0x04,         /* Interrupt Request Register */
  Hcr             = 0x04,         /* 8216 hardware control */
  Laar            = 0x05,         /* LA Address Register */
  Ijr             = 0x06,         /* Initialisation Jumpers */
  Gp2             = 0x07,         /* General Purpose Data Register */
  Lar             = 0x08,         /* LAN Address Registers */
  Id              = 0x0E,         /* Card ID byte */
  Cksum           = 0x0F,         /* Checksum */
};

enum {                                  /* Msr */
  Rst             = 0x80,         /* software reset */
  Menb            = 0x40,         /* memory enable */
};

enum {                                  /* Icr */
  Bit16           = 0x01,         /* 16-bit bus */
  Other           = 0x02,         /* other register access */
  Ir2             = 0x04,         /* IR2 */
  Msz             = 0x08,         /* SRAM size */
  Rla             = 0x10,         /* recall LAN address */
  Rx7             = 0x20,         /* recall all but I/O and LAN address */
  Rio             = 0x40,         /* recall I/O address from EEROM */
  Sto             = 0x80,         /* non-volatile EEROM store */
};

enum {                                  /* Laar */
  ZeroWS16        = 0x20,         /* zero wait states for 16-bit ops */
  L16en           = 0x40,         /* enable 16-bit LAN operation */
  M16en           = 0x80,         /* enable 16-bit memory access */
};

enum {                                  /* Ijr */
  Ienable         = 0x01,         /* 8216 interrupt enable */
};

/*
 * Mapping from configuration bits to interrupt level.
 */
static int irq8003[8] = {
  9, 3, 5, 7, 10, 11, 15, 4,
};

static int irq8216[8] = {
  0, 9, 3, 5, 7, 10, 11, 15,
};

static void
reset8003(Ether* ether, uchar ea[Eaddrlen], uchar ic[8])
{
  Dp8390 *ctlr;
  ulong port;

  ctlr = ether->ctlr;
  port = ether->port;

  /*
   * Check for old, dumb 8003E, which doesn't have an interface
   * chip. Only Msr exists out of the 1st eight registers, reads
   * of the others just alias the 2nd eight registers, the LAN
   * address ROM. Can check Icr, Irr and Laar against the ethernet
   * address read above and if they match it's an 8003E (or an
   * 8003EBT, 8003S, 8003SH or 8003WT, doesn't matter), in which
   * case the default irq gets used.
   */
  if(memcmp(&ea[1], &ic[1], 5) == 0){
          memset(ic, 0, sizeof(ic));
          ic[Msr] = (((ulong)ether->mem)>>13) & 0x3F;
  }
  else{
          /*
           * As a final sanity check for the 8013EBT, which doesn't have
           * the 83C584 interface chip, but has 2 real registers, write Gp2
           * and if it reads back the same, it's not an 8013EBT.
           */
          outb(port+Gp2, 0xAA);
          inb(port+Msr);                          /* wiggle bus */
          if(inb(port+Gp2) != 0xAA){
                  memset(ic, 0, sizeof(ic));
                  ic[Msr] = (((ulong)ether->mem)>>13) & 0x3F;
          }
          else
                  ether->irq = irq8003[((ic[Irr]>>5) & 0x3)|(ic[Icr] & 0x4)];

          /*
           * Check if 16-bit card.
           * If Bit16 is read/write, then it's an 8-bit card.
           * If Bit16 is set, it's in a 16-bit slot.
           */
          outb(port+Icr, ic[Icr]^Bit16);
          inb(port+Msr);                          /* wiggle bus */
          if((inb(port+Icr) & Bit16) == (ic[Icr] & Bit16)){
                  ctlr->width = 2;
                  ic[Icr] &= ~Bit16;
          }
          outb(port+Icr, ic[Icr]);

          if(ctlr->width == 2 && (inb(port+Icr) & Bit16) == 0)
                  ctlr->width = 1;
  }

  ether->mem = (ulong)KADDR((ic[Msr] & 0x3F)<<13);
  if(ctlr->width == 2)
          ether->mem |= (ic[Laar] & 0x1F)<<19;
  else
          ether->mem |= 0x80000;

  if(ic[Icr] & (1<<3))
          ether->size = 32*1024;
  if(ctlr->width == 2)
          ether->size <<= 1;

  /*
   * Enable interface RAM, set interface width.
   */
```

```
137     outb(port+Msr, ic[Msr]|Menb);
138     if(ctlr->width == 2)
139             outb(port+Laar, ic[Laar]|L16en|M16en|ZeroWS16);
140   }
141
142   static void
143   reset8216(Ether* ether, uchar[8])
144   {
145     uchar hcr, irq, x;
146     ulong addr, port;
147     Dp8390 *ctlr;
148
149     ctlr = ether->ctlr;
150     port = ether->port;
151
152     ctlr->width = 2;
153
154     /*
155      * Switch to the alternate register set and retrieve the memory
156      * and irq information.
157      */
158     hcr = inb(port+Hcr);
159     outb(port+Hcr, 0x80|hcr);
160     addr = inb(port+0x0B) & 0xFF;
161     irq = inb(port+0x0D);
162     outb(port+Hcr, hcr);
163
164     ether->mem = (ulong)KADDR(0xC0000+((((addr>>2) & 0x30)|(addr & 0x0F))<<13));
165     ether->size = 8192*(1<<((addr>>4) & 0x03));
166     ether->irq = irq8216[((irq>>4) & 0x04)|((irq>>2) & 0x03)];
167
168     /*
169      * Enable interface RAM, set interface width, and enable interrupts.
170      */
171     x = inb(port+Msr) & ~Rst;
172     outb(port+Msr, Menb|x);
173     x = inb(port+Laar);
174     outb(port+Laar, M16en|x);
175     outb(port+Ijr, Ienable);
176   }
177
178   /*
179    * Get configuration parameters, enable memory.
180    * There are opportunities here for buckets of code, try to resist.
181    */
182   int
183   wd8003reset(Ether* ether)
184   {
185     int i;
186     uchar ea[Eaddrlen], ic[8], id, nullea[Eaddrlen], sum;
187     ulong port;
188     Dp8390 *ctlr;
189
190     /*
191      * Set up the software configuration.
192      * Use defaults for port, irq, mem and size if not specified.
193      * Defaults are set for the dumb 8003E which can't be
194      * autoconfigured.
195      */
196     if(ether->port == 0)
197             ether->port = 0x280;
198     if(ether->irq == 0)
199             ether->irq = 3;
200     if(ether->mem == 0)
201             ether->mem = 0xD0000;
202     if(ether->size == 0)
203             ether->size = 8*1024;
204
205     /*
206      * Look for the interface. Read the LAN address ROM
207      * and validate the checksum - the sum of all 8 bytes
208      * should be 0xFF.
209      * At the same time, get the (possible) interface chip
210      * registers, they'll be used later to check for aliasing.
211      */
212     port = ether->port;
213     sum = 0;
214     for(i = 0; i < sizeof(ea); i++){
215             ea[i] = inb(port+Lar+i);
216             sum += ea[i];
217             ic[i] = inb(port+i);
218     }
219     id = inb(port+Id);
220     sum += id;
221     sum += inb(port+Cksum);
222     if(sum != 0xFF)
223             return -1;
224
225     ether->ctlr = malloc(sizeof(Dp8390));
226     ctlr = ether->ctlr;
227     ctlr->ram = 1;
228
229     if((id & 0xFE) == 0x2A)
230             reset8216(ether, ic);
231     else
232             reset8003(ether, ea, ic);
233
234     /*
235      * Set the DP8390 ring addresses.
236      */
237     ctlr->port = port+0x10;
238     ctlr->tstart = 0;
239     ctlr->pstart = HOWMANY(sizeof(Etherpkt), Dp8390BufSz);
240     ctlr->pstop = HOWMANY(ether->size, Dp8390BufSz);
241
242     /*
243      * Finally, init the 8390, set the ethernet address
244      * and claim the memory used.
245      */
246     dp8390reset(ether);
247     memset(nullea, 0, Eaddrlen);
248     if(memcmp(nullea, ether->ea, Eaddrlen) == 0){
249             for(i = 0; i < sizeof(ether->ea); i++)
250                     ether->ea[i] = ea[i];
251     }
252     dp8390setea(ether);
253
254     if(umbrwmalloc(PADDR(ether->mem), ether->size, 0) == 0)
255             print("ether8003: warning - 0x%luX unavailable", PADDR(ether->mem));
256
257     return 0;
258   }
```

```
1     void      aamloop(int);
2     void      addconf(char*, ...);
3     Alarm*    alarm(int, void (*)(Alarm*), void*);
4     void      alarminit(void);
5     Block*    allocb(int);
6     int       bootp(int, char*, Boot*);
7     int       bootpass(Boot*, void*, int);
8     void      cancel(Alarm*);
9     void      cgascreenputs(char*, int);
10    int       cistrcmp(char*, char*);
11    int       cistrncmp(char*, char*, int);
12    void      checkalarms(void);
13    void      clockinit(void);
14    int       conschar(void);
15    void      consdrain(void);
16    void      consinit(void);
17    void      consputs(char*, int);
18    void      delay(int);
19    uchar*    etheraddr(int);
20    int       etherinit(void);
21    void      etherinitdev(int, char*);
22    int       etherrxpkt(int, Etherpkt*, int);
23    int       ethertxpkt(int, Etherpkt*, int, int);
24    #define   evenaddr(x)             /* 386 doesn't care */
25    int       floppyboot(int, char*, Boot*);
26    int       floppyinit(void);
27    void      floppyinitdev(int, char*);
28    void* floppygetdospart(int, char*);
29    void      freeb(Block*);
30    char*     getconf(char*);
31    ulong     getcr0(void);
32    ulong     getcr2(void);
33    ulong     getcr3(void);
34    int       getfields(char*, char**, int, char);
35    int       getstr(char*, char*, int, char*, int);
36    int       gunzip(uchar*, int, uchar*, int);
37    void      i8042a20(void);
38    void      i8042reset(void);
39    void*     ialloc(ulong, int);
40    void      idle(void);
41    void      ilock(Lock*);
42    int       inb(int);
43    ushort    ins(int);
44    ulong     inl(int);
45    void      insb(int, void*, int);
46    void      inss(int, void*, int);
47    void      insl(int, void*, int);
48    void      iunlock(Lock*);
49    int       isaconfig(char*, int, ISAConf*);
50    void      kbdinit(void);
51    void      kbdchar(int);
52    void      machinit(void);
53    void      meminit(ulong);
54    void      microdelay(int);
55    void      mmuinit(void);
56    #define   nelem(x)        (sizeof(x)/sizeof(x[0]))
57    uchar     nvramread(int);
58    void      outb(int, int);
59    void      outs(int, ushort);
60    void      outl(int, ulong);
61    void      outsb(int, void*, int);
62    void      outss(int, void*, int);
63    void      outsl(int, void*, int);
64    void      panic(char*, ...);
65    int       pcicfgr8(Pcidev*, int);
66    int       pcicfgr16(Pcidev*, int);
67    int       pcicfgr32(Pcidev*, int);
68    void      pcicfgw8(Pcidev*, int, int);
69    void      pcicfgw16(Pcidev*, int, int);
70    void      pcicfgw32(Pcidev*, int, int);
71    Pcidev* pcimatch(Pcidev*, int, int);
72    void      pcireset(void);
73    void      pcisetbme(Pcidev*);
74    int       pcmcistuple(int, int, void*, int);
75    int       pcmspecial(char*, ISAConf*);
76    void      pcmspecialclose(int);
77    void      pcmunmap(int, PCMmap*);
78    void      putcr3(ulong);
79    void      putidt(Segdesc*, int);
80    void      qinit(IOQ*);
81    void      sdaddconf(int);
82    int       sdboot(int, char*, Boot*);
83    void*     sdgetdospart(int, char*);
84    int       sdinit(void);
85    void      sdinitdev(int, char*);
86    int       sdsetpart(int, char*);
87    void      setvec(int, void (*)(Ureg*, void*), void*);
88    int       splhi(void);
89    int       spllo(void);
90    void      splx(int);
91    void      trapinit(void);
92    void      uartdrain(void);
93    void      uartspecial(int, void (*)(int), int (*)(void), int);
94    void      uartputs(IOQ*, char*, int);
95    ulong     umbmalloc(ulong, int, int);
96    void      umbfree(ulong, int);
97    ulong     umbrwmalloc(ulong, int, int);
98    void      warp9(ulong);
99    int       x86cpuid(int*, int*);
100
101   #define malloc(n) ialloc(n, 0)
102   #define free(v)
103
104   #define   GSHORT(p)       (((p)[1]<<8)|(p)[0])
105   #define   GLONG(p)        ((GSHORT(p+2)<<16)|GSHORT(p))
106   #define   GLSHORT(p)      (((p)[0]<<8)|(p)[1])
107   #define   GLLONG(p)       ((GLSHORT(p)<<16)|GLSHORT(p+2))
108
109   #define KADDR(a)   ((void*)((ulong)(a)|KZERO))
110   #define PADDR(a)   ((ulong)(a)&~KZERO)
111
112   #define   HOWMANY(x, y)   (((x)+((y)-1))/(y))
113   #define ROUNDUP(x, y)     (HOWMANY((x), (y))*(y))
114
115
116   #define xalloc(n) ialloc(n, 0)
117   #define xfree(v)
118   #define lock(l)   if(l);else;
119   #define unlock(l) if(l);else;
120
121   int       dmacount(int);
122   int       dmadone(int);
123   void      dmaend(int);
124   void      dmainit(int);
125   long      dmasetup(int, void*, long, int);
126
```

```
/*
 *  programmable interrupt vectors (for the 8259's)
 */
enum
{
  Bptvec=         3,              /* breakpoints */
  Mathemuvec=     7,              /* math coprocessor emulation interrupt */
  Mathovervec=    9,              /* math coprocessor overrun interrupt */
  Matherr1vec=    16,             /* math coprocessor error interrupt */
  Faultvec=       14,             /* page fault */

#ifdef notdef
  Int0vec=        24,             /* first 8259 */
   Clockvec=      Int0vec+0,      /*  clock interrupts */
   Kbdvec=        Int0vec+1,      /*  keyboard interrupts */
   Uart1vec=      Int0vec+3,      /*  modem line */
   Uart0vec=      Int0vec+4,      /*  serial line */
   PCMCIAvec=     Int0vec+5,      /*  PCMCIA card change */
   Floppyvec=     Int0vec+6,      /*  floppy interrupts */
   Parallelvec=   Int0vec+7,      /*  parallel port interrupts */
  Int1vec=        Int0vec+8,
   Vector9=       Int0vec+9,      /*  unassigned */
   Vector10=      Int0vec+10,     /*  unassigned, usually ethernet */
   Vector11=      Int0vec+11,     /*  unassigned, usually scsi */
   Mousevec=      Int0vec+12,     /*  mouse interrupt */
   Matherr2vec=   Int0vec+13,     /*  math coprocessor */
   ATAvec0=       Int0vec+14,     /*  ATA controller #1 */
   ATAvec1=       Int0vec+15,     /*  ATA controller #2 */
#endif /* notdef */

  Syscallvec=     64,

  VectorPIC       = 24,           /* external [A]PIC interrupts */
  VectorCLOCK     = VectorPIC+0,
  VectorKBD       = VectorPIC+1,
  VectorUART1     = VectorPIC+3,
  VectorUART0     = VectorPIC+4,
  VectorPCMCIA    = VectorPIC+5,
  VectorFLOPPY    = VectorPIC+6,
  VectorLPT       = VectorPIC+7,
  VectorIRQ7      = VectorPIC+7,
  VectorAUX       = VectorPIC+12, /* PS/2 port */
  VectorIRQ13     = VectorPIC+13, /* coprocessor on x386 */
  VectorATA0      = VectorPIC+14,
  VectorATA1      = VectorPIC+15,
  MaxVectorPIC    = VectorPIC+15,
};

enum {
  BusCBUS         = 0,            /* Corollary CBUS */
  BusCBUSII,                      /* Corollary CBUS II */
  BusEISA,                        /* Extended ISA */
  BusFUTURE,                      /* IEEE Futurebus */
  BusINTERN,                      /* Internal bus */
  BusISA,                         /* Industry Standard Architecture */
  BusMBI,                         /* Multibus I */
  BusMBII,                        /* Multibus II */
  BusMCA,                         /* Micro Channel Architecture */
  BusMPI,                         /* MPI */
  BusMPSA,                        /* MPSA */
  BusNUBUS,                       /* Apple Macintosh NuBus */
  BusPCI,                         /* Peripheral Component Interconnect */
  BusPCMCIA,                      /* PC Memory Card International Association */
  BusTC,                          /* DEC TurboChannel */
  BusVL,                          /* VESA Local bus */
  BusVME,                         /* VMEbus */
  BusXPRESS,                      /* Express System Bus */
};

#define MKBUS(t,b,d,f)    (((t)<<24)|(((b)&0xFF)<<16)|(((d)&0x1F)<<11)|(((f)&0x07)<<8))
#define BUSFNO(tbdf)      (((tbdf)>>8)&0x07)
#define BUSDNO(tbdf)      (((tbdf)>>11)&0x1F)
#define BUSBNO(tbdf)      (((tbdf)>>16)&0xFF)
#define BUSTYPE(tbdf)     ((tbdf)>>24)
#define BUSBDF(tbdf)      ((tbdf)&0x00FFFF00)
#define BUSUNKNOWN        (-1)

enum {
  MaxEISA         = 16,
  EISAconfig      = 0xC80,
};

/*
 * PCI support code.
 */
enum {                                  /* type 0 and type 1 pre-defined header */
  PciVID          = 0x00,         /* vendor ID */
  PciDID          = 0x02,         /* device ID */
  PciPCR          = 0x04,         /* command */
  PciPSR          = 0x06,         /* status */
  PciRID          = 0x08,         /* revision ID */
  PciCCRp         = 0x09,         /* programming interface class code */
  PciCCRu         = 0x0A,         /* sub-class code */
  PciCCRb         = 0x0B,         /* base class code */
  PciCLS          = 0x0C,         /* cache line size */
  PciLTR          = 0x0D,         /* latency timer */
  PciHDT          = 0x0E,         /* header type */
  PciBST          = 0x0F,         /* BIST */

  PciBAR0         = 0x10,         /* base address */
  PciBAR1         = 0x14,

  PciINTL         = 0x3C,         /* interrupt line */
  PciINTP         = 0x3D,         /* interrupt pin */
};

enum {                                  /* type 0 pre-defined header */
  PciBAR2         = 0x18,
  PciBAR3         = 0x1C,
  PciBAR4         = 0x20,
  PciBAR5         = 0x24,
  PciCIS          = 0x28,         /* cardbus CIS pointer */
  PciSVID         = 0x2C,         /* subsystem vendor ID */
  PciSID          = 0x2E,         /* cardbus CIS pointer */
  PciEBAR0        = 0x30,         /* expansion ROM base address */
  PciMGNT         = 0x3E,         /* burst period length */
  PciMLT          = 0x3F,         /* maximum latency between bursts */
};

enum {                                  /* type 1 pre-defined header */
  PciPBN          = 0x18,         /* primary bus number */
  PciSBN          = 0x19,         /* secondary bus number */
  PciUBN          = 0x1A,         /* subordinate bus number */
  PciSLTR         = 0x1B,         /* secondary latency timer */
  PciIBR          = 0x1C,         /* I/O base */
  PciILR          = 0x1D,         /* I/O limit */
  PciSPSR         = 0x1E,         /* secondary status */
  PciMBR          = 0x20,         /* memory base */
  PciMLR          = 0x22,         /* memory limit */
  PciPMBR         = 0x24,         /* prefetchable memory base */
  PciPMLR         = 0x26,         /* prefetchable memory limit */
  PciPUBR         = 0x28,         /* prefetchable base upper 32 bits */
  PciPULR         = 0x2C,         /* prefetchable limit upper 32 bits */
  PciIUBR         = 0x30,         /* I/O base upper 16 bits */
  PciIULR         = 0x32,         /* I/O limit upper 16 bits */
  PciEBAR1        = 0x28,         /* expansion ROM base address */
```

```
137     PciBCR          = 0x3E,         /* bridge control register */
138   };
139
140   typedef struct Pcidev Pcidev;
141   typedef struct Pcidev {
142     int     tbdf;                   /* type+bus+device+function */
143     ushort  vid;                    /* vendor ID */
144     ushort  did;                    /* device ID */
145
146     struct {
147             ulong   bar;            /* base address */
148             int     size;
149     } mem[6];
150
151     uchar   intl;                   /* interrupt line */
152     uchar   ccrp;
153     uchar   ccru;
154     uchar   ccrb;
155
156     Pcidev* list;
157     Pcidev* bridge;                 /* down a bus */
158     Pcidev* link;                   /* next device on this bno */
159   };
160
161   /*
162    * PCMCIA support code.
163    */
164   /*
165    * Map between ISA memory space and PCMCIA card memory space.
166    */
167   struct PCMmap {
168     ulong   ca;                     /* card address */
169     ulong   cea;                    /* card end address */
170     ulong   isa;                    /* ISA address */
171     int     len;                    /* length of the ISA area */
172     int     attr;                   /* attribute memory */
173     int     ref;
174   };
```

```
#include  "u.h"
#include  "lib.h"
#include  "mem.h"
#include  "dat.h"
#include  "fns.h"
#include  "io.h"

enum {
  Data=           0x60,   /* data port */

  Status=         0x64,   /* status port */
   Inready=       0x01,   /*  input character ready */
   Outbusy=       0x02,   /*  output busy */
   Sysflag=       0x04,   /*  system flag */
   Cmddata=       0x08,   /*  cmd==0, data==1 */
   Inhibit=       0x10,   /*  keyboard/mouse inhibited */
   Minready=      0x20,   /*  mouse character ready */
   Rtimeout=      0x40,   /*  general timeout */
   Parity=        0x80,

  Cmd=            0x64,   /* command port (write only) */

  Spec=   0x80,

  PF=     Spec|0x20,      /* num pad function key */
  View=   Spec|0x00,      /* view (shift window up) */
  KF=     Spec|0x40,      /* function key */
  Shift=  Spec|0x60,
  Break=  Spec|0x61,
  Ctrl=   Spec|0x62,
  Latin=  Spec|0x63,
  Caps=   Spec|0x64,
  Num=    Spec|0x65,
  No=     Spec|0x7F,      /* no mapping */

  Home=   KF|13,
  Up=     KF|14,
  Pgup=   KF|15,
  Print=  KF|16,
  Left=   View,
  Right=  View,
  End=    '\r',
  Down=   View,
  Pgdown= View,
  Ins=    KF|20,
  Del=    0x7F,
};

uchar kbtab[] =
{
[0x00]    No,     0x1b,   '1',    '2',    '3',    '4',    '5',    '6',
[0x08]    '7',    '8',    '9',    '0',    '-',    '=',    '\b',   '\t',
[0x10]    'q',    'w',    'e',    'r',    't',    'y',    'u',    'i',
[0x18]    'o',    'p',    '[',    ']',    '\n',   Ctrl,   'a',    's',
[0x20]    'd',    'f',    'g',    'h',    'j',    'k',    'l',    ';',
[0x28]    '\'',   '`',    Shift,  '\\',   'z',    'x',    'c',    'v',
[0x30]    'b',    'n',    'm',    ',',    '.',    '/',    Shift,  No,
[0x38]    Latin,  ' ',    Caps,   KF|1,   KF|2,   KF|3,   KF|4,   KF|5,
[0x40]    KF|6,   KF|7,   KF|8,   KF|9,   KF|10,  Num,    KF|12,  Home,
[0x48]    No,     No,     No,     No,     No,     No,     No,     No,
[0x50]    No,     No,     No,     No,     No,     No,     No,     KF|11,
[0x58]    KF|12,  No,     No,     No,     No,     No,     No,     No,
};

uchar kbtabshift[] =
{
[0x00]    No,     0x1b,   '!',    '@',    '#',    '$',    '%',    '^',
[0x08]    '&',    '*',    '(',    ')',    '_',    '+',    '\b',   '\t',
[0x10]    'Q',    'W',    'E',    'R',    'T',    'Y',    'U',    'I',
[0x18]    'O',    'P',    '{',    '}',    '\n',   Ctrl,   'A',    'S',
[0x20]    'D',    'F',    'G',    'H',    'J',    'K',    'L',    ':',
[0x28]    '"',    '~',    Shift,  '|',    'Z',    'X',    'C',    'V',
[0x30]    'B',    'N',    'M',    '<',    '>',    '?',    Shift,  No,
[0x38]    Latin,  ' ',    Caps,   KF|1,   KF|2,   KF|3,   KF|4,   KF|5,
[0x40]    KF|6,   KF|7,   KF|8,   KF|9,   KF|10,  Num,    KF|12,  Home,
[0x48]    No,     No,     No,     No,     No,     No,     No,     No,
[0x50]    No,     No,     No,     No,     No,     No,     No,     KF|11,
[0x58]    KF|12,  No,     No,     No,     No,     No,     No,     No,
};

uchar kbtabesc1[] =
{
[0x00]    No,     No,     No,     No,     No,     No,     No,     No,
[0x08]    No,     No,     No,     No,     No,     No,     No,     No,
[0x10]    No,     No,     No,     No,     No,     No,     No,     No,
[0x18]    No,     No,     No,     No,     No,     Ctrl,   No,     No,
[0x20]    No,     No,     No,     No,     No,     No,     No,     No,
[0x28]    No,     No,     No,     No,     No,     No,     No,     No,
[0x30]    No,     No,     No,     No,     No,     No,     No,     Print,
[0x38]    Latin,  No,     No,     No,     No,     No,     No,     No,
[0x40]    No,     No,     No,     No,     No,     No,     Break,  Home,
[0x48]    Up,     Pgup,   No,     Down,   No,     Right,  No,     End,
[0x50]    Left,   Pgdown, Ins,    Del,    No,     No,     No,     No,
[0x58]    No,     No,     No,     No,     No,     No,     No,     No,
};

struct latin
{
  uchar   l;
  char    c[2];
}latintab[] = {
  L'¡',   "!!",   /* spanish initial ! */
  L'¢',   "c|",   /* cent */
  L'¢',   "c$",   /* cent */
  L'£',   "l$",   /* pound sterling */
  L'¤',   "g$",   /* general currency */
  L'¥',   "y$",   /* yen */
  L'¥',   "j$",   /* yen */
  L'¦',   "||",   /* broken vertical bar */
  L'§',   "SS",   /* section symbol */
  L'¨',   "\"\"", /* dieresis */
  L'©',   "cr",   /* copyright */
  L'©',   "cO",   /* copyright */
  L'ª',   "sa",   /* super a, feminine ordinal */
  L'«',   "<<",   /* left angle quotation */
  L'¬',   "no",   /* not sign, hooked overbar */
  L'-',   "--",   /* soft hyphen */
  L'®',   "rg",   /* registered trademark */
  L'¯',   "__",   /* macron */
  L'°',   "s0",   /* degree (sup o) */
  L'±',   "+-",   /* plus-minus */
  L'²',   "s2",   /* sup 2 */
  L'³',   "s3",   /* sup 3 */
  L'´',   "''",   /* grave accent */
  L'µ',   "mu",   /* mu */
  L'¶',   "pg",   /* paragraph (pilcrow) */
  L'·',   "..",   /* centered . */
  L'¸',   ",,",   /* cedilla */
  L'¹',   "s1",   /* sup 1 */
  L'º',   "so",   /* sup o */
  L'»',   ">>",   /* right angle quotation */
  L'¼',   "14",   /* 1/4 */
  L'½',   "12",   /* 1/2 */
  L'¾',   "34",   /* 3/4 */
  L'¿',   "??",   /* spanish initial ? */
  L'À',   "A`",   /* A grave */
```

```
    L'Á',   "A'",   /* A acute */
    L'Â',   "A^",   /* A circumflex */
    L'Ã',   "A~",   /* A tilde */
    L'Ä',   "A\"",  /* A dieresis */
    L'Ä',   "A:",   /* A dieresis */
    L'Å',   "Ao",   /* A circle */
    L'Å',   "AO",   /* A circle */
    L'Æ',   "Ae",   /* AE ligature */
    L'Æ',   "AE",   /* AE ligature */
    L'Ç',   "C,",   /* C cedilla */
    L'È',   "E`",   /* E grave */
    L'É',   "E'",   /* E acute */
    L'Ê',   "E^",   /* E circumflex */
    L'Ë',   "E\"",  /* E dieresis */
    L'Ë',   "E:",   /* E dieresis */
    L'Ì',   "I`",   /* I grave */
    L'Í',   "I'",   /* I acute */
    L'Î',   "I^",   /* I circumflex */
    L'Ï',   "I\"",  /* I dieresis */
    L'Ï',   "I:",   /* I dieresis */
    L'Ð',   "D-",   /* Eth */
    L'Ñ',   "N~",   /* N tilde */
    L'Ò',   "O`",   /* O grave */
    L'Ó',   "O'",   /* O acute */
    L'Ô',   "O^",   /* O circumflex */
    L'Õ',   "O~",   /* O tilde */
    L'Ö',   "O\"",  /* O dieresis */
    L'Ö',   "O:",   /* O dieresis */
    L'Ö',   "OE",   /* O dieresis */
    L'Ö',   "Oe",   /* O dieresis */
    L'×',   "xx",   /* times sign */
    L'Ø',   "O/",   /* O slash */
    L'Ù',   "U`",   /* U grave */
    L'Ú',   "U'",   /* U acute */
    L'Û',   "U^",   /* U circumflex */
    L'Ü',   "U\"",  /* U dieresis */
    L'Ü',   "U:",   /* U dieresis */
    L'Ü',   "UE",   /* U dieresis */
    L'Ü',   "Ue",   /* U dieresis */
    L'Ý',   "Y'",   /* Y acute */
    L'Þ',   "P|",   /* Thorn */
    L'Þ',   "Th",   /* Thorn */
    L'Þ',   "TH",   /* Thorn */
    L'ß',   "ss",   /* sharp s */
    L'à',   "a`",   /* a grave */
    L'á',   "a'",   /* a acute */
    L'â',   "a^",   /* a circumflex */
    L'ã',   "a~",   /* a tilde */
    L'ä',   "a\"",  /* a dieresis */
    L'ä',   "a:",   /* a dieresis */
    L'å',   "ao",   /* a circle */
    L'æ',   "ae",   /* ae ligature */
    L'ç',   "c,",   /* c cedilla */
    L'è',   "e`",   /* e grave */
    L'é',   "e'",   /* e acute */
    L'ê',   "e^",   /* e circumflex */
    L'ë',   "e\"",  /* e dieresis */
    L'ë',   "e:",   /* e dieresis */
    L'ì',   "i`",   /* i grave */
    L'í',   "i'",   /* i acute */
    L'î',   "i^",   /* i circumflex */
    L'ï',   "i\"",  /* i dieresis */
    L'ï',   "i:",   /* i dieresis */
    L'ð',   "d-",   /* eth */
    L'ñ',   "n~",   /* n tilde */
    L'ò',   "o`",   /* o grave */
    L'ó',   "o'",   /* o acute */
    L'ô',   "o^",   /* o circumflex */
    L'õ',   "o~",   /* o tilde */
    L'ö',   "o\"",  /* o dieresis */
    L'ö',   "o:",   /* o dieresis */
    L'ö',   "oe",   /* o dieresis */
    L'÷',   "-:",   /* divide sign */
    L'ø',   "o/",   /* o slash */
    L'ù',   "u`",   /* u grave */
    L'ú',   "u'",   /* u acute */
    L'û',   "u^",   /* u circumflex */
    L'ü',   "u\"",  /* u dieresis */
    L'ü',   "u:",   /* u dieresis */
    L'ü',   "ue",   /* u dieresis */
    L'ý',   "y'",   /* y acute */
    L'þ',   "th",   /* thorn */
    L'þ',   "þ|",   /* thorn */
    L'ÿ',   "y\"",  /* y dieresis */
    L'ÿ',   "y:",   /* y dieresis */
    0,      0,
};

enum
{
  /* controller command byte */
  Cscs1=          (1<<6),         /* scan code set 1 */
  Cmousedis=      (1<<5),         /* mouse disable */
  Ckbddis=        (1<<4),         /* kbd disable */
  Csf=            (1<<2),         /* system flag */
  Cmouseint=      (1<<1),         /* mouse interrupt enable */
  Ckbdint=        (1<<0),         /* kbd interrupt enable */
};

static uchar ccc;

int
latin1(int k1, int k2)
{
  struct latin *l;

  for(l=latintab; l->l; l++)
          if(k1==l->c[0] && k2==l->c[1])
                  return l->l;
  return 0;
}

/*
 *  wait for output no longer busy
 */
static int
outready(void)
{
  int tries;

  for(tries = 0; (inb(Status) & Outbusy); tries++){
          if(tries > 500)
                  return -1;
          delay(2);
  }
  return 0;
}

/*
 *  wait for input
 */
static int
inready(void)
{
  int tries;

```

```
273     for(tries = 0; !(inb(Status) & Inready); tries++){
274             if(tries > 500)
275                     return -1;
276             delay(2);
277     }
278     return 0;
279   }
280
281   /*
282    *  ask 8042 to enable the use of address bit 20
283    */
284   void
285   i8042a20(void)
286   {
287     outready();
288     outb(Cmd, 0xD1);
289     outready();
290     outb(Data, 0xDF);
291     outready();
292   }
293
294   /*
295    *  ask 8042 to reset the machine
296    */
297   void
298   i8042reset(void)
299   {
300     int i, x;
301   #ifdef notdef
302     ushort *s = (ushort*)(KZERO|0x472);
303
304     *s = 0x1234;            /* BIOS warm-boot flag */
305   #endif /* notdef */
306
307     outready();
308     outb(Cmd, 0xFE);        /* pulse reset line (means resend on AT&T machines) */
309     outready();
310
311     /*
312      *  Pulse it by hand (old somewhat reliable)
313      */
314     x = 0xDF;
315     for(i = 0; i < 5; i++){
316             x ^= 1;
317             outready();
318             outb(Cmd, 0xD1);
319             outready();
320             outb(Data, x);  /* toggle reset */
321             delay(100);
322     }
323   }
324
325   /*
326    *  keyboard interrupt
327    */
328   static void
329   kbdintr(Ureg*, void*)
330   {
331     int s, c;
332     static int esc1, esc2;
333     static int alt, caps, ctl, num, shift;
334     static int lstate, k1, k2;
335     int keyup;
336
337     /*
338      *  get status
339      */
340     s = inb(Status);
341     if(!(s&Inready))
342             return;
343
344     /*
345      *  get the character
346      */
347     c = inb(Data);
348
349     /*
350      *  if it's the aux port...
351      */
352     if(s & Minready)
353             return;
354
355     /*
356      *  e0's is the first of a 2 character sequence
357      */
358     if(c == 0xe0){
359             esc1 = 1;
360             return;
361     } else if(c == 0xe1){
362             esc2 = 2;
363             return;
364     }
365
366     keyup = c&0x80;
367     c &= 0x7f;
368     if(c > sizeof kbtab){
369             c |= keyup;
370             if(c != 0xFF)   /* these come fairly often: CAPSLOCK U Y */
371                     print("unknown key %ux\n", c);
372             return;
373     }
374
375     if(esc1){
376             c = kbtabesc1[c];
377             esc1 = 0;
378     } else if(esc2){
379             esc2--;
380             return;
381     } else if(shift)
382             c = kbtabshift[c];
383     else
384             c = kbtab[c];
385
386     if(caps && c<='z' && c>='a')
387             c += 'A' - 'a';
388
389     /*
390      *  keyup only important for shifts
391      */
392     if(keyup){
393             switch(c){
394             case Latin:
395                     alt = 0;
396                     break;
397             case Shift:
398                     shift = 0;
399                     break;
400             case Ctrl:
401                     ctl = 0;
402                     break;
403             }
404             return;
405     }
406
407     /*
408      *  normal character
```

```
409      */
410     if(!(c & Spec)){
411             if(ctl){
412                     if(alt && c == Del)
413                             panic("Ctrl-Alt-Del");
414                     c &= 0x1f;
415             }
416             switch(lstate){
417             case 1:
418                     k1 = c;
419                     lstate = 2;
420                     return;
421             case 2:
422                     k2 = c;
423                     lstate = 0;
424                     c = latin1(k1, k2);
425                     if(c == 0){
426                             kbdchar(k1);
427                             c = k2;
428                     }
429                     /* fall through */
430             default:
431                     break;
432             }
433     } else {
434             switch(c){
435             case Caps:
436                     caps ^= 1;
437                     return;
438             case Num:
439                     num ^= 1;
440                     return;
441             case Shift:
442                     shift = 1;
443                     return;
444             case Latin:
445                     alt = 1;
446                     lstate = 1;
447                     return;
448             case Ctrl:
449                     ctl = 1;
450                     return;
451             }
452     }
453     kbdchar(c);
454   }
455
456   static char *initfailed = "kbd init failed\n";
457
458   void
459   kbdinit(void)
460   {
461     int c;
462
463     /* wait for a quiescent controller */
464     while((c = inb(Status)) & (Outbusy | Inready))
465             if(c & Inready)
466                     inb(Data);
467
468     /* get current controller command byte */
469     outb(Cmd, 0x20);
470     if(inready() < 0){
471             print("kbdinit: can't read ccc\n");
472             ccc = 0;
473     } else
474             ccc = inb(Data);
475
476     /* enable kbd xfers and interrupts */
477     ccc &= ~Ckbddis;
478     ccc |= Csf | Ckbdint | Cscs1;
479     if(outready() < 0)
480             print(initfailed);
481     outb(Cmd, 0x60);
482     if(outready() < 0)
483             print(initfailed);
484     outb(Data, ccc);
485     if(outready() < 0)
486             print(initfailed);
487
488     setvec(VectorKBD, kbdintr, 0);
489   }
```

```
/*
 * functions (possibly) linked in, complete, from libc.
 */

/*
 * mem routines
 */
extern    void*   memccpy(void*, void*, int, long);
extern    void*   memset(void*, int, long);
extern    int     memcmp(void*, void*, long);
extern    void*   memmove(void*, void*, long);
extern    void*   memchr(void*, int, long);

/*
 * string routines
 */
extern    char*   strcat(char*, char*);
extern    char*   strchr(char*, char);
extern    int     strcmp(char*, char*);
extern    char*   strcpy(char*, char*);
extern    char*   strncat(char*, char*, long);
extern    char*   strncpy(char*, char*, long);
extern    int     strncmp(char*, char*, long);
extern    long    strlen(char*);
extern    char*   strrchr(char*, char);

/*
 * print routines
 *        Fconv isn't used but is defined to satisfy prototypes in libg.h
 *        that are never called.
 */
typedef   struct Fconv Fconv;

extern    char*   donprint(char*, char*, char*, void*);
extern    int     sprint(char*, char*, ...);
extern    int     snprint(char*, int, char*, ...);
extern    int     print(char*, ...);

#define   PRINTSIZE       256

/*
 * one-of-a-kind
 */
extern    int     atoi(char*);
extern    void    longjmp(jmp_buf, int);
extern    int     setjmp(jmp_buf);
extern    long    strtol(char*, char**, int);
extern    ulong   strtoul(char*, char**, int);
extern    long    end;

/*
 * Syscall data structures
 */

#define   MORDER  0x0003  /* mask for bits defining order of mounting */
#define   MREPL   0x0000  /* mount replaces object */
#define   MBEFORE 0x0001  /* mount goes before others in union directory */
#define   MAFTER  0x0002  /* mount goes after others in union directory */
#define   MCREATE 0x0004  /* permit creation in mounted directory */
#define   MMASK   0x0007  /* all bits on */

#define   OREAD   0       /* open for read */
#define   OWRITE  1       /* write */
#define   ORDWR   2       /* read and write */
#define   OEXEC   3       /* execute, == read but check execute permission */
#define   OTRUNC  16      /* or'ed in (except for exec), truncate file first */
#define   OCEXEC  32      /* or'ed in, close on exec */
#define   ORCLOSE 64      /* or'ed in, remove on close */

#define   NCONT   0       /* continue after note */
#define   NDFLT   1       /* terminate after note */

typedef struct Qid        Qid;
typedef struct Dir        Dir;
typedef struct Waitmsg    Waitmsg;

#define   ERRLEN  64
#define   DIRLEN  116
#define   NAMELEN 28

struct Qid
{
  ulong   path;
  ulong   vers;
};

struct Dir
{
  char    name[NAMELEN];
  char    uid[NAMELEN];
  char    gid[NAMELEN];
  Qid     qid;
  ulong   mode;
  long    atime;
  long    mtime;
  Length;
  short   type;
  short   dev;
};

struct Waitmsg
{
  int     pid;            /* of loved one */
  int     status;         /* unused; a placeholder */
  ulong   time[3];        /* of loved one */
  char    msg[ERRLEN];
};
```

```
#include "u.h"
#include "lib.h"
#include "mem.h"
#include "dat.h"
#include "fns.h"
#include "io.h"

#include "dosfs.h"

Type types[] = {
  {       Tfloppy,
          Fini|Fdos,
          floppyinit, floppyinitdev,
          floppygetdospart, 0, floppyboot,
  },
  {       Tether,
          Fbootp,
          etherinit, etherinitdev,
          0, 0, bootp,
  },
  {       Tsd,
          Fini|Fdos,
          sdinit, sdinitdev,
          sdgetdospart, sdaddconf, sdboot,
  },
  {       Tnil,
          0,
          0, 0, 0, 0,
          { 0, },
  },
};

#include "etherif.h"

extern int ether2114xreset(Ether*);
extern int elnk3reset(Ether*);
extern int i82557reset(Ether*);
extern int elnk3reset(Ether*);
extern int ether589reset(Ether*);
extern int ne2000reset(Ether*);
extern int wd8003reset(Ether*);
extern int ec2treset(Ether*);

struct {
  char    *type;
  int     (*reset)(Ether*);
} ethercards[] = {
  { "21140", ether2114xreset, },
  { "2114x", ether2114xreset, },
  { "i82557", i82557reset, },
  { "elnk3", elnk3reset, },
  { "3C509", elnk3reset, },
  { "3C589", ether589reset, },
  { "3C562", ether589reset, },
  { "589E", ether589reset, },
  { "NE2000", ne2000reset, },
  { "WD8003", wd8003reset, },
  { "EC2T", ec2treset, },

  { 0, }
};

typedef struct Mode Mode;

enum {
  Maxdev          = 7,
  Dany            = -1,
  Nmedia          = 16,
  Nini            = 10,
};

enum {                                          /* mode */
  Mauto           = 0x00,
  Mlocal          = 0x01,
  Manual          = 0x02,
  NMode           = 0x03,
};

typedef struct Medium Medium;
struct Medium {
  Type*   type;
  int     flag;
  int     dev;
  char name[NAMELEN];

  Dos *inidos;
  char *part;
  char *ini;

  Medium* next;
};

typedef struct Mode {
  char*   name;
  int     mode;
} Mode;

static Medium media[Nmedia];
static Medium *curmedium = media;

static Mode modes[NMode+1] = {
  [Mauto]         { "auto",   Mauto,  },
  [Mlocal]        { "local",  Mlocal, },
  [Manual]        { "manual", Manual, },
};

static char *inis[] = {
  "plan9/plan9.ini",
  "plan9.ini",
  0
};

char **ini;

int scsi0port;

static Medium*
parse(char *line, char **file)
{
  char *p;
  Type *tp;
  Medium *mp;

  if(p = strchr(line, '!')) {
          *p++ = 0;
          *file = p;
  } else
          *file = "";

  for(tp = types; tp->type != Tnil; tp++)
          for(mp = tp->media; mp; mp = mp->next)
                  if(strcmp(mp->name, line) == 0)
                          return mp;
  if(p)
          *--p = '!';
  return nil;
```

```
}

static int
boot(Medium *mp, char *file)
{
  Type *tp;
  Medium *xmp;
  static int didaddconf;
  Boot b;

  memset(&b, 0, sizeof b);
  b.state = INITKERNEL;

  if(didaddconf == 0) {
          didaddconf = 1;
          for(tp = types; tp->type != Tnil; tp++)
                  if(tp->addconf)
                          for(xmp = tp->media; xmp; xmp = xmp->next)
                                  (*tp->addconf)(xmp->dev);
  }

  sprint(BOOTLINE, "%s!%s", mp->name, file);
  return (*mp->type->boot)(mp->dev, file, &b);
}

static Medium*
allocm(Type *tp)
{
  Medium **l;

  if(curmedium >= &media[Nmedia])
          return 0;

  for(l = &tp->media; *l; l = &(*l)->next)
          ;
  *l = curmedium++;
  return *l;
}

char *parts[] = { "dos", "9fat", 0 };

Medium*
probe(int type, int flag, int dev)
{
  Type *tp;
  int i;
  Medium *mp;
  Dosfile df;
  Dos *dos;
  char **partp;

  for(tp = types; tp->type != Tnil; tp++){
          if(type != Tany && type != tp->type)
                  continue;

          if(flag != Fnone){
                  for(mp = tp->media; mp; mp = mp->next){
                          if((flag & mp->flag) && (dev == Dany || dev == mp->dev))
                                  return mp;
                  }
          }

          if((tp->flag & Fprobe) == 0){
                  tp->flag |= Fprobe;
                  tp->mask = (*tp->init)();
          }

          for(i = 0; tp->mask; i++){
                  if((tp->mask & (1<<i)) == 0)
                          continue;
                  tp->mask &= ~(1<<i);

                  if((mp = allocm(tp)) == 0)
                          continue;

                  mp->dev = i;
                  mp->flag = tp->flag;
                  mp->type = tp;
                  (*tp->initdev)(i, mp->name);

                  if(mp->flag & Fini){
                          mp->flag &= ~Fini;
                          for(partp = parts; *partp; partp++){
                                  if((dos = (*tp->getdospart)(i, *partp)) == nil)
                                          continue;

                                  for(ini = inis; *ini; ini++){
                                          if(dosstat(dos, *ini, &df) > 0){
                                                  mp->inidos = dos;
                                                  mp->part = *partp;
                                                  mp->ini = *ini;
                                                  mp->flag |= Fini;
                                                  goto Break2;
                                          }
                                  }
                          }
                  }
          Break2:
                  if((flag & mp->flag) && (dev == Dany || dev == i))
                          return mp;
          }
  }

  return 0;
}

void
main(void)
{
  Medium *mp;
  int flag, i, mode, tried;
  char def[2*NAMELEN], line[80], *p, *file;
  Type *tp;

  i8042a20();
  memset(m, 0, sizeof(Mach));
  trapinit();
  clockinit();
  alarminit();
  spllo();

  if((ulong)&end > (KZERO|(640*1024)))
          panic("i'm too big\n");

  for(tp = types; tp->type != Tnil; tp++){
          if(tp->type == Tether)
                  continue;
          if((mp = probe(tp->type, Fini, Dany)) && (mp->flag & Fini)){
print("using %s!%s!%s\n", mp->name, mp->part, mp->ini);
                  dotini(mp->inidos);
                  break;
          }
  }

  consinit();

```

```
273     /*
274      * Even after we find the ini file, we keep probing disks,
275      * because we have to collect the partition tables and
276      * have boot devices for parse.
277      */
278     probe(Tany, Fnone, Dany);
279
280     tried = 0;
281     mode = Mauto;
282
283     p = getconf("bootfile");
284
285     if(p != 0) {
286             mode = Manual;
287             for(i = 0; i < NMode; i++){
288                     if(strcmp(p, modes[i].name) == 0){
289                             mode = modes[i].mode;
290                             goto done;
291                     }
292             }
293             if((mp = parse(p, &file)) == nil) {
294                     print("Bad bootfile syntax: %s\n", p);
295                     goto done;
296             }
297             tried = boot(mp, file);
298     }
299   done:
300     if(tried == 0 && mode != Manual){
301             flag = Fany;
302             if(mode == Mlocal)
303                     flag &= ~Fbootp;
304             if((mp = probe(Tany, flag, Dany)) && mp->type->type != Tfloppy)
305                     boot(mp, 0);
306     }
307
308     def[0] = 0;
309     probe(Tany, Fnone, Dany);
310     if(p = getconf("bootdef"))
311             strcpy(def, p);
312
313     flag = 0;
314     for(tp = types; tp->type != Tnil; tp++){
315             for(mp = tp->media; mp; mp = mp->next){
316                     if(flag == 0){
317                             flag = 1;
318                             print("Boot devices:");
319                     }
320                     print(" %s", mp->name);
321             }
322     }
323     if(flag)
324             print("\n");
325
326     for(;;){
327             if(getstr("boot from", line, sizeof(line), def, mode != Manual) >= 0)
328                     if(mp = parse(line, &file))
329                             boot(mp, file);
330             def[0] = 0;
331     }
332   }
333
334   int
335   getfields(char *lp, char **fields, int n, char sep)
336   {
337     int i;
338
339     for(i = 0; lp && *lp && i < n; i++){
340             while(*lp == sep)
341                     *lp++ = 0;
342             if(*lp == 0)
343                     break;
344             fields[i] = lp;
345             while(*lp && *lp != sep){
346                     if(*lp == '\\' && *(lp+1) == '\n')
347                             *lp++ = ' ';
348                     lp++;
349             }
350     }
351     return i;
352   }
353
354   int
355   cistrcmp(char *a, char *b)
356   {
357     int ac, bc;
358
359     for(;;){
360             ac = *a++;
361             bc = *b++;
362
363             if(ac >= 'A' && ac <= 'Z')
364                     ac = 'a' + (ac - 'A');
365             if(bc >= 'A' && bc <= 'Z')
366                     bc = 'a' + (bc - 'A');
367             ac -= bc;
368             if(ac)
369                     return ac;
370             if(bc == 0)
371                     break;
372     }
373     return 0;
374   }
375
376   int
377   cistrncmp(char *a, char *b, int n)
378   {
379     unsigned ac, bc;
380
381     while(n > 0){
382             ac = *a++;
383             bc = *b++;
384             n--;
385
386             if(ac >= 'A' && ac <= 'Z')
387                     ac = 'a' + (ac - 'A');
388             if(bc >= 'A' && bc <= 'Z')
389                     bc = 'a' + (bc - 'A');
390
391             ac -= bc;
392             if(ac)
393                     return ac;
394             if(bc == 0)
395                     break;
396     }
397
398     return 0;
399   }
400
401   void*
402   ialloc(ulong n, int align)
403   {
404
405     static ulong palloc;
406     ulong p;
407     int a;
408
```

```
409     if(palloc == 0)
410             palloc = 3*1024*1024;
411
412     p = palloc;
413     if(align <= 0)
414             align = 4;
415     if(a = n % align)
416             n += align - a;
417     if(a = p % align)
418             p += align - a;
419
420     palloc = p+n;
421
422     return memset((void*)(p|KZERO), 0, n);
423   }
424
425   static Block *allocbp;
426
427   Block*
428   allocb(int size)
429   {
430     Block *bp, **lbp;
431     ulong addr;
432
433     lbp = &allocbp;
434     for(bp = *lbp; bp; bp = bp->next){
435             if((bp->lim - bp->base) >= size){
436                     *lbp = bp->next;
437                     break;
438             }
439             lbp = &bp->next;
440     }
441     if(bp == 0){
442             bp = ialloc(sizeof(Block)+size+64, 0);
443             addr = (ulong)bp;
444             addr = ROUNDUP(addr + sizeof(Block), 8);
445             bp->base = (uchar*)addr;
446             bp->lim = ((uchar*)bp) + sizeof(Block)+size+64;
447     }
448
449     if(bp->flag)
450             panic("allocb reuse\n");
451
452     bp->rp = bp->base;
453     bp->wp = bp->rp;
454     bp->next = 0;
455     bp->flag = 1;
456
457     return bp;
458   }
459
460   void
461   freeb(Block* bp)
462   {
463     bp->next = allocbp;
464     allocbp = bp;
465
466     bp->flag = 0;
467   }
468
469   enum {
470     Paddr=          0x70,   /* address port */
471     Pdata=          0x71,   /* data port */
472   };
473
474   uchar
475   nvramread(int offset)
476   {
477     outb(Paddr, offset);
478     return inb(Pdata);
479   }
480
481   void (*etherdetach)(void);
482
483   void
484   warp9(ulong entry)
485   {
486     if(etherdetach)
487             etherdetach();
488     consdrain();
489     (*(void(*)(void))(PADDR(entry)))();
490   }
```

```
#include  "u.h"
#include  "lib.h"
#include  "mem.h"
#include  "dat.h"
#include  "fns.h"
#include  "io.h"
#include  "ureg.h"

void      intr0(void), intr1(void), intr2(void), intr3(void);
void      intr4(void), intr5(void), intr6(void), intr7(void);
void      intr8(void), intr9(void), intr10(void), intr11(void);
void      intr12(void), intr13(void), intr14(void), intr15(void);
void      intr16(void);
void      intr24(void), intr25(void), intr26(void), intr27(void);
void      intr28(void), intr29(void), intr30(void), intr31(void);
void      intr32(void), intr33(void), intr34(void), intr35(void);
void      intr36(void), intr37(void), intr38(void), intr39(void);
void      intr64(void);
void      intrbad(void);

/*
 *  8259 interrupt controllers
 */
enum
{
  Int0ctl=        0x20,           /* control port (ICW1, OCW2, OCW3) */
  Int0aux=        0x21,           /* everything else (ICW2, ICW3, ICW4, OCW1) */
  Int1ctl=        0xA0,           /* control port */
  Int1aux=        0xA1,           /* everything else (ICW2, ICW3, ICW4, OCW1) */

  Icw1=           0x10,           /* select bit in ctl register */
  Ocw2=           0x00,
  Ocw3=           0x08,

  EOI=            0x20,           /* non-specific end of interrupt */
};

int       int0mask = 0xff;        /* interrupts enabled for first 8259 */
int       int1mask = 0xff;        /* interrupts enabled for second 8259 */

/*
 *  trap/interrupt gates
 */
Segdesc ilt[256];

enum
{
  Maxhandler=     32,             /* max number of interrupt handlers */
};

typedef struct Handler    Handler;
struct Handler
{
  void    (*r)(Ureg*, void*);
  void    *arg;
  Handler *next;
};

struct
{
  Handler *ivec[256];
  Handler h[Maxhandler];
  int     nextfree;
} halloc;

void
sethvec(int v, void (*r)(void), int type, int pri)
{
  ilt[v].d0 = ((ulong)r)&0xFFFF|(KESEL<<16);
  ilt[v].d1 = ((ulong)r)&0xFFFF0000|SEGP|SEGPL(pri)|type;
}

void
setvec(int v, void (*r)(Ureg*, void*), void *arg)
{
  Handler *h;

  if(halloc.nextfree >= Maxhandler)
          panic("out of interrupt handlers");
  h = &halloc.h[halloc.nextfree++];
  h->next = halloc.ivec[v];
  h->r = r;
  h->arg = arg;
  halloc.ivec[v] = h;

  /*
   *  enable corresponding interrupt in 8259
   */
  if((v&~0x7) == VectorPIC){
          int0mask &= ~(1<<(v&7));
          outb(Int0aux, int0mask);
  } else if((v&~0x7) == VectorPIC+8){
          int1mask &= ~(1<<(v&7));
          outb(Int1aux, int1mask);
  }
}

/*
 *  set up the interrupt/trap gates
 */
void
trapinit(void)
{
  int i;

  /*
   *  set all interrupts to panics
   */
  for(i = 0; i < 256; i++)
          sethvec(i, intrbad, SEGTG, 0);

  /*
   *  80386 processor (and coprocessor) traps
   */
  sethvec(0, intr0, SEGTG, 0);
  sethvec(1, intr1, SEGTG, 0);
  sethvec(2, intr2, SEGTG, 0);
  sethvec(3, intr3, SEGTG, 0);
  sethvec(4, intr4, SEGTG, 0);
  sethvec(5, intr5, SEGTG, 0);
  sethvec(6, intr6, SEGTG, 0);
  sethvec(7, intr7, SEGTG, 0);
  sethvec(8, intr8, SEGTG, 0);
  sethvec(9, intr9, SEGTG, 0);
  sethvec(10, intr10, SEGTG, 0);
  sethvec(11, intr11, SEGTG, 0);
  sethvec(12, intr12, SEGTG, 0);
  sethvec(13, intr13, SEGTG, 0);
  sethvec(14, intr14, SEGTG, 0);
  sethvec(15, intr15, SEGTG, 0);
  sethvec(16, intr16, SEGTG, 0);

  /*
   *  device interrupts
   */
  sethvec(24, intr24, SEGIG, 0);
```

```
    sethvec(25, intr25, SEGIG, 0);
    sethvec(26, intr26, SEGIG, 0);
    sethvec(27, intr27, SEGIG, 0);
    sethvec(28, intr28, SEGIG, 0);
    sethvec(29, intr29, SEGIG, 0);
    sethvec(30, intr30, SEGIG, 0);
    sethvec(31, intr31, SEGIG, 0);
    sethvec(32, intr32, SEGIG, 0);
    sethvec(33, intr33, SEGIG, 0);
    sethvec(34, intr34, SEGIG, 0);
    sethvec(35, intr35, SEGIG, 0);
    sethvec(36, intr36, SEGIG, 0);
    sethvec(37, intr37, SEGIG, 0);
    sethvec(38, intr38, SEGIG, 0);
    sethvec(39, intr39, SEGIG, 0);

    /*
     *  tell the hardware where the table is (and how long)
     */
    putidt(ilt, sizeof(ilt));

    /*
     *  Set up the first 8259 interrupt processor.
     *  Make 8259 interrupts start at CPU vector VectorPIC.
     *  Set the 8259 as master with edge triggered
     *  input with fully nested interrupts.
     */
    outb(Int0ctl, Icw1|0x01);       /* ICW1 - edge triggered, master,
                                       ICW4 will be sent */
    outb(Int0aux, VectorPIC);               /* ICW2 - interrupt vector offset */
    outb(Int0aux, 0x04);            /* ICW3 - have slave on level 2 */
    outb(Int0aux, 0x01);            /* ICW4 - 8086 mode, not buffered */

    /*
     *  Set up the second 8259 interrupt processor.
     *  Make 8259 interrupts start at CPU vector VectorPIC+8.
     *  Set the 8259 as master with edge triggered
     *  input with fully nested interrupts.
     */
    outb(Int1ctl, Icw1|0x01);       /* ICW1 - edge triggered, master,
                                       ICW4 will be sent */
    outb(Int1aux, VectorPIC+8);             /* ICW2 - interrupt vector offset */
    outb(Int1aux, 0x02);            /* ICW3 - I am a slave on level 2 */
    outb(Int1aux, 0x01);            /* ICW4 - 8086 mode, not buffered */
    outb(Int1aux, int1mask);

    /*
     *  pass #2 8259 interrupts to #1
     */
    int0mask &= ~0x04;
    outb(Int0aux, int0mask);

    /*
     * Set Ocw3 to return the ISR when ctl read.
     */
    outb(Int0ctl, Ocw3|0x03);
    outb(Int1ctl, Ocw3|0x03);
}

/*
 *  dump registers
 */
static void
dumpregs(Ureg *ur)
{
  print("FLAGS=%lux TRAP=%lux ECODE=%lux PC=%lux\n",
          ur->flags, ur->trap, ur->ecode, ur->pc);
  print("  AX %8.8lux  BX %8.8lux  CX %8.8lux  DX %8.8lux\n",
          ur->ax, ur->bx, ur->cx, ur->dx);
  print("  SI %8.8lux  DI %8.8lux  BP %8.8lux\n",
          ur->si, ur->di, ur->bp);
  print("  CS %4.4ux DS %4.4ux  ES %4.4ux  FS %4.4ux  GS %4.4ux\n",
          ur->cs & 0xFF, ur->ds & 0xFFFF, ur->es & 0xFFFF, ur->fs & 0xFFFF, ur->gs & 0xFFFF);
  print("  CR0 %8.8lux CR2 %8.8lux CR3 %8.8lux\n",
          getcr0(), getcr2(), getcr3());
}

/*
 *  All traps
 */
void
trap(Ureg *ur)
{
  int v;
  int c;
  Handler *h;
  ushort isr;

  v = ur->trap;
  /*
   *  tell the 8259 that we're done with the
   *  highest level interrupt (interrupts are still
   *  off at this point)
   */
  c = v&~0x7;
  isr = 0;
  if(c==VectorPIC || c==VectorPIC+8){
          isr = inb(Int0ctl);
          outb(Int0ctl, EOI);
          if(c == VectorPIC+8){
                  isr |= inb(Int1ctl)<<8;
                  outb(Int1ctl, EOI);
          }
  }

  if(v>=256 || (h = halloc.ivec[v]) == 0){
          if(v >= VectorPIC && v < VectorPIC+16){
                  v -= VectorPIC;
                  /*
                   * Check for a default IRQ7. This can happen when
                   * the IRQ input goes away before the acknowledge.
                   * In this case, a 'default IRQ7' is generated, but
                   * the corresponding bit in the ISR isn't set.
                   * In fact, just ignore all such interrupts.
                   */
                  if(isr & (1<<v))
                          print("unknown interrupt %d pc=0x%lux\n", v, ur->pc);
                  return;
          }

          switch(v){

          case 0x02:                              /* NMI */
                  print("NMI: nmisc=0x%2.2ux, nmiertc=0x%2.2ux, nmiesc=0x%2.2ux\n",
                          inb(0x61), inb(0x70), inb(0x461));
                  return;

          default:
                  print("exception/interrupt %d\n", v);
                  dumpregs(ur);
                  //spllo();
                  print("^P to reset\n");
                  for(;;);
          }
  }

```

```
273     /*
274      *  call the trap routines
275      */
276     do {
277             (*h->r)(ur, h->arg);
278             h = h->next;
279     } while(h);
280   }
```

```
1     typedef struct Ureg       Ureg;
2
3     struct Ureg
4     {
5       ulong   di;             /* general registers */
6       ulong   si;             /* ... */
7       ulong   bp;             /* ... */
8       ulong   nsp;
9       ulong   bx;             /* ... */
10      ulong   dx;             /* ... */
11      ulong   cx;             /* ... */
12      ulong   ax;             /* ... */
13      ulong   gs;             /* data segments */
14      ulong   fs;             /* ... */
15      ulong   es;             /* ... */
16      ulong   ds;             /* ... */
17      ulong   trap;           /* trap type */
18      ulong   ecode;          /* error code (or zero) */
19      ulong   pc;             /* pc */
20      ulong   cs;             /* old context */
21      ulong   flags;          /* old flags */
22      union {
23              ulong   usp;
24              ulong   sp;
25      };
26      ulong   ss;             /* old stack segment */
27    };
```

```
#include <u.h>
#include <libc.h>

/*
 * In place, rewrite name to compress multiple /, eliminate ., and process ..
 */
#define SEP(x)    ((x)=='/' || (x) == 0)
char*
cleanname(char *name)
{
  char *p, *q, *dotdot;
  int rooted;

  rooted = name[0] == '/';

  /*
   * invariants:
   *      p points at beginning of path element we're considering.
   *      q points just past the last path element we wrote (no slash).
   *      dotdot points just past the point where .. cannot backtrack
   *              any further (no slash).
   */
  p = q = dotdot = name+rooted;
  while(*p) {
          if(p[0] == '/') /* null element */
                  p++;
          else if(p[0] == '.' && SEP(p[1]))
                  p += 1; /* don't count the separator in case it is nul */
          else if(p[0] == '.' && p[1] == '.' && SEP(p[2])) {
                  p += 2;
                  if(q > dotdot) {        /* can backtrack */
                          while(--q > dotdot && *q != '/')
                                  ;
                  } else if(!rooted) {    /* /.. is / but ./../ is .. */
                          if(q != name)
                                  *q++ = '/';
                          *q++ = '.';
                          *q++ = '.';
                          dotdot = q;
                  }
          } else {        /* real path element */
                  if(q != name+rooted)
                          *q++ = '/';
                  while((*q = *p) != '/' && *q != 0)
                          p++, q++;
          }
  }
  if(q == name)   /* empty string is really ``.'' */
          *q++ = '.';
  *q = '\0';
  return name;
}
```

```
/*
 * This allocator takes blocks from a coarser allocator (p->alloc) and
 * uses them as arenas.
 *
 * An arena is split into a sequence of blocks of variable size.  The
 * blocks begin with a Bhdr that denotes the length (including the Bhdr)
 * of the block.  An arena begins with an Arena header block (Arena,
 * ARENA_MAGIC) and ends with a Bhdr block with magic ARENATAIL_MAGIC and
 * size 0.  Intermediate blocks are either allocated or free.  At the end
 * of each intermediate block is a Btail, which contains information
 * about where the block starts.  This is useful for walking backwards.
 *
 * Free blocks (Free*) have a magic value of FREE_MAGIC in their Bhdr
 * headers.  They are kept in a binary tree (p->freeroot) traversible by
 * walking ->left and ->right.  Each node of the binary tree is a pointer
 * to a circular doubly-linked list (next, prev) of blocks of identical
 * size.  Blocks are added to this ``tree of lists'' by pooladd(), and
 * removed by pooldel().
 *
 * When freed, adjacent blocks are coalesced to create larger blocks when
 * possible.
 *
 * Allocated blocks (Alloc*) have one of two magic values: KEMPT_MAGIC or
 * UNKEMPT_MAGIC.  When blocks are released from the pool, they have
 * magic value UNKEMPT_MAGIC.  Once the block has been trimmed by kemb()
 * and the amount of user-requested data has been recorded in the
 * datasize field of the tail, the magic value is changed to KEMPT_MAGIC.
 * All blocks returned to callers should be of type KEMPT_MAGIC, as
 * should all blocks passed to us by callers.  The amount of data the user
 * asked us for can be found by subtracting the short in tail->datasize
 * from header->size.  Further, the up to at most four bytes between the
 * end of the user-requested data block and the actual Btail structure are
 * marked with a magic value, which is checked to detect user overflow.
 *
 * The arenas returned by p->alloc are kept in a doubly-linked list
 * (p->arenalist) running through the arena headers, sorted by descending
 * base address (prev, next).  When a new arena is allocated, we attempt
 * to merge it with its two neighbors via p->merge.
 */

#include <u.h>
#include <libc.h>
#include <pool.h>

typedef struct Alloc      Alloc;
typedef struct Arena      Arena;
typedef struct Bhdr       Bhdr;
typedef struct Btail      Btail;
typedef struct Free       Free;

struct Bhdr {
  ulong   magic;
  ulong   size;
};
enum {
  NOT_MAGIC = 0xdeadfa11,
};
#define B2NB(b) ((Bhdr*)((uchar*)(b)+(b)->size))

#define SHORT(x) (((x)[0] << 8) | (x)[1])
#define PSHORT(p, x) \
  (((uchar*)(p))[0] = ((x)>>8)&0xFF, \
  ((uchar*)(p))[1] = (x)&0xFF)

enum {
  TAIL_MAGIC0 = 0xBE,
  TAIL_MAGIC1 = 0xEF
};

struct Btail {
  uchar   magic0;
  uchar   datasize[2];
  uchar   magic1;
  ulong   size;   /* same as Bhdr->size */
};
#define B2T(b)    ((Btail*)((uchar*)(b)+(b)->size-sizeof(Btail)))
#define B2PT(b) ((Btail*)((uchar*)(b)-sizeof(Btail)))
#define T2HDR(t) ((Bhdr*)((uchar*)(t)+sizeof(Btail)-(t)->size))
struct Free {
                  Bhdr;
  Free*   left;
  Free*   right;
  Free*   next;
  Free*   prev;
};
enum {
  FREE_MAGIC = 0xBA5EBA11,
};

/*
 * the point of the notused fields is to make 8c differentiate
 * between Bhdr and Allocblk, and between Kempt and Unkempt.
 */
struct Alloc {
                  Bhdr;
};
enum {
  KEMPT_MAGIC = 0x0A110C09,
  UNKEMPT_MAGIC = 0xCAB00D1E+1,
};

struct Arena {
                  Bhdr;
  Arena*  aup;
  Arena*  down;
  ulong   asize;
  ulong   pad;    /* to a multiple of 8 bytes */
};
enum {
  ARENA_MAGIC = 0xC0A1E5CE+1,
  ARENATAIL_MAGIC = 0xEC5E1A0C+1,
};
#define A2TB(a)   ((Bhdr*)((uchar*)(a)+(a)->asize-sizeof(Bhdr)))
#define A2B(a)    B2NB(a)

enum {
  MINBLOCKSIZE = sizeof(Free)+sizeof(Btail)
};

static uchar datamagic[] = { 0xFE, 0xF1, 0xF0, 0xFA };

#define   Poison  (void*)0xCafeBabe

#define _B2D(a)   ((void*)((uchar*)a+sizeof(Bhdr)))
#define _D2B(v)   ((Alloc*)((uchar*)v-sizeof(Bhdr)))

// static void*   _B2D(void*);
// static void*   _D2B(void*);
static void*      B2D(Pool*, Alloc*);
static Alloc*     D2B(Pool*, void*);
static Arena*     arenamerge(Pool*, Arena*, Arena*);
static void               blockcheck(Pool*, Bhdr*);
static Alloc*     blockmerge(Pool*, Bhdr*, Bhdr*);
static Alloc*     blocksetdsize(Pool*, Alloc*, ulong);
static Bhdr*      blocksetsize(Bhdr*, ulong);
static ulong      bsize2asize(Pool*, ulong);
static ulong      dsize2bsize(Pool*, ulong);
```

```
137  static ulong      getdsize(Alloc*);
138  static Alloc*     kemb(Pool*, Alloc*, ulong);
139  static Free*      listadd(Free*, Free*);
140  static Free*      listdelete(Free*, Free*);
141  static void               logstack(Pool*);
142  static Free**     ltreewalk(Free**, ulong);
143  static void               memmark(void*, int, ulong);
144  static Free*      pooladd(Pool*, Alloc*);
145  static void*      poolallocl(Pool*, ulong);
146  static void               poolcheckl(Pool*);
147  static void               poolcheckarena(Pool*, Arena*);
148  static int                poolcompactl(Pool*);
149  static Alloc*     pooldel(Pool*, Free*);
150  static void               pooldumpl(Pool*);
151  static void               pooldumparena(Pool*, Arena*);
152  static void               poolfreel(Pool*, void*);
153  static void               poolnewarena(Pool*, ulong);
154  static void*      poolreallocl(Pool*, void*, ulong);
155  static Free*      treedelete(Free*, Free*);
156  static Free*      treeinsert(Free*, Free*);
157  static Free*      treelookup(Free*, ulong);
158  static Free*      treelookupgt(Free*, ulong);
159
160  /*
161   * Debugging
162   *
163   * Antagonism causes blocks to always be filled with garbage if their
164   * contents are undefined.  This tickles both programs and the library.
165   * It's a linear time hit but not so noticeable during nondegenerate use.
166   * It would be worth leaving in except that it negates the benefits of the
167   * kernel's demand-paging.  The tail magic and end-of-data magic
168   * provide most of the user-visible benefit that antagonism does anyway.
169   *
170   * Paranoia causes the library to recheck the entire pool on each lock
171   * or unlock.  A failed check on unlock means we tripped over ourselves,
172   * while a failed check on lock tends to implicate the user.  Paranoia has
173   * the potential to slow things down a fair amount for pools with large
174   * numbers of allocated blocks.  It completely negates all benefits won
175   * by the binary tree.  Turning on paranoia in the kernel makes it painfully
176   * slow.
177   *
178   * Verbosity induces the dumping of the pool via p->print at each lock operation.
179   * By default, only one line is logged for each alloc, free, and realloc.
180   */
181
182  /* the if(!x);else avoids ``dangling else'' problems */
183  #define antagonism        if(!(p->flags & POOL_ANTAGONISM));else
184  #define paranoia  if(!(p->flags & POOL_PARANOIA));else
185  #define verbosity if(!(p->flags & POOL_VERBOSITY));else
186
187  #define DPRINT    if(!(p->flags & POOL_DEBUGGING));else p->print
188  #define LOG               if(!(p->flags & POOL_LOGGING));else p->print
189
190  /*
191   * Tree walking
192   */
193
194  /* ltreewalk: return address of pointer to node of size == size */
195  static Free**
196  ltreewalk(Free **t, ulong size)
197  {
198    assert(t != nil /* ltreewalk */);
199
200    for(;;) {
201            if(*t == nil)
202                    return t;
203
204            assert((*t)->magic == FREE_MAGIC);
205
206            if(size == (*t)->size)
207                    return t;
208            if(size < (*t)->size)
209                    t = &(*t)->left;
210            else
211                    t = &(*t)->right;
212    }
213    return nil;     /* not reached */
214  }
215
216  /* treelookup: find node in tree with size == size */
217  static Free*
218  treelookup(Free *t, ulong size)
219  {
220    return *ltreewalk(&t, size);
221  }
222
223  /* treeinsert: insert node into tree */
224  static Free*
225  treeinsert(Free *tree, Free *node)
226  {
227    Free **loc, *repl;
228
229    assert(node != nil /* treeinsert */);
230
231    loc = ltreewalk(&tree, node->size);
232    if(*loc == nil) {
233            node->left = nil;
234            node->right = nil;
235    } else {        /* replace existing node */
236            repl = *loc;
237            node->left = repl->left;
238            node->right = repl->right;
239    }
240    *loc = node;
241    return tree;
242  }
243
244  /* treedelete: remove node from tree */
245  static Free*
246  treedelete(Free *tree, Free *node)
247  {
248    Free **loc, **lsucc, *succ;
249
250    assert(node != nil /* treedelete */);
251
252    loc = ltreewalk(&tree, node->size);
253    assert(*loc == node);
254
255    if(node->left == nil)
256            *loc = node->right;
257    else if(node->right == nil)
258            *loc = node->left;
259    else {
260            /* have two children, use inorder successor as replacement */
261            for(lsucc = &node->right; (*lsucc)->left; lsucc = &(*lsucc)->left)
262                    ;
263            succ = *lsucc;
264            *lsucc = succ->right;
265            succ->left = node->left;
266            succ->right = node->right;
267            *loc = succ;
268    }
269
270    node->left = node->right = Poison;
271    return tree;
272  }
```

```
273
274   /* treelookupgt: find smallest node in tree with size >= size */
275   static Free*
276   treelookupgt(Free *t, ulong size)
277   {
278     Free *lastgood; /* last node we saw that was big enough */
279
280     lastgood = nil;
281     for(;;) {
282             if(t == nil)
283                     return lastgood;
284             if(size == t->size)
285                     return t;
286             if(size < t->size) {
287                     lastgood = t;
288                     t = t->left;
289             } else {
290                     t = t->right;
291             }
292     }
293     return nil;     /* not reached */
294   }
295
296   /*
297    * List maintenance
298    */
299
300   /* listadd: add a node to a doubled linked list */
301   static Free*
302   listadd(Free *list, Free *node)
303   {
304     if(list == nil) {
305             node->next = node;
306             node->prev = node;
307             return node;
308     }
309
310     node->prev = list->prev;
311     node->next = list;
312
313     node->prev->next = node;
314     node->next->prev = node;
315
316     return list;
317   }
318
319   /* listdelete: remove node from a doubly linked list */
320   static Free*
321   listdelete(Free *list, Free *node)
322   {
323     if(node->next == node) {        /* singular list */
324             node->prev = node->next = Poison;
325             return nil;
326     }
327
328     node->next->prev = node->prev;
329     node->prev->next = node->next;
330
331     if(list == node)
332             list = node->next;
333
334     node->prev = node->next = Poison;
335     return list;
336   }
337
338   /*
339    * Pool maintenance
340    */
341
342   /* pooladd: add anode to the free pool */
343   static Free*
344   pooladd(Pool *p, Alloc *anode)
345   {
346     Free *lst, *olst;
347     Free *node;
348     Free **parent;
349
350     antagonism {
351             memmark(_B2D(anode), 0xF7, anode->size-sizeof(Bhdr)-sizeof(Btail));
352     }
353
354     node = (Free*)anode;
355     node->magic = FREE_MAGIC;
356     parent = ltreewalk(&p->freeroot, node->size);
357     olst = *parent;
358     lst = listadd(olst, node);
359     if(olst != lst) /* need to update tree */
360             *parent = treeinsert(*parent, lst);
361     p->curfree += node->size;
362     return node;
363   }
364
365   /* pooldel: remove node from the free pool */
366   static Alloc*
367   pooldel(Pool *p, Free *node)
368   {
369     Free *lst, *olst;
370     Free **parent;
371
372     parent = ltreewalk(&p->freeroot, node->size);
373     olst = *parent;
374     assert(olst != nil /* pooldel */);
375
376     lst = listdelete(olst, node);
377     if(lst == nil)
378             *parent = treedelete(*parent, olst);
379     else if(lst != olst)
380             *parent = treeinsert(*parent, lst);
381
382     node->left = node->right = Poison;
383     p->curfree -= node->size;
384
385     antagonism {
386             memmark(_B2D(node), 0xF9, node->size-sizeof(Bhdr)-sizeof(Btail));
387     }
388
389     node->magic = UNKEMPT_MAGIC;
390     return (Alloc*)node;
391   }
392
393   /*
394    * Block maintenance
395    */
396   /* block allocation */
397   static ulong
398   dsize2bsize(Pool *p, ulong sz)
399   {
400     sz += sizeof(Bhdr)+sizeof(Btail);
401     if(sz < p->minblock)
402             sz = p->minblock;
403     sz = (sz+p->quantum-1)&~(p->quantum-1);
404     return sz;
405   }
406
407   static ulong
408   bsize2asize(Pool *p, ulong sz)
```

```
{
  sz += sizeof(Arena)+sizeof(Btail);
  if(sz < p->minarena)
          sz = p->minarena;
  sz = (sz+p->quantum)&~(p->quantum-1);
  return sz;
}

/* blockmerge: merge a and b, known to be adjacent */
/* both are removed from pool if necessary. */
static Alloc*
blockmerge(Pool *pool, Bhdr *a, Bhdr *b)
{
  Btail *t;

  assert(B2NB(a) == b);

  if(a->magic == FREE_MAGIC)
          pooldel(pool, (Free*)a);
  if(b->magic == FREE_MAGIC)
          pooldel(pool, (Free*)b);

  t = B2T(a);
  t->size = (ulong)Poison;
  t->magic0 = NOT_MAGIC;
  t->magic1 = NOT_MAGIC;
  PSHORT(t->datasize, NOT_MAGIC);

  a->size += b->size;
  t = B2T(a);
  t->size = a->size;
  PSHORT(t->datasize, 0xFFFF);

  b->size = NOT_MAGIC;
  b->magic = NOT_MAGIC;

  a->magic = UNKEMPT_MAGIC;
  return (Alloc*)a;
}

/* blocksetsize: set the total size of a block, fixing tail pointers */
static Bhdr*
blocksetsize(Bhdr *b, ulong bsize)
{
  Btail *t;

  assert(b->magic != FREE_MAGIC /* blocksetsize */);

  b->size = bsize;
  t = B2T(b);
  t->size = b->size;
  t->magic0 = TAIL_MAGIC0;
  t->magic1 = TAIL_MAGIC1;
  return b;
}

/* getdsize: return the requested data size for an allocated block */
static ulong
getdsize(Alloc *b)
{
  Btail *t;
  t = B2T(b);
  return b->size - SHORT(t->datasize);
}

/* blocksetdsize: set the user data size of a block */
static Alloc*
blocksetdsize(Pool *p, Alloc *b, ulong dsize)
{
  Btail *t;
  uchar *q, *eq;

  assert(b->size >= dsize2bsize(p, dsize));
  assert(b->size - dsize < 0x10000);

  t = B2T(b);
  PSHORT(t->datasize, b->size - dsize);

  q=(uchar*)_B2D(b)+dsize;
  eq = (uchar*)t;
  if(eq > q+4)
          eq = q+4;
  for(; q<eq; q++)
          *q = datamagic[((ulong)q)%nelem(datamagic)];

  return b;
}

/* kemb: trim a block down to what is needed to hold dsize bytes of user data */
static Alloc*
kemb(Pool *p, Alloc *b, ulong dsize)
{
  ulong extra, bsize;
  Alloc *frag;

  bsize = dsize2bsize(p, dsize);
  extra = b->size - bsize;
  if(b->size - dsize >= 0x10000 ||
    (extra >= bsize>>2 && extra >= MINBLOCKSIZE && extra >= p->minblock)) {
          blocksetsize(b, bsize);
          frag = (Alloc*) B2NB(b);

          antagonism {
                  memmark(frag, 0xF1, extra);
          }

          frag->magic = UNKEMPT_MAGIC;
          blocksetsize(frag, extra);
          pooladd(p, frag);
  }

  b->magic = KEMPT_MAGIC;
  blocksetdsize(p, b, dsize);
  return b;
}

/*
 * Arena maintenance
 */

/* arenasetsize: set arena size, updating tail */
static void
arenasetsize(Arena *a, ulong asize)
{
  Bhdr *atail;

  a->asize = asize;
  atail = A2TB(a);
  atail->magic = ARENATAIL_MAGIC;
  atail->size = 0;
}

/* poolnewarena: allocate new arena */
static void
poolnewarena(Pool *p, ulong asize)
{
```

```
545     Arena *a;
546     Arena *ap, *lastap;
547     Alloc *b;
548
549     LOG(p, "newarena %lud\n", asize);
550     if(p->cursize+asize > p->maxsize) {
551             if(poolcompactl(p) == 0)
552                     LOG(p, "pool too big: %lud+%lud > %lud\n",
553                             p->cursize, asize, p->maxsize);
554             return;
555     }
556
557     if((a = p->alloc(asize)) == nil) {
558             /* assume errstr set by p->alloc */
559             return;
560     }
561
562     p->cursize += asize;
563
564     /* arena hdr */
565     a->magic = ARENA_MAGIC;
566     blocksetsize(a, sizeof(Arena));
567     arenasetsize(a, asize);
568     blockcheck(p, a);
569
570     /* create one large block in arena */
571     b = (Alloc*)A2B(a);
572     b->magic = UNKEMPT_MAGIC;
573     blocksetsize(b, (uchar*)A2TB(a)-(uchar*)b);
574     blockcheck(p, b);
575     pooladd(p, b);
576     blockcheck(p, b);
577
578     /* sort arena into descending sorted arena list */
579     for(lastap=nil, ap=p->arenalist; ap > a; lastap=ap, ap=ap->down)
580             ;
581
582     if(a->down = ap)        /* assign = */
583             a->down->aup = a;
584
585     if(a->aup = lastap)     /* assign = */
586             a->aup->down = a;
587     else
588             p->arenalist = a;
589
590     /* merge with surrounding arenas if possible */
591     /* must do a with up before down with a (think about it) */
592     if(a->aup)
593             arenamerge(p, a, a->aup);
594     if(a->down)
595             arenamerge(p, a->down, a);
596   }
597
598   /* blockresize: grow a block to encompass space past its end, possibly by */
599   /* trimming it into two different blocks. */
600   static void
601   blockgrow(Pool *p, Bhdr *b, ulong nsize)
602   {
603     if(b->magic == FREE_MAGIC) {
604             Alloc *a;
605             Bhdr *bnxt;
606             a = pooldel(p, (Free*)b);
607             blockcheck(p, a);
608             blocksetsize(a, nsize);
609             blockcheck(p, a);
610             bnxt = B2NB(a);
611             if(bnxt->magic == FREE_MAGIC)
612                     a = blockmerge(p, a, bnxt);
613             blockcheck(p, a);
614             pooladd(p, a);
615     } else {
616             Alloc *a;
617             ulong dsize;
618
619             a = (Alloc*)b;
620             dsize = getdsize(a);
621             blocksetsize(a, nsize);
622             kemb(p, a, dsize);
623     }
624   }
625
626   /* arenamerge: attempt to coalesce to arenas that might be adjacent */
627   static Arena*
628   arenamerge(Pool *p, Arena *bot, Arena *top)
629   {
630     Bhdr *bbot, *btop;
631     Btail *t;
632
633     blockcheck(p, bot);
634     blockcheck(p, top);
635     assert(bot->aup == top && top > bot);
636
637     if(p->merge == nil || p->merge(bot, top) == 0)
638             return nil;
639
640     /* remove top from list */
641     if(bot->aup = top->aup) /* assign = */
642             bot->aup->down = bot;
643     else
644             p->arenalist = bot;
645
646     /* save ptrs to last block in bot, first block in top */
647     t = B2PT(A2TB(bot));
648     bbot = T2HDR(t);
649     btop = A2B(top);
650     blockcheck(p, bbot);
651     blockcheck(p, btop);
652
653     /* grow bottom arena to encompass top */
654     arenasetsize(bot, top->asize + ((uchar*)top - (uchar*)bot));
655
656     /* grow bottom block to encompass space between arenas */
657     blockgrow(p, bbot, (uchar*)btop-(uchar*)bbot);
658     blockcheck(p, bbot);
659     return bot;
660   }
661
662   /* dumpblock: print block's vital stats */
663   static void
664   dumpblock(Pool *p, Bhdr *b)
665   {
666     ulong *dp;
667     ulong dsize;
668     uchar *cp;
669
670     dp = (ulong*)b;
671     p->print(p, "pool %s block %p\nhdr %.8lux %.8lux %.8lux %.8lux %.8lux %.8lux\n",
672             p->name, b, dp[0], dp[1], dp[2], dp[3], dp[4], dp[5], dp[6]);
673
674     if(b->size >= 1024*1024*1024)   /* tail pointer corrupt; printing tail will fault */
675             return;
676     dp = (ulong*)B2T(b);
677     p->print(p, "tail %.8lux %.8lux %.8lux %.8lux %.8lux %.8lux\n",
678             dp[-6], dp[-5], dp[-4], dp[-3], dp[-2], dp[-1]);
679
680     if(b->magic == KEMPT_MAGIC){
```

```
681             dsize = getdsize((Alloc*)b);
682             if(dsize >= b->size)    /* user data size corrupt */
683                     return;
684
685             cp = (uchar*)_B2D(b)+dsize;
686             p->print(p, "user data ");
687             p->print(p, "%.2ux %.2ux %.2ux %.2ux  %.2ux %.2ux %.2ux %.2ux",
688                     cp[-8], cp[-7], cp[-6], cp[-5], cp[-4], cp[-3], cp[-2], cp[-1]);
689             p->print(p, " | %.2ux %.2ux %.2ux %.2ux  %.2ux %.2ux %.2ux %.2ux\n",
690                     cp[0], cp[1], cp[2], cp[3], cp[4], cp[5], cp[6], cp[7]);
691     }
692   }
693
694   static void
695   printblock(Pool *p, Bhdr *b, char *msg)
696   {
697     p->print(p, "%s\n", msg);
698     dumpblock(p, b);
699   }
700
701   static void
702   panicblock(Pool *p, Bhdr *b, char *msg)
703   {
704     p->print(p, "%s\n", msg);
705     dumpblock(p, b);
706     p->panic(p, "pool panic");
707   }
708
709   /* blockcheck: ensure a block consistent with our expectations */
710   /* should only be called when holding pool lock */
711   static void
712   blockcheck(Pool *p, Bhdr *b)
713   {
714     Alloc *a;
715     Btail *t;
716     uchar *q, *bq, *eq;
717     ulong dsize;
718
719     switch(b->magic) {
720     default:
721             panicblock(p, b, "bad magic");
722     case FREE_MAGIC:
723     case UNKEMPT_MAGIC:
724             t = B2T(b);
725             if(t->magic0 != TAIL_MAGIC0 || t->magic1 != TAIL_MAGIC1)
726                     panicblock(p, b, "corrupt tail magic");
727             if(T2HDR(t) != b)
728                     panicblock(p, b, "corrupt tail ptr");
729             break;
730     case ARENA_MAGIC:
731             b = A2TB((Arena*)b);
732             if(b->magic != ARENATAIL_MAGIC)
733                     panicblock(p, b, "bad arena size");
734             /* fall through */
735     case ARENATAIL_MAGIC:
736             if(b->size != 0)
737                     panicblock(p, b, "bad arena tail size");
738             break;
739     case KEMPT_MAGIC:
740             a = (Alloc*)b;
741             if(a->size > 1024*1024*1024)
742                     panicblock(p, b, "block too big");
743
744             t = B2T(b);
745             dsize = getdsize(a);
746             bq = (uchar*)_B2D(a)+dsize;
747             eq = (uchar*)t;
748
749             if(t->magic0 != TAIL_MAGIC0){
750                     /* if someone wrote exactly one byte over and it was a NUL, we sometimes on
751                     if((p->flags & POOL_TOLERANCE) && bq == eq && t->magic0 == 0)
752                             printblock(p, b, "mem user overflow (magic0)");
753                     else
754                             panicblock(p, b, "corrupt tail magic0");
755             }
756
757             if(t->magic1 != TAIL_MAGIC1)
758                     panicblock(p, b, "corrupt tail magic1");
759             if(T2HDR(t) != b)
760                     panicblock(p, b, "corrupt tail ptr");
761
762             if(dsize2bsize(p, dsize)  > a->size)
763                     panicblock(p, b, "too much block data");
764
765             if(eq > bq+4)
766                     eq = bq+4;
767             for(q=bq; q<eq; q++){
768                     if(*q != datamagic[((ulong)q)%nelem(datamagic)]){
769                             if(q == bq && *q == 0 && (p->flags & POOL_TOLERANCE)){
770                                     printblock(p, b, "mem user overflow");
771                                     continue;
772                             }
773                             panicblock(p, b, "mem user overflow");
774                     }
775             }
776             break;
777     }
778   }
779
780   /*
781    * compact an arena by shifting all the free blocks to the end.
782    * assumes pool lock is held.
783    */
784   enum {
785     FLOATING_MAGIC = 0xCBCBCBCB,    /* temporarily neither allocated nor in the free tree */
786   };
787
788   static int
789   arenacompact(Pool *p, Arena *a)
790   {
791     Bhdr *b, *wb, *eb, *nxt;
792     int compacted;
793
794     if(p->move == nil)
795             p->panic(p, "don't call me when pool->move is nil\n");
796
797     poolcheckarena(p, a);
798     eb = A2TB(a);
799     compacted = 0;
800     for(b=wb=A2B(a); b && b < eb; b=nxt) {
801             nxt = B2NB(b);
802             switch(b->magic) {
803             case FREE_MAGIC:
804                     pooldel(p, (Free*)b);
805                     b->magic = FLOATING_MAGIC;
806                     break;
807             case KEMPT_MAGIC:
808                     if(wb != b) {
809                             memmove(wb, b, b->size);
810                             p->move(_B2D(b), _B2D(wb));
811                             compacted = 1;
812                     }
813                     wb = B2NB(wb);
814                     break;
815             }
816     }
```

```
817
818     /*
819      * the only free data is now at the end of the arena, pointed
820      * at by wb.  all we need to do is set its size and get out.
821      */
822     if(wb < eb) {
823             wb->magic = UNKEMPT_MAGIC;
824             blocksetsize(wb, (uchar*)eb-(uchar*)wb);
825             pooladd(p, (Alloc*)wb);
826     }
827
828     return compacted;
829   }
830
831   /*
832    * compact a pool by compacting each individual arena.
833    * 'twould be nice to shift blocks from one arena to the
834    * next but it's a pain to code.
835    */
836   static int
837   poolcompactl(Pool *pool)
838   {
839     Arena *a;
840     int compacted;
841
842     if(pool->move == nil || pool->lastcompact == pool->nfree)
843             return 0;
844
845     pool->lastcompact = pool->nfree;
846     compacted = 0;
847     for(a=pool->arenalist; a; a=a->down)
848             compacted |= arenacompact(pool, a);
849     return compacted;
850   }
851
852   /*
853   static int
854   poolcompactl(Pool*)
855   {
856     return 0;
857   }
858   */
859
860   /*
861    * Actual allocators
862    */
863
864   /*
865   static void*
866   _B2D(void *a)
867   {
868     return (uchar*)a+sizeof(Bhdr);
869   }
870   */
871
872   static void*
873   B2D(Pool *p, Alloc *a)
874   {
875     if(a->magic != KEMPT_MAGIC)
876             p->panic(p, "B2D called on unworthy block");
877     return _B2D(a);
878   }
879
880   /*
881   static void*
882   _D2B(void *v)
883   {
884     Alloc *a;
885     a = (Alloc*)((uchar*)v-sizeof(Bhdr));
886     return a;
887   }
888   */
889
890   static Alloc*
891   D2B(Pool *p, void *v)
892   {
893     Alloc *a;
894     a = _D2B(v);
895     if(a->magic != KEMPT_MAGIC)
896             p->panic(p, "D2B called on non-block %p", v);
897     return a;
898   }
899
900   /* poolallocl: attempt to allocate block to hold dsize user bytes; assumes lock held */
901   static void*
902   poolallocl(Pool *p, ulong dsize)
903   {
904     ulong bsize;
905     Free *fb;
906     Alloc *ab;
907
908     if(dsize < 0 || dsize >= 0x80000000UL)  /* for sanity, overflow */
909             return nil;
910
911     bsize = dsize2bsize(p, dsize);
912
913     fb = treelookupgt(p->freeroot, bsize);
914     if(fb == nil) {
915             poolnewarena(p, bsize2asize(p, bsize));
916             if((fb = treelookupgt(p->freeroot, bsize)) == nil) {
917                     /* assume poolnewarena failed and set %r */
918                     return nil;
919             }
920     }
921
922     ab = kemb(p, pooldel(p, fb), dsize);
923     p->curalloc += ab->size;
924     return B2D(p, ab);
925   }
926
927   /* poolreallocl: attempt to grow v to ndsize bytes; assumes lock held */
928   static void*
929   poolreallocl(Pool *p, void *v, ulong ndsize)
930   {
931     Alloc *a;
932     Bhdr *left, *right, *newb;
933     Btail *t;
934     ulong nbsize;
935     ulong odsize;
936     ulong obsize;
937     void *nv;
938
939     if(v == nil)    /* for ANSI */
940             return poolallocl(p, ndsize);
941     if(ndsize == 0) {
942             poolfreel(p, v);
943             return nil;
944     }
945     a = D2B(p, v);
946     blockcheck(p, a);
947     odsize = getdsize(a);
948     obsize = a->size;
949
950     /* can reuse the same block? */
951     nbsize = dsize2bsize(p, ndsize);
952     if(nbsize <= a->size) {
```

```
953     Returnblock:
954             if(v != _B2D(a))
955                     memmove(_B2D(a), v, odsize);
956             a = kemb(p, a, ndsize);
957             p->curalloc -= obsize;
958             p->curalloc += a->size;
959             v = B2D(p, a);
960             return v;
961     }
962
963     /* can merge with surrounding blocks? */
964     right = B2NB(a);
965     if(right->magic == FREE_MAGIC && a->size+right->size >= nbsize) {
966             a = blockmerge(p, a, right);
967             goto Returnblock;
968     }
969
970     t = B2PT(a);
971     left = T2HDR(t);
972     if(left->magic == FREE_MAGIC && left->size+a->size >= nbsize) {
973             a = blockmerge(p, left, a);
974             goto Returnblock;
975     }
976
977     if(left->magic == FREE_MAGIC && right->magic == FREE_MAGIC
978     && left->size+a->size+right->size >= nbsize) {
979             a = blockmerge(p, blockmerge(p, left, a), right);
980             goto Returnblock;
981     }
982
983     if((nv = poolallocl(p, ndsize)) == nil)
984             return nil;
985
986     /* maybe the new block is next to us; if so, merge */
987     left = T2HDR(B2PT(a));
988     right = B2NB(a);
989     newb = D2B(p, nv);
990     if(left == newb || right == newb) {
991             if(left == newb || left->magic == FREE_MAGIC)
992                     a = blockmerge(p, left, a);
993             if(right == newb || right->magic == FREE_MAGIC)
994                     a = blockmerge(p, a, right);
995             assert(a->size >= nbsize);
996             goto Returnblock;
997     }
998
999     /* enough cleverness */
1000    memmove(nv, v, odsize);
1001    poolfreel(p, v);
1002    return nv;
1003  }
1004
1005  /* poolfree: free block obtained from poolalloc; assumes lock held */
1006  static void
1007  poolfreel(Pool *p, void *v)
1008  {
1009    Alloc *ab;
1010    Bhdr *back, *fwd;
1011
1012    if(v == nil)    /* for ANSI */
1013            return;
1014
1015    ab = D2B(p, v);
1016    blockcheck(p, ab);
1017
1018    p->nfree++;
1019    p->curalloc -= ab->size;
1020    back = T2HDR(B2PT(ab));
1021    if(back->magic == FREE_MAGIC)
1022            ab = blockmerge(p, back, ab);
1023
1024    fwd = B2NB(ab);
1025    if(fwd->magic == FREE_MAGIC)
1026            ab = blockmerge(p, ab, fwd);
1027
1028    pooladd(p, ab);
1029  }
1030
1031  void*
1032  poolalloc(Pool *p, ulong n)
1033  {
1034    void *v;
1035
1036    p->lock(p);
1037    paranoia {
1038            poolcheckl(p);
1039    }
1040    verbosity {
1041            pooldumpl(p);
1042    }
1043    v = poolallocl(p, n);
1044    paranoia {
1045            poolcheckl(p);
1046    }
1047    verbosity {
1048            pooldumpl(p);
1049    }
1050    if(p->logstack && (p->flags & POOL_LOGGING)) p->logstack(p);
1051    LOG(p, "poolalloc %p %lud = %p\n", p, n, v);
1052    p->unlock(p);
1053    return v;
1054  }
1055
1056  int
1057  poolcompact(Pool *p)
1058  {
1059    int rv;
1060
1061    p->lock(p);
1062    paranoia {
1063            poolcheckl(p);
1064    }
1065    verbosity {
1066            pooldumpl(p);
1067    }
1068    rv = poolcompactl(p);
1069    paranoia {
1070            poolcheckl(p);
1071    }
1072    verbosity {
1073            pooldumpl(p);
1074    }
1075    LOG(p, "poolcompact %p\n", p);
1076    p->unlock(p);
1077    return rv;
1078  }
1079
1080  void*
1081  poolrealloc(Pool *p, void *v, ulong n)
1082  {
1083    void *nv;
1084
1085    p->lock(p);
1086    paranoia {
1087            poolcheckl(p);
1088    }
```

```
1089    verbosity {
1090            pooldumpl(p);
1091    }
1092    nv = poolreallocl(p, v, n);
1093    paranoia {
1094            poolcheckl(p);
1095    }
1096    verbosity {
1097            pooldumpl(p);
1098    }
1099    if(p->logstack && (p->flags & POOL_LOGGING)) p->logstack(p);
1100    LOG(p, "poolrealloc %p %p %ld = %p\n", p, v, n, nv);
1101    p->unlock(p);
1102    return nv;
1103  }
1104
1105  void
1106  poolfree(Pool *p, void *v)
1107  {
1108    p->lock(p);
1109    paranoia {
1110            poolcheckl(p);
1111    }
1112    verbosity {
1113            pooldumpl(p);
1114    }
1115    poolfreel(p, v);
1116    paranoia {
1117            poolcheckl(p);
1118    }
1119    verbosity {
1120            pooldumpl(p);
1121    }
1122    if(p->logstack && (p->flags & POOL_LOGGING)) p->logstack(p);
1123    LOG(p, "poolfree %p %p\n", p, v);
1124    p->unlock(p);
1125  }
1126
1127  /*
1128   * Return the real size of a block, and let the user use it.
1129   */
1130  ulong
1131  poolmsize(Pool *p, void *v)
1132  {
1133    Alloc *b;
1134    ulong dsize;
1135
1136    p->lock(p);
1137    paranoia {
1138            poolcheckl(p);
1139    }
1140    verbosity {
1141            pooldumpl(p);
1142    }
1143    if(v == nil)    /* consistency with other braindead ANSI-ness */
1144            dsize = 0;
1145    else {
1146            b = D2B(p, v);
1147            dsize = (b->size&~(p->quantum-1)) - sizeof(Bhdr) - sizeof(Btail);
1148            assert(dsize >= getdsize(b));
1149            blocksetdsize(p, b, dsize);
1150    }
1151    paranoia {
1152            poolcheckl(p);
1153    }
1154    verbosity {
1155            pooldumpl(p);
1156    }
1157    if(p->logstack && (p->flags & POOL_LOGGING)) p->logstack(p);
1158    LOG(p, "poolmsize %p %p = %ld\n", p, v, dsize);
1159    p->unlock(p);
1160    return dsize;
1161  }
1162
1163  /*
1164   * Debugging
1165   */
1166
1167  static void
1168  poolcheckarena(Pool *p, Arena *a)
1169  {
1170    Bhdr *b;
1171    Bhdr *atail;
1172
1173    atail = A2TB(a);
1174    for(b=a; b->magic != ARENATAIL_MAGIC && b<atail; b=B2NB(b))
1175            blockcheck(p, b);
1176    blockcheck(p, b);
1177    if(b != atail)
1178            p->panic(p, "found wrong tail");
1179  }
1180
1181  static void
1182  poolcheckl(Pool *p)
1183  {
1184    Arena *a;
1185
1186    for(a=p->arenalist; a; a=a->down)
1187            poolcheckarena(p, a);
1188  }
1189
1190  void
1191  poolcheck(Pool *p)
1192  {
1193    p->lock(p);
1194    poolcheckl(p);
1195    p->unlock(p);
1196  }
1197
1198  void
1199  poolblockcheck(Pool *p, void *v)
1200  {
1201    if(v == nil)
1202            return;
1203
1204    p->lock(p);
1205    blockcheck(p, D2B(p, v));
1206    p->unlock(p);
1207  }
1208
1209  static void
1210  pooldumpl(Pool *p)
1211  {
1212    Arena *a;
1213
1214    p->print(p, "pool %p %s\n", p, p->name);
1215    for(a=p->arenalist; a; a=a->down)
1216            pooldumparena(p, a);
1217  }
1218
1219  void
1220  pooldump(Pool *p)
1221  {
1222    p->lock(p);
1223    pooldumpl(p);
1224    p->unlock(p);
```

```
}

static void
pooldumparena(Pool *p, Arena *a)
{
  Bhdr *b;

  for(b=a; b->magic != ARENATAIL_MAGIC; b=B2NB(b))
          p->print(p, "(%p %.8lux %lud)", b, b->magic, b->size);
  p->print(p, "\n");
}

/*
 * mark the memory in such a way that we know who marked it
 * (via the signature) and we know where the marking started.
 */
static void
memmark(void *v, int sig, ulong size)
{
  uchar *p, *ep;
  ulong *lp, *elp;
  lp = v;
  elp = lp+size/4;
  while(lp < elp)
          *lp++ = (sig<<24) ^ (long)v;
  p = (uchar*)lp;
  ep = (uchar*)v+size;
  while(p<ep)
          *p++ = sig;
}

```