Multiprocessor Kernel Debugging Using Acid

Steven Stallion
stallion@coraid.com

Introduction

This document describes a method of debugging multiprocessor kernels using *acid*(1). While in-situ debugging is yet to be fully realized, the mechanism detailed herein provides superior postmortem visibility over previous approaches. It is expected that this body of work will result in a debugging environment supportive of both in-situ and postmortem analysis in the future. This approach encourages use of existing tools and culminates in a new *acid* library, which provides a foundation for discovery and analysis of multiprocessor related defects.

The reader should not consider this document an introductory text but merely an exposition of completed work [1].

Mechanism

A panicking kernel will issue a nonmaskable inter-processor interrupt (IPI) to all processors, excluding the caller. Upon receiving the trap, each `Mach` records a pointer to the current `Ureg` structure and the stack pointer to the `dbgreg` and `dbgsp` members, respectively. After which, the processor enters a halt state with interrupts disabled. The debugging processor will maintain a value of zero for both members as a hint to use current register state at debug time. Use of a nonmaskable interrupt yields the ability to record state regardless of processor context; interrupt processing no longer interferes with stack generation.

If `consdebug` is non-nil, `panic` will branch to a debug routine, typically `rdb`. The system is then prepared to accept remote debugging commands, ostensibly via *rdbfs*(4). To simplify setting `consdebug` at boot, a new configuration variable, named `rdb` was introduced to obviate the need for the `^T^Td` control sequence (see *cons*(3)).

To further simplify remote debugging, `rdb` was modified to return if a serial break is received. *Rdbfs* was also updated to send a serial break upon receiving a kill message, allowing *acid* to terminate the debugging session and transitively reboot the system using the `kill` builtin function.

Using the Library Functions

The `mach` library provides a number of functions the user may employ to examine multiprocessor state. `Mach` was developed as a companion to the `kernel` library. As such, both libraries must be defined on the command line when starting *acid*. Normal `kernel` initialization rules apply; `kinit` must be called to establish the proper mapping for the kernel prior to calling functions defined in `mach`.

The following example attaches to a remote kernel with *rdbfs* and gathers basic information using the `mach` library:

```
% rdbfs /mnt/consoles/sys
attach /mnt/consoles/sys
% acid -k -l kernel -l mach -r 9pc
9pc:386 plan 9 boot image
/sys/lib/acid/port
/sys/lib/acid/386
/sys/lib/acid/kernel
/sys/lib/acid/mach
acid: kinit()
        rc("cd /sys/src/9/pc; mk proc.acid")
        include("/sys/src/9/pc/proc.acid")
acid: rc("cd /sys/src/9/pc; mk proc.acid")
8c -FTVw -a -I.  ../port/proc.c >proc.acid
acid: include("/sys/src/9/pc/proc.acid")
acid: machno()
0
acid: machs()
0x80017000 0 up 0x00000000
0x8003d000 1 up 0x00000000
```

Library Functions

The `mach` library is located in the directory `/sys/lib/acid`. As with other libraries, these functions may be overridden, personalized, or added to by code defined in `$home/lib/acid`. The implementation of these functions can be examined using the `whatis` operator and then modified during debugging sessions.

`{}` `mach(`*Mach*`)` — Print summary for Mach

`mach` prints a one line summary for the given *Mach*. The first printed column is the address of *Mach*, followed by the `machno`, and ends with a summary of the currently scheduled process (see `proc`).

```
acid: mach(machp[1])
0x8003d000 1 up 0x00000000
```

`{}` `machgpr(`*Mach*`)` — Print general purpose registers for Mach

`machgpr` prints the general purpose registers for the given *Mach*. While `machgpr` may be used interactively, this function is typically only called by `machregs`.

```
acid: machgpr(machp[1])
AX 0xc9121c18 BX 0x8025f34c CX 0x000000d8 DX 0x00000000
DI 0x80279b44 SI 0x0005d203 BP 0x80016000
```

`{}` `machlstk(`*Mach*`)` — Stack trace with local variables for Mach

`machlstk` produces a long format stack trace for the given *Mach*, similar to `lstk`. Unlike `lstk`, the `:` operator should not be used to address variables on processors other than the debugging `Mach` (see `machno`). This is a limitation imposed by the current implementation of *acid* and *libmach*.

```
acid: machlstk(machp[1])
runproc()+0x53 proc.c:531
   start=0xc9121c18
   p=0x80279948
   rq=0x8025f34c
   i=0x5d203
sched()+0x165 proc.c:164
   p=0x80279948
schedinit()+0x90 proc.c:107
squidboy(apic=0x8026b144)+0x96 mp.c:421
0x80003091 ?file?:0
```

{} machno() — Display debugging Mach number

machno prints the number of the debugging Mach.

```
acid: machno()
0
```

{} machregs(*Mach*) — Print registers for Mach

machregs prints the contents of both the general and special purpose registers for the given *Mach*. machregs calls machspr then machgpr to display the contents of the registers.

```
acid: machregs(machp[1])
PC 0x80196e10 runproc+0x53 proc.c:531
SP 0x80016f80 ECODE 0x801006f8 EFLAG 0x00000202
CS 0x00000010 DS 0x80010008 SS 0x0003b000
GS 0x0000001b FS 0x0000001b ES 0x00000008
TRAP 0x00000002 nonmaskable interrupt
AX 0xc9121c18 BX 0x8025f34c CX 0x000000d8 DX 0x00000000
DI 0x80279b44 SI 0x0005d203 BP 0x80016000
```

{} machs() — Print summaries for all Machs

machs prints summaries for all Machs in the system.

```
acid: machs()
0x80017000 0 up 0x00000000
0x8003d000 1 up 0x00000000
```

{} machspr(*Mach*) — Print special purpose registers for Mach

machspr prints the special purpose registers for the given *Mach*. While machspr may be used interactively, this function is typically only called by machregs.

```
acid: machspr(machp[1])
PC 0x80196e10 runproc+0x53 proc.c:531
SP 0x80016f80 ECODE 0x801006f8 EFLAG 0x00000202
CS 0x00000010 DS 0x80010008 SS 0x0003b000
GS 0x0000001b FS 0x0000001b ES 0x00000008
TRAP 0x00000002 nonmaskable interrupt
```

{} machstacks() — Stack traces for all Machs

machstacks prints a stack trace for all Machs in the system, similar to stacks.

```
acid: machstacks()
=====================================================
0x80017000 0 up 0x00000000
runproc()+0x14d proc.c:530
sched()+0x165 proc.c:164
schedinit()+0x90 proc.c:107
main()+0x158 main.c:130
idle l.s:233
=====================================================
0x8003d000 1 up 0x00000000
runproc()+0x53 proc.c:531
sched()+0x165 proc.c:164
schedinit()+0x90 proc.c:107
squidboy(apic=0x8026b144)+0x96 mp.c:421
0x80003091 ?file?:0
```

`{}` `machstk(`*Mach*`)` Stack trace for Mach

`machstk` produces a short format stack trace for the given *Mach*, similar to `stk`. Unlike `stk`, the `:` operator should not be used to address variables on processors other than the current Mach (see `machno`). This is a limitation imposed by the current implementation of *acid* and *libmach*.

```
acid: machstk(machp[1])
runproc()+0x53 proc.c:531
sched()+0x165 proc.c:164
schedinit()+0x90 proc.c:107
squidboy(apic=0x8026b144)+0x96 mp.c:421
0x80003091 ?file?:0
```

`{}` `machunwind(`*Mach*`)` Dump stack contents for Mach

`machunwind` dumps the contents of the stack for the given *Mach*. This is a function of last resort; it is primarily used to debug the above functions.

```
acid: machunwind(machp[1])
...
0x8003dfd8: schedinit+0x90
0x8003dfdc: 0x80279948
0x8003dfe0: microdelay+0x3c
0x8003dfe4: 0x32e8
0x8003dfe8: 0x0
0x8003dfec: squidboy+0x96
0x8003dff0: 0x64
0x8003dff4: 0x0
0x8003dff8: 0x80003091
0x8003dffc: mplapic+0xb0
```

Support Functions

These functions provide utility to other library functions.

integer `machaddr(`*Mach, integer*`)` Convert address for Mach

`machaddr` converts the given *integer* address to a global address. If the address is mapped to processor-local memory, `machaddr` will provide an alternative that can be addressed by any processor. This function is idempotent; any address, regardless of mapping, may be passed to this function.

```
acid: MACHADDR = KZERO+0x16000;
acid: print(machaddr(machp[1], MACHADDR)\X)
0x8003d000
```

integer `machpc(`*Mach*`)` Find program counter for Mach

`machpc` provides the program counter for the given *Mach*.

```
acid: print(machpc(machp[1])\X)
0x80196e10
```

integer `machsp(`*Mach*`)` Find stack pointer for Mach

`machsp` provides the stack pointer for the given *Mach*.

```
acid: print(machsp(machp[1])\X)
0x80016f80
```

Ureg `machureg(`*Mach*`)` Find Ureg structure for Mach

`machureg` provides the *Ureg* structure for the given *Mach*.

```
acid: print(machureg(machp[1])\X)
0x8003df3c
```

Redefined Functions

A handful of functions defined in other modules are redefined to augment behavior on multiprocessors. This necessitates the `mach` library be defined after augmented libraries on the command line.

`{}` `proclstk(`*Proc*`)` Stack trace with local variables for Proc

`proclstk` produces a long format stack trace for the given *Proc*, similar to `lstk`. Unlike `lstk`, the `:` operator should not be used to address variables unless the process is scheduled on the current Mach (see `machno`). This is a limitation imposed by the current implementation of *acid* and *libmach*. This function was added to supplement the redefined `procstk` function below.

```
acid: proclstk(0x8027ca08)
gotolabel(label=0x80016030)+0x0 l.s:1000
sched()+0x160 proc.c:164
   p=0x286
sysrendezvous(arg=0x8027cc64)+0x143 sysproc.c:836
   rendval=0x0
   tag=0x624bc
   l=0x8b1ce87c
   val=0x7c86d798
   p=0x8027d3c8
syscall(ureg=0x8b20a1a4)+0x238 trap.c:726
   sp=0xcfffee9c
   scallnr=0x22
   startns=0x0
   ret=0xffffffff
   i=0x1
   stopns=0x0
   s=0x0
_syscallintr()+0x18 plan9l.s:44
0x8b20a1a4 ?file?:0
```

{} `procstk(`*Proc*`)` Stack trace for Proc

`procstk` produces a short format stack trace for the given *Proc*, similar to `stk`. Unlike `stk`, the `:` operator should not be used to address variables unless the process is scheduled on the current Mach (see `machno`). This is a limitation imposed by the current implementation of *acid* and *libmach*. This function overrides the definition in `kernel`. This was necessary as `kernel` assumes any given process terminates in a call to `gotolabel`. This is not always the case on a multiprocessor where the process may be actively scheduled at debug time.

```
acid: procstk(0x8027ca08)
gotolabel(label=0x80016030)+0x0 l.s:1000
sched()+0x160 proc.c:164
sysrendezvous(arg=0x8027cc64)+0x143 sysproc.c:836
syscall(ureg=0x8b20a1a4)+0x238 trap.c:726
_syscallintr()+0x18 plan9l.s:44
0x8b20a1a4 ?file?:0
```

string `reason(`*integer*`)` Return cause of Mach stoppage

`reason` uses machine-dependent information to generate a string explaining why a Mach has stopped. The *integer* argument is the value of an architecture dependent status register. This function overrides the builtin definition. This was necessary as the builtin function would discard the *integer* argument and always consult the status register on the debugging Mach.

```
acid: print(reason(machureg(machp[1]).trap))
nonmaskable interrupt
```

Future Work

Only `pc` kernels are supported.

In-situ debugging is not supported.

Floating point is not supported. This is further complicated by support for XSAVE/XRESTOR and YMM register state in `pc` kernels.

Acid and its constituent libraries assume uniprocessor, which causes complications in kernel context. Use of redefined functions have largely addressed these issues.

Users must be aware of processor-local address ranges. A good understanding of kernel memory mapping is essential. The addition of per-processor address translation to *acid* and *libmach* could ease this burden considerably.

The somewhat portable nature of the `kernel` library is at odds with the `pc`-specific `mach` library. Once remaining portability issues are resolved, both libraries could be merged.

Acknowledgements

Portions of this document use descriptions and formatting from the ''Acid Manual'', by Phil Winterbottom.

References

[1] P. Winterbottom, ''Acid: A Debugger Built from A Language'', *USENIX Proc. of the Winter 1994 Conf.*, San Francisco, CA.