Plan 9™

Programmer s Manual
Volume 2

Fourth Edition
2000

Computing Science Research Center
Bell Laboratories
Lucent Technologies
Murray Hill, New Jersey

Cover Design: Gerard J. Holzmann

Trademarks referenced in this document:

Plan 9 is a trademark of Lucent Technologies Inc.
Aladdin Ghostscript is a trademark of Aladdin Enterprises.
ARM is a trademark of ARM Limited.
Avanstar is a registered trademark of Star Gate Technologies, Inc.
CGA and VGA are trademarks of International Business Machines Corporation.
Silicon Graphics, IRIS Indigo, IRIS, IRIX, Challenge, and Indigo
are registered trademarks of Silicon Graphics, Inc.
Indy and POWER Series are trademarks of Silicon Graphics, Inc.
Ethernet is a trademark of Xerox Corporation.
IBM, PS/2, and PowerPC are registered trademarks of
International Business Machines Corporation.
Intel and Pentium are registered trademarks of Intel Corporation.
8088, 80286, 80386, and 80486 are trademarks of Intel Corporation.
Lucida is a registered trademark of Bigelow & Holmes.
Pellucida is a trademark of Bigelow & Holmes.
MIPS, R3000, R4000, and R4400 are registered trademarks of MIPS Technologies, Inc.
R2000 and R6000 are trademarks of MIPS Technologies, Inc.
Microsoft, Microsoft Word and Microsoft Office, and MS-DOS are
registered trademarks of Microsoft Corporation.
NFS is a registered trademark of Sun Microsystems, Inc.
PDP and VAX are registered trademarks of Digital Equipment Corp.
PostScript is a registered trademark of Adobe Systems Incorporated.
R2000, R6000, R4000, and R4400 are trademarks of MIPS Technologies, Inc.
SecureNet is a trademark of Digital Pathways, Inc.
Sound Blaster is a registered trademark of Creative Labs, Inc.
SPARC is a registered trademark of SPARC International, Inc.
Unicode is a registered certification mark of Unicode, Inc.
UNIX is a registered trademark in the USA and other countries licensed
exclusively through X/Open Company Limited.

Table of Contents

Introduction

Plan 9 From Bell Labs 1

Rob Pike, Dave Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, and Phil Winterbottom

An overview of the system; read at least this paper before you install.

The Use of Name Spaces in Plan 9 23

Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, and Phil Winterbottom

What s in a name?

The Organization of Networks in Plan 9 29

Dave Presotto and Phil Winterbottom

Connecting the pieces. The details in the paper are outdated but the ideas still apply.

Security in Plan 9 43

Russ Cox, Eric Grosse, Rob Pike, Dave Presotto, and Sean Quinlan

An overview of the security architecture.

Programming

How to Use the Plan 9 C Compiler 63

Rob Pike

The starting point for C programming under Plan 9.

Changes to the Programming Environment in the Fourth Release of Plan 9 77

Rob Pike

An overview of the changes to the C library since the Third Release.

APE   The ANSI/POSIX Environment 83

Howard Trickey

Moving C code between UNIX and Plan 9.

Acid: A Debugger Built From A Language 87

Phil Winterbottom

An overview paper about the Acid debugger.

Acid Manual 101

Phil Winterbottom

The reference manual for the language and its libraries.

Maintaining Files on Plan 9 with Mk 127

Andrew G. Hume and Bob Flandrena

An introduction to Plan 9 s replacement for make .

Plan 9 Mkfiles 141

Bob Flandrena

The conventions for using mk in Plan 9.

A Manual for the Plan 9 assembler 149

Rob Pike

Things you d rather not know.

User Interfaces

8½, the Plan 9 Window System 159
Rob Pike
An introduction to the (previous) window system and its unusual implementation.

Rc — The Plan 9 Shell 169
Tom Duff
An introduction to the new shell, complete with examples.

The Text Editor sam 185
Rob Pike
Sam is the standard editor on Plan 9.

Acme: A User Interface for Programmers 215
Rob Pike
A system with a more radical approach to programming and editing.

Plumbing and Other Utilities 231
Rob Pike
Inter-process communication that enlivens the interactive user interface.

Implementation

Hello World, or ˚±»·¼–Á± °ÌÃ¼µ, or こんにちは世界 247
Rob Pike and Ken Thompson
The details about Plan 9 s character set: the Unicode Standard plus an ASCII-compatible encoding.

Plan 9 C Compilers 259
Ken Thompson
The design and some internals of the compiler suite.

Adding Application Support for a New Architecture in Plan 9 271
Bob Flandrena
The procedures necessary to add a new instruction set to Plan 9 s programming environment.

The Plan 9 File Server 279
Ken Thompson
The design of the central file server and its novel backup system.

Venti: A new approach to archival storage 287
Sean Quinlan and Sean Dorward
Archival block-level storage using secure hashes as block identifiers.

The IL protocol 301
Dave Presotto and Phil Winterbottom
A description of the Internet protocol Plan 9 uses for internal communication.

Lexical File Names in Plan 9, or, Getting Dot-Dot Right 307
Rob Pike
A vexing old problem solved: how to make pwd get the right answer in the face of multiply-bound directories.

Process Sleep and Wakeup on a Shared-memory Multiprocessor 319
Rob Pike, Dave Presotto, Ken Thompson, and Gerard Holzmann
A detailed study of a central issue in the Plan 9 kernel.

Miscellany

A Guide to the Lp Printer Spooler 325
Paul Glick
Adminstering the suite of tools to drive PostScript printers.

Troff Users Manual 333

Joseph F. Ossanna and Brian W. Kernighan
The old warhorse, updated for Unicode characters.

Using SPIN 373
Gerard Holzmann
An introduction to a tool for analyzing parallel and distributed programs.

Installation

The Various Ports 400
The hardware requirements for the Plan 9 compilers and kernels.

Plan 9 from Bell Labs

Rob Pike
Dave Presotto
Sean Dorward
Bob Flandrena
Ken Thompson
Howard Trickey
Phil Winterbottom

Bell Laboratories
Murray Hill, New Jersey 07974
USA

Motivation

By the mid 1980's, the trend in computing was away from large centralized time-shared computers towards networks of smaller, personal machines, typically UNIX 'workstations'. People had grown weary of overloaded, bureaucratic timesharing machines and were eager to move to small, self-maintained systems, even if that meant a net loss in computing power. As microcomputers became faster, even that loss was recovered, and this style of computing remains popular today.

In the rush to personal workstations, though, some of their weaknesses were overlooked. First, the operating system they run, UNIX, is itself an old timesharing system and has had trouble adapting to ideas born after it. Graphics and networking were added to UNIX well into its lifetime and remain poorly integrated and difficult to administer. More important, the early focus on having private machines made it difficult for networks of machines to serve as seamlessly as the old monolithic timesharing systems. Timesharing centralized the management and amortization of costs and resources; personal computing fractured, democratized, and ultimately amplified administrative problems. The choice of an old timesharing operating system to run those personal machines made it difficult to bind things together smoothly.

Plan 9 began in the late 1980's as an attempt to have it both ways: to build a system that was centrally administered and cost-effective using cheap modern microcomputers as its computing elements. The idea was to build a time-sharing system out of workstations, but in a novel way. Different computers would handle different tasks: small, cheap machines in people's offices would serve as terminals providing access to large, central, shared resources such as computing servers and file servers. For the central machines, the coming wave of shared-memory multiprocessors seemed obvious candidates. The philosophy is much like that of the Cambridge Distributed System [NeHe82]. The early catch phrase was to build a UNIX out of a lot of little systems, not a system out of a lot of little UNIXes.

Appeared in a slightly different form in *Computing Systems*, Vol 8 #3, Summer 1995, pp. 221-254.

The problems with UNIX were too deep to fix, but some of its ideas could be brought along. The best was its use of the file system to coordinate naming of and access to resources, even those, such as devices, not traditionally treated as files. For Plan 9, we adopted this idea by designing a network-level protocol, called 9P, to enable machines to access files on remote systems. Above this, we built a naming system that lets people and their computing agents build customized views of the resources in the network. This is where Plan 9 first began to look different: a Plan 9 user builds a private computing environment and recreates it wherever desired, rather than doing all computing on a private machine. It soon became clear that this model was richer than we had foreseen, and the ideas of per-process name spaces and file-system-like resources were extended throughout the system—to processes, graphics, even the network itself.

By 1989 the system had become solid enough that some of us began using it as our exclusive computing environment. This meant bringing along many of the services and applications we had used on UNIX. We used this opportunity to revisit many issues, not just kernel-resident ones, that we felt UNIX addressed badly. Plan 9 has new compilers, languages, libraries, window systems, and many new applications. Many of the old tools were dropped, while those brought along have been polished or rewritten.

Why be so all-encompassing? The distinction between operating system, library, and application is important to the operating system researcher but uninteresting to the user. What matters is clean functionality. By building a complete new system, we were able to solve problems where we thought they should be solved. For example, there is no real tty driver in the kernel; that is the job of the window system. In the modern world, multi-vendor and multi-architecture computing are essential, yet the usual compilers and tools assume the program is being built to run locally; we needed to rethink these issues. Most important, though, the test of a system is the computing environment it provides. Producing a more efficient way to run the old UNIX warhorses is empty engineering; we were more interested in whether the new ideas suggested by the architecture of the underlying system encourage a more effective way of working. Thus, although Plan 9 provides an emulation environment for running POSIX commands, it is a backwater of the system. The vast majority of system software is developed in the native Plan 9 environment.

There are benefits to having an all-new system. First, our laboratory has a history of building experimental peripheral boards. To make it easy to write device drivers, we want a system that is available in source form (no longer guaranteed with UNIX, even in the laboratory in which it was born). Also, we want to redistribute our work, which means the software must be locally produced. For example, we could have used some vendors C compilers for our system, but even had we overcome the problems with cross-compilation, we would have difficulty redistributing the result.

This paper serves as an overview of the system. It discusses the architecture from the lowest building blocks to the computing environment seen by users. It also serves as an introduction to the rest of the Plan 9 Programmer s Manual, which it accompanies. More detail about topics in this paper can be found elsewhere in the manual.

Design

The view of the system is built upon three principles. First, resources are named and accessed like files in a hierarchical file system. Second, there is a standard protocol, called 9P, for accessing these resources. Third, the disjoint hierarchies provided by different services are joined together into a single private hierarchical file name space. The unusual properties of Plan 9 stem from the consistent, aggressive application of these principles.

A large Plan 9 installation has a number of computers networked together, each providing a particular class of service. Shared multiprocessor servers provide computing cycles; other large machines offer file storage. These machines are located in an air-conditioned machine room and are connected by high-performance networks. Lower bandwidth networks such as Ethernet or ISDN connect these servers to office- and home-resident workstations or PCs, called terminals in Plan 9 terminology. Figure 1 shows the arrangement.

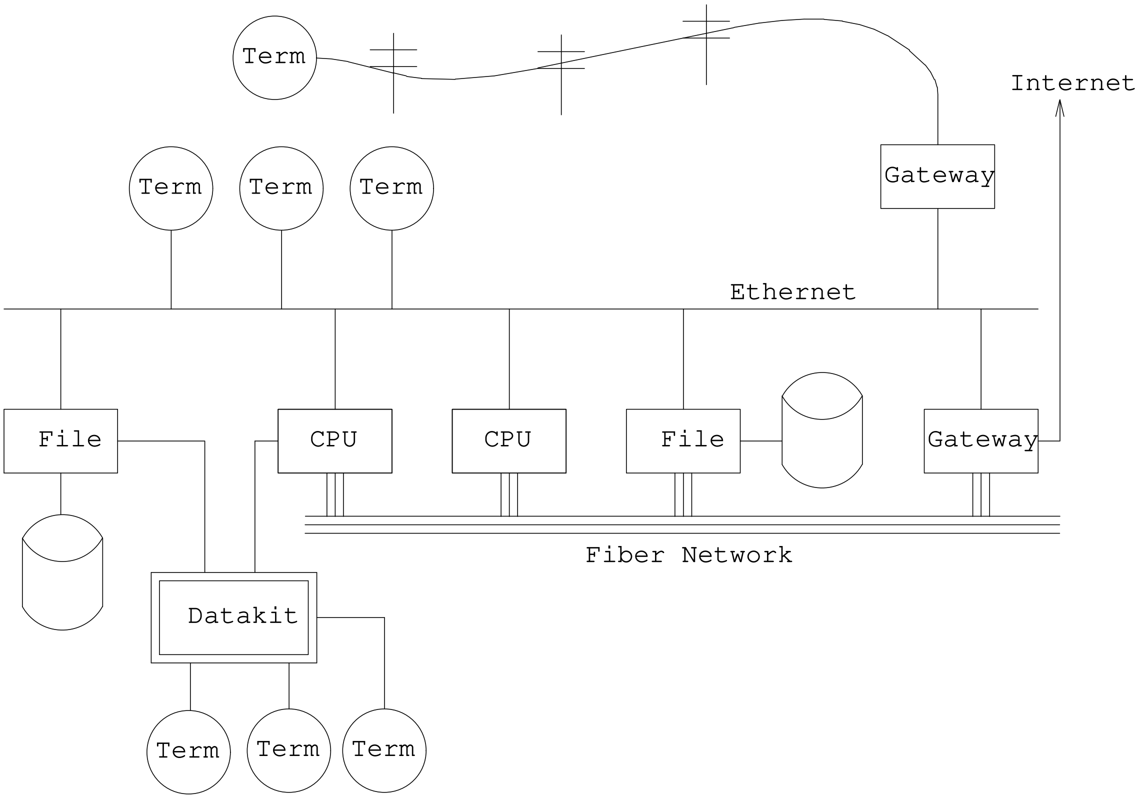

Figure 1. Structure of a large Plan 9 installation. CPU servers and file servers share fast local-area networks, while terminals use slower wider-area networks such as Ethernet, Datakit, or telephone lines to connect to them. Gateway machines, which are just CPU servers connected to multiple networks, allow machines on one network to see another.

The modern style of computing offers each user a dedicated workstation or PC. Plan 9 s approach is different. The various machines with screens, keyboards, and mice all provide access to the resources of the network, so they are functionally equivalent, in the manner of the terminals attached to old timesharing systems. When someone uses the system, though, the terminal is temporarily personalized by that user. Instead of customizing the hardware, Plan 9 offers the ability to customize one s view of the system provided by the software. That customization is accomplished by giving local, personal names for the publicly visible resources in the network. Plan 9 provides the mechanism to assemble a personal view of the public space with local names for globally accessible resources. Since the most important resources of the network are files, the model of that view is file-oriented.

The client s local name space provides a way to customize the user s view of the network. The services available in the network all export file hierarchies. Those important to the user are gathered together into a custom name space; those of no immediate interest are ignored. This is a different style of use from the idea of a uniform global name space. In Plan 9, there are known names for services and uniform names for files exported by those services, but the view is entirely local. As an analogy, consider the difference between the phrase my house and the precise address of the speaker s home. The latter may be used by anyone but the former is easier to say and makes sense when spoken. It also changes meaning depending on who says it, yet that does not cause confusion. Similarly, in Plan 9 the name /dev/cons always refers to the user's terminal and /bin/date the correct version of the date command to run, but which files those names represent depends on circumstances such as the architecture of the machine executing date. Plan 9, then, has local name spaces that obey globally understood conventions; it is the conventions that guarantee sane behavior in the presence of local names.

The 9P protocol is structured as a set of transactions that send a request from a client to a (local or remote) server and return the result. 9P controls file systems, not just files: it includes procedures to resolve file names and traverse the name hierarchy of the file system provided by the server. On the other hand, the client's name space is held by the client system alone, not on or with the server, a distinction from systems such as Sprite [OCDNW88]. Also, file access is at the level of bytes, not blocks, which distinguishes 9P from protocols like NFS and RFS. A paper by Welch compares Sprite, NFS, and Plan 9's network file system structures [Welc94].

This approach was designed with traditional files in mind, but can be extended to many other resources. Plan 9 services that export file hierarchies include I/O devices, backup services, the window system, network interfaces, and many others. One example is the process file system, /proc, which provides a clean way to examine and control running processes. Precursor systems had a similar idea [Kill84], but Plan 9 pushes the file metaphor much further [PPTTW93]. The file system model is well-understood, both by system builders and general users, so services that present file-like interfaces are easy to build, easy to understand, and easy to use. Files come with agreed-upon rules for protection, naming, and access both local and remote, so services built this way are ready-made for a distributed system. (This is a distinction from object-oriented models, where these issues must be faced anew for every class of object.) Examples in the sections that follow illustrate these ideas in action.

The Command-level View

Plan 9 is meant to be used from a machine with a screen running the window system. It has no notion of teletype in the UNIX sense. The keyboard handling of the bare system is rudimentary, but once the window system, 8½ [Pike91], is running, text can be edited with cut and paste operations from a pop-up menu, copied between windows, and so on. 8½ permits editing text from the past, not just on the current input line. The text-editing capabilities of 8½ are strong enough to displace special features such as history in the shell, paging and scrolling, and mail editors. 8½ windows do not support cursor addressing and, except for one terminal emulator to simplify connecting to traditional systems, there is no cursor-addressing software in Plan 9.

Each window is created in a separate name space. Adjustments made to the name space in a window do not affect other windows or programs, making it safe to experiment with local modifications to the name space, for example to substitute files from the dump file system when debugging. Once the debugging is done, the window can be deleted and all trace of the experimental apparatus is gone. Similar arguments apply to the private space each window has for environment variables, notes (analogous to UNIX signals), etc.

Each window is created running an application, such as the shell, with standard input and output connected to the editable text of the window. Each window also has a private bitmap and multiplexed access to the keyboard, mouse, and other graphical resources through files like /dev/mouse, /dev/bitblt, and /dev/cons (analogous to UNIX's /dev/tty). These files are provided by 8½, which is implemented as a file server. Unlike X windows, where a new application typically creates a new window to run in, an 8½ graphics application usually runs in the window where it starts. It is possible and efficient for an application to create a new window, but that is not the style of the system. Again contrasting to X, in which a remote application makes a network call to the X server to start running, a remote 8½ application sees the mouse , bitblt , and cons files for the window as usual in /dev ; it does not know whether the files are local. It just reads and writes them to control the window; the network connection is already there and multiplexed.

The intended style of use is to run interactive applications such as the window system and text editor on the terminal and to run computation- or file-intensive applications on remote servers. Different windows may be running programs on different machines over different networks, but by making the name space equivalent in all windows, this is transparent: the same commands and resources are available, with the same names, wherever the computation is performed.

The command set of Plan 9 is similar to that of UNIX. The commands fall into several broad classes. Some are new programs for old jobs: programs like ls , cat , and who have familiar names and functions but are new, simpler implementations. Who , for example, is a shell script, while ps is just 95 lines of C code. Some commands are essentially the same as their UNIX ancestors: awk , troff , and others have been converted to ANSI C and extended to handle Unicode, but are still the familiar tools. Some are entirely new programs for old niches: the shell rc , text editor sam , debugger acid , and others displace the better-known UNIX tools with similar jobs. Finally, about half the commands are new.

Compatibility was not a requirement for the system. Where the old commands or notation seemed good enough, we kept them. When they didn't, we replaced them.

The File Server

A central file server stores permanent files and presents them to the network as a file hierarchy exported using 9P. The server is a stand-alone system, accessible only over the network, designed to do its one job well. It runs no user processes, only a fixed set of routines compiled into the boot image. Rather than a set of disks or separate file systems, the main hierarchy exported by the server is a single tree, representing files on many disks. That hierarchy is shared by many users over a wide area on a variety of networks. Other file trees exported by the server include special-purpose systems such as temporary storage and, as explained below, a backup service.

The file server has three levels of storage. The central server in our installation has about 100 megabytes of memory buffers, 27 gigabytes of magnetic disks, and 350 gigabytes of bulk storage in a write-once-read-many (WORM) jukebox. The disk is a cache for the WORM and the memory is a cache for the disk; each is much faster, and sees about an order of magnitude more traffic, than the level it caches. The addressable data in the file system can be larger than the size of the magnetic disks, because they are only a cache; our main file server has about 40 gigabytes of active storage.

The most unusual feature of the file server comes from its use of a WORM device for stable storage. Every morning at 5 o'clock, a *dump* of the file system occurs automatically. The file system is frozen and all blocks modified since the last dump are queued to be written to the WORM. Once the blocks are queued, service is restored and the read-only root of the dumped file system appears in a hierarchy of all dumps ever taken, named by its date. For example, the directory /n/dump/1995/0315 is the root directory of an image of the file system as it appeared in the early morning of March 15, 1995. It takes a few minutes to queue the blocks, but the process to copy blocks to the WORM, which runs in the background, may take hours.

There are two ways the dump file system is used. The first is by the users themselves, who can browse the dump file system directly or attach pieces of it to their name space. For example, to track down a bug, it is straightforward to try the compiler from three months ago or to link a program with yesterday's library. With daily snapshots of all files, it is easy to find when a particular change was made or what changes were made on a particular date. People feel free to make large speculative changes to files in the knowledge that they can be backed out with a single copy command. There is no backup system as such; instead, because the dump is in the file name space, backup problems can be solved with standard tools such as cp, ls, grep, and diff.

The other (very rare) use is complete system backup. In the event of disaster, the active file system can be initialized from any dump by clearing the disk cache and setting the root of the active file system to be a copy of the dumped root. Although easy to do, this is not to be taken lightly: besides losing any change made after the date of the dump, this recovery method results in a very slow system. The cache must be reloaded from WORM, which is much slower than magnetic disks. The file system takes a few days to reload the working set and regain its full performance.

Access permissions of files in the dump are the same as they were when the dump was made. Normal utilities have normal permissions in the dump without any special arrangement. The dump file system is read-only, though, which means that files in the dump cannot be written regardless of their permission bits; in fact, since directories are part of the read-only structure, even the permissions cannot be changed.

Once a file is written to WORM, it cannot be removed, so our users never see please clean up your files messages and there is no df command. We regard the WORM jukebox as an unlimited resource. The only issue is how long it will take to fill. Our WORM has served a community of about 50 users for five years and has absorbed daily dumps, consuming a total of 65% of the storage in the jukebox. In that time, the manufacturer has improved the technology, doubling the capacity of the individual disks. If we were to upgrade to the new media, we would have more free space than in the original empty jukebox. Technology has created storage faster than we can use it.

Unusual file servers

Plan 9 is characterized by a variety of servers that offer a file-like interface to unusual services. Many of these are implemented by user-level processes, although the distinction is unimportant to their clients; whether a service is provided by the kernel, a user process, or a remote server is irrelevant to the way it is used. There are dozens of such servers; in this section we present three representative ones.

Perhaps the most remarkable file server in Plan 9 is 8½, the window system. It is discussed at length elsewhere [Pike91], but deserves a brief explanation here. 8½ provides two interfaces: to the user seated at the terminal, it offers a traditional style of interaction with multiple windows, each running an application, all controlled by a mouse and keyboard. To the client programs, the view is also fairly traditional: programs running in a window see a set of files in /dev with names like mouse, screen, and cons. Programs that want to print text to their window write to /dev/cons; to read the mouse, they read /dev/mouse. In the Plan 9 style, bitmap graphics is implemented by providing a file /dev/bitblt on which clients write encoded messages to execute graphical operations such as bitblt (RasterOp). What is unusual is how this is done: 8½ is a file server, serving the files in /dev to the clients running in each window. Although every window looks the same to its client, each window has a distinct set of files in /dev. 8½ multiplexes its clients' access to the resources of the terminal by serving multiple sets of files. Each client is given a private name space with a *different* set of files that behave the same as in all other windows. There are many advantages to this structure. One is that 8½ serves the same files it needs for its own implementation—it multiplexes its own interface—so it may be run, recursively, as a client of itself. Also, consider the implementation of /dev/tty in UNIX, which requires special code in the kernel to redirect open calls to the appropriate device. Instead, in 8½ the equivalent service falls out automatically: 8½ serves /dev/cons as its basic function; there is nothing extra to do. When a program wants to read from the keyboard, it opens /dev/cons, but it is a private file, not a shared one with special properties. Again, local name spaces make this possible; conventions about the consistency of the files within them make it natural.

8½ has a unique feature made possible by its design. Because it is implemented as a file server, it has the power to postpone answering read requests for a particular window. This behavior is toggled by a reserved key on the keyboard. Toggling once suspends client reads from the window; toggling again resumes normal reads, which absorb whatever text has been prepared, one line at a time. This allows the user to edit multi-line input text on the screen before the application sees it, obviating the need to invoke a separate editor to prepare text such as mail messages. A related property is that reads are answered directly from the data structure defining the text on the display: text may be edited until its final newline makes the prepared line of text readable by the client. Even then, until the line is read, the text the client will read can be changed. For example, after typing

```
% make
rm *
```

to the shell, the user can backspace over the final newline at any time until make finishes, holding off execution of the rm command, or even point with the mouse before the rm and type another command to be executed first.

There is no ftp command in Plan 9. Instead, a user-level file server called ftpfs dials the FTP site, logs in on behalf of the user, and uses the FTP protocol to examine files in the remote directory. To the local user, it offers a file hierarchy, attached to /n/ftp in the local name space, mirroring the contents of the FTP site. In other words, it translates the FTP protocol into 9P to offer Plan 9 access to FTP sites. The implementation is tricky; ftpfs must do some sophisticated caching for efficiency and use heuristics to decode remote directory information. But the result is worthwhile: all the local file management tools such as cp, grep, diff, and of course ls are available to FTP-served files exactly as if they were local files. Other systems such as Jade and Prospero have exploited the same opportunity [Rao81, Neu92], but because of local name spaces and the simplicity of implementing 9P, this approach fits more naturally into Plan 9 than into other environments.

One server, exportfs, is a user process that takes a portion of its own name space and makes it available to other processes by translating 9P requests into system calls to the Plan 9 kernel. The file hierarchy it exports may contain files from multiple servers. Exportfs is usually run as a remote server started by a local program, either import or cpu. Import makes a network call to the remote machine, starts exportfs there, and attaches its 9P connection to the local name space. For example,

```
import helix /net
```

makes Helix's network interfaces visible in the local /net directory. Helix is a central server and has many network interfaces, so this permits a machine with one network to access to any of Helix's networks. After such an import, the local machine may make calls on any of the networks connected to Helix. Another example is

```
import helix /proc
```

which makes Helix's processes visible in the local /proc, permitting local debuggers to examine remote processes.

The cpu command connects the local terminal to a remote CPU server. It works in the opposite direction to import: after calling the server, it starts a *local* exportfs and mounts it in the name space of a process, typically a newly created shell, on the server. It then rearranges the name space to make local device files (such as those served by the terminal's window system) visible in the server's /dev directory. The effect of running a *cpu* command is therefore to start a shell on a fast machine, one more tightly coupled to the file server, with a name space analogous to the local one. All local device files are visible remotely, so remote applications have full access to local services such as bitmap graphics, */dev/cons*, and so on. This is not the same as *rlogin*, which does nothing to reproduce the local name space on the remote system, nor is it the same as file sharing with, say, NFS, which can achieve some name space equivalence but not the combination of access to local hardware devices, remote files, and remote CPU resources. The *cpu* command is a uniquely transparent mechanism. For example, it is reasonable to start a window system in a window running a *cpu* command; all windows created there automatically start processes on the CPU server.

Configurability and administration

The uniform interconnection of components in Plan 9 makes it possible to configure a Plan 9 installation many different ways. A single laptop PC can function as a stand-alone Plan 9 system; at the other extreme, our setup has central multiprocessor CPU servers and file servers and scores of terminals ranging from small PCs to high-end graphics workstations. It is such large installations that best represent how Plan 9 operates.

The system software is portable and the same operating system runs on all hardware. Except for performance, the appearance of the system on, say, an SGI workstation is the same as on a laptop. Since computing and file services are centralized, and terminals have no permanent file storage, all terminals are functionally identical. In this way, Plan 9 has one of the good properties of old timesharing systems, where a user could sit in front of any machine and see the same system. In the modern workstation community, machines tend to be owned by people who customize them by storing private information on local disk. We reject this style of use, although the system itself can be used this way. In our group, we have a laboratory with many public-access machines a terminal room and a user may sit down at any one of them and work.

Central file servers centralize not just the files, but also their administration and maintenance. In fact, one server is the main server, holding all system files; other servers provide extra storage or are available for debugging and other special uses, but the system software resides on one machine. This means that each program has a single copy of the binary for each architecture, so it is trivial to install updates and bug fixes. There is also a single user database; there is no need to synchronize distinct */etc/passwd* files. On the other hand, depending on a single central server does limit the size of an installation.

Another example of the power of centralized file service is the way Plan 9 administers network information. On the central server there is a directory, */lib/ndb*, that contains all the information necessary to administer the local Ethernet and other networks. All the machines use the same database to talk to the network; there is no need to manage a distributed naming system or keep parallel files up to date. To install a new machine on the local Ethernet, choose a name and IP address and add these to a single file in */lib/ndb*; all the machines in the installation will be able to talk to it immediately. To start running, plug the machine into the network, turn it on, and use BOOTP and TFTP to load the kernel. All else is automatic.

Finally, the automated dump file system frees all users from the need to maintain their systems, while providing easy access to backup files without tapes, special commands, or the involvement of support staff. It is difficult to overstate the improvement in lifestyle afforded by this service.

Plan 9 runs on a variety of hardware without constraining how to configure an installation. In our laboratory, we chose to use central servers because they amortize costs and administration. A sign that this is a good decision is that our cheap terminals remain comfortable places to work for about five years, much longer than workstations that must provide the complete computing environment. We do, however, upgrade the central machines, so the computation available from even old Plan 9 terminals improves with time. The money saved by avoiding regular upgrades of terminals is instead spent on the newest, fastest multiprocessor servers. We estimate this costs about half the money of networked workstations yet provides general access to more powerful machines.

C Programming

Plan 9 utilities are written in several languages. Some are scripts for the shell, rc [Duff90]; a handful are written in a new C-like concurrent language called Alef [Wint95], described below. The great majority, though, are written in a dialect of ANSI C [ANSIC]. Of these, most are entirely new programs, but some originate in pre-ANSI C code from our research UNIX system [UNIX85]. These have been updated to ANSI C and reworked for portability and cleanliness.

The Plan 9 C dialect has some minor extensions, described elsewhere [Pike95], and a few major restrictions. The most important restriction is that the compiler demands that all function definitions have ANSI prototypes and all function calls appear in the scope of a prototyped declaration of the function. As a stylistic rule, the prototyped declaration is placed in a header file included by all files that call the function. Each system library has an associated header file, declaring all functions in that library. For example, the standard Plan 9 library is called libc, so all C source files include <libc.h>. These rules guarantee that all functions are called with arguments having the expected types something that was not true with pre-ANSI C programs.

Another restriction is that the C compilers accept only a subset of the preprocessor directives required by ANSI. The main omission is #if, since we believe it is never necessary and often abused. Also, its effect is better achieved by other means. For instance, an #if used to toggle a feature at compile time can be written as a regular if statement, relying on compile-time constant folding and dead code elimination to discard object code.

Conditional compilation, even with #ifdef, is used sparingly in Plan 9. The only architecture-dependent #ifdefs in the system are in low-level routines in the graphics library. Instead, we avoid such dependencies or, when necessary, isolate them in separate source files or libraries. Besides making code hard to read, #ifdefs make it impossible to know what source is compiled into the binary or whether source protected by them will compile or work properly. They make it harder to maintain software.

The standard Plan 9 library overlaps much of ANSI C and POSIX [POSIX], but diverges when appropriate to Plan 9's goals or implementation. When the semantics of a function change, we also change the name. For instance, instead of UNIX's creat, Plan 9 has a create function that takes three arguments, the original two plus a third that, like the second argument of open, defines whether the returned file descriptor is to be opened for reading, writing, or both. This design was forced by the way 9P implements creation, but it also simplifies the common use of create to initialize a temporary file.

Another departure from ANSI C is that Plan 9 uses a 16-bit character set called Unicode [ISO10646, Unicode]. Although we stopped short of full internationalization, Plan 9 treats the representation of all major languages uniformly throughout all its software. To simplify the exchange of text between programs, the characters are packed into a byte stream by an encoding we designed, called UTF-8, which is now becoming accepted as a standard [FSSUTF]. It has several attractive properties, including byte-order independence, backwards compatibility with ASCII, and ease of implementation.

There are many problems in adapting existing software to a large character set with an encoding that represents characters with a variable number of bytes. ANSI C addresses some of the issues but falls short of solving them all. It does not pick a character set encoding and does not define all the necessary I/O library routines. Furthermore, the functions it *does* define have engineering problems. Since the standard left too many problems unsolved, we decided to build our own interface. A separate paper has the details [Pike93].

A small class of Plan 9 programs do not follow the conventions discussed in this section. These are programs imported from and maintained by the UNIX community; tex is a representative example. To avoid reconverting such programs every time a new version is released, we built a porting environment, called the ANSI C/POSIX Environment, or APE [Tric95]. APE comprises separate include files, libraries, and commands, conforming as much as possible to the strict ANSI C and base-level POSIX specifications. To port network-based software such as X Windows, it was necessary to add some extensions to those specifications, such as the BSD networking functions.

Portability and Compilation

Plan 9 is portable across a variety of processor architectures. Within a single computing session, it is common to use several architectures: perhaps the window system running on an Intel processor connected to a MIPS-based CPU server with files resident on a SPARC system. For this heterogeneity to be transparent, there must be conventions about data interchange between programs; for software maintenance to be straightforward, there must be conventions about cross-architecture compilation.

To avoid byte order problems, data is communicated between programs as text whenever practical. Sometimes, though, the amount of data is high enough that a binary format is necessary; such data is communicated as a byte stream with a pre-defined encoding for multi-byte values. In the rare cases where a format is complex enough to be defined by a data structure, the structure is never communicated as a unit; instead, it is decomposed into individual fields, encoded as an ordered byte stream, and then reassembled by the recipient. These conventions affect data ranging from kernel or application program state information to object file intermediates generated by the compiler.

Programs, including the kernel, often present their data through a file system interface, an access mechanism that is inherently portable. For example, the system clock is represented by a decimal number in the file /dev/time; the time library function (there is no time system call) reads the file and converts it to binary. Similarly, instead of encoding the state of an application process in a series of flags and bits in private memory, the kernel presents a text string in the file named status in the /proc file system associated with each process. The Plan 9 ps command is trivial: it prints the contents of the desired status files after some minor reformatting; moreover, after

```
import helix /proc
```

a local ps command reports on the status of Helix processes.

Each supported architecture has its own compilers and loader. The C and Alef compilers produce intermediate files that are portably encoded; the contents are unique to the target architecture but the format of the file is independent of compiling processor type. When a compiler for a given architecture is compiled on another type of processor and then used to compile a program there, the intermediate produced on the new architecture is identical to the intermediate produced on the native processor. From the compiler's point of view, every compilation is a cross-compilation.

Although each architecture's loader accepts only intermediate files produced by compilers for that architecture, such files could have been generated by a compiler executing on any type of processor. For instance, it is possible to run the MIPS compiler on a 486, then use the MIPS loader on a SPARC to produce a MIPS executable.

Since Plan 9 runs on a variety of architectures, even in a single installation, distinguishing the compilers and intermediate names simplifies multi-architecture development from a single source tree. The compilers and the loader for each architecture are uniquely named; there is no cc command. The names are derived by concatenating a code letter associated with the target architecture with the name of the compiler or loader. For example, the letter ‘8’ is the code letter for Intel x86 processors; the C compiler is named 8c, the Alef compiler 8al, and the loader is called 8l. Similarly, the compiler intermediate files are suffixed .8, not .o.

The Plan 9 build program mk, a relative of make, reads the names of the current and target architectures from environment variables called $cputype and $objtype. By default the current processor is the target, but setting $objtype to the name of another architecture before invoking mk results in a cross-build:

```
% objtype=sparc mk
```

builds a program for the SPARC architecture regardless of the executing machine. The value of $objtype selects a file of architecture-dependent variable definitions that configures the build to use the appropriate compilers and loader. Although simple-minded, this technique works well in practice: all applications in Plan 9 are built from a single source tree and it is possible to build the various architectures in parallel without conflict.

Parallel programming

Plan 9’s support for parallel programming has two aspects. First, the kernel provides a simple process model and a few carefully designed system calls for synchronization and sharing. Second, a new parallel programming language called Alef supports concurrent programming. Although it is possible to write parallel programs in C, Alef is the parallel language of choice.

There is a trend in new operating systems to implement two classes of processes: normal UNIX-style processes and light-weight kernel threads. Instead, Plan 9 provides a single class of process but allows fine control of the sharing of a process’s resources such as memory and file descriptors. A single class of process is a feasible approach in Plan 9 because the kernel has an efficient system call interface and cheap process creation and scheduling.

Parallel programs have three basic requirements: management of resources shared between processes, an interface to the scheduler, and fine-grain process synchronization using spin locks. On Plan 9, new processes are created using the rfork system call. Rfork takes a single argument, a bit vector that specifies which of the parent process’s resources should be shared, copied, or created anew in the child. The resources controlled by rfork include the name space, the environment, the file descriptor table, memory segments, and notes (Plan 9’s analog of UNIX signals). One of the bits controls whether the rfork call will create a new process; if the bit is off, the resulting modification to the resources occurs in the process making the call. For example, a process calls rfork(RFNAMEG) to disconnect its name space from its parent’s. Alef uses a fine-grained fork in which all the resources, including memory, are shared between parent and child, analogous to creating a kernel thread in many systems.

An indication that rfork is the right model is the variety of ways it is used. Other than the canonical use in the library routine fork, it is hard to find two calls to rfork with the same bits set; programs use it to create many different forms of sharing and resource allocation. A system with just two types of processes—regular processes and threads—could not handle this variety.

There are two ways to share memory. First, a flag to rfork causes all the memory segments of the parent to be shared with the child (except the stack, which is forked copy-on-write regardless). Alternatively, a new segment of memory may be attached using the segattach system call; such a segment will always be shared between parent and child.

The rendezvous system call provides a way for processes to synchronize. Alef uses it to implement communication channels, queuing locks, multiple reader/writer locks, and the sleep and wakeup mechanism. Rendezvous takes two arguments, a tag and a value. When a process calls rendezvous with a tag it sleeps until another process presents a matching tag. When a pair of tags match, the values are exchanged between the two processes and both rendezvous calls return. This primitive is sufficient to implement the full set of synchronization routines.

Finally, spin locks are provided by an architecture-dependent library at user level. Most processors provide atomic test and set instructions that can be used to implement locks. A notable exception is the MIPS R3000, so the SGI Power series multiprocessors have special lock hardware on the bus. User processes gain access to the lock hardware by mapping pages of hardware locks into their address space using the segattach system call.

A Plan 9 process in a system call will block regardless of its weight. This means that when a program wishes to read from a slow device without blocking the entire calculation, it must fork a process to do the read for it. The solution is to start a satellite process that does the I/O and delivers the answer to the main program through shared memory or perhaps a pipe. This sounds onerous but works easily and efficiently in practice; in fact, most interactive Plan 9 applications, even relatively ordinary ones written in C, such as the text editor Sam [Pike87], run as multiprocess programs.

The kernel support for parallel programming in Plan 9 is a few hundred lines of portable code; a handful of simple primitives enable the problems to be handled cleanly at user level. Although the primitives work fine from C, they are particularly expressive from within Alef. The creation and management of slave I/O processes can be written in a few lines of Alef, providing the foundation for a consistent means of multiplexing data flows between arbitrary processes. Moreover, implementing it in a language rather than in the kernel ensures consistent semantics between all devices and provides a more general multiplexing primitive. Compare this to the UNIX select system call: select applies only to a restricted set of devices, legislates a style of multiprogramming in the kernel, does not extend across networks, is difficult to implement, and is hard to use.

Another reason parallel programming is important in Plan 9 is that multi-threaded user-level file servers are the preferred way to implement services. Examples of such servers include the programming environment Acme [Pike94], the name space exporting tool exportfs [PPTTW93], the HTTP daemon, and the network name servers cs and dns [PrWi93]. Complex applications such as Acme prove that careful operating system support can reduce the difficulty of writing multi-threaded applications without moving threading and synchronization primitives into the kernel.

Implementation of Name Spaces

User processes construct name spaces using three system calls: mount, bind, and unmount. The mount system call attaches a tree served by a file server to the current name space. Before calling mount, the client must (by outside means) acquire a connection to the server in the form of a file descriptor that may be written and read to transmit 9P messages. That file descriptor represents a pipe or network connection.

The mount call attaches a new hierarchy to the existing name space. The bind system call, on the other hand, duplicates some piece of existing name space at another point in the name space. The unmount system call allows components to be removed.

Using either `bind` or `mount`, multiple directories may be stacked at a single point in the name space. In Plan 9 terminology, this is a *union* directory and behaves like the concatenation of the constituent directories. A flag argument to `bind` and `mount` specifies the position of a new directory in the union, permitting new elements to be added either at the front or rear of the union or to replace it entirely. When a file lookup is performed in a union directory, each component of the union is searched in turn and the first match taken; likewise, when a union directory is read, the contents of each of the component directories is read in turn. Union directories are one of the most widely used organizational features of the Plan 9 name space. For instance, the directory `/bin` is built as a union of `/$cputype/bin` (program binaries), `/rc/bin` (shell scripts), and perhaps more directories provided by the user. This construction makes the shell `$PATH` variable unnecessary.

One question raised by union directories is which element of the union receives a newly created file. After several designs, we decided on the following. By default, directories in unions do not accept new files, although the `create` system call applied to an existing file succeeds normally. When a directory is added to the union, a flag to `bind` or `mount` enables create permission (a property of the name space) in that directory. When a file is being created with a new name in a union, it is created in the first directory of the union with create permission; if that creation fails, the entire `create` fails. This scheme enables the common use of placing a private directory anywhere in a union of public ones, while allowing creation only in the private directory.

By convention, kernel device file systems are bound into the `/dev` directory, but to bootstrap the name space building process it is necessary to have a notation that permits direct access to the devices without an existing name space. The root directory of the tree served by a device driver can be accessed using the syntax `#`*c*, where *c* is a unique character (typically a letter) identifying the *type* of the device. Simple device drivers serve a single level directory containing a few files. As an example, each serial port is represented by a data and a control file:

```
% bind -a '#t' /dev
% cd /dev
% ls -l eia*
--rw-rw-rw-  t 0 bootes  bootes  0 Feb 24 21:14 eia1
--rw-rw-rw-  t 0 bootes  bootes  0 Feb 24 21:14 eia1ctl
--rw-rw-rw-  t 0 bootes  bootes  0 Feb 24 21:14 eia2
--rw-rw-rw-  t 0 bootes  bootes  0 Feb 24 21:14 eia2ctl
```

The `bind` program is an encapsulation of the `bind` system call; its `-a` flag positions the new directory at the end of the union. The data files `eia1` and `eia2` may be read and written to communicate over the serial line. Instead of using special operations on these files to control the devices, commands written to the files `eia1ctl` and `eia2ctl` control the corresponding device; for example, writing the text string `b1200` to `/dev/eia1ctl` sets the speed of that line to 1200 baud. Compare this to the UNIX `ioctl` system call: in Plan 9, devices are controlled by textual messages, free of byte order problems, with clear semantics for reading and writing. It is common to configure or debug devices using shell scripts.

It is the universal use of the 9P protocol that connects Plan 9's components together to form a distributed system. Rather than inventing a unique protocol for each service such as `rlogin`, FTP, TFTP, and X windows, Plan 9 implements services in terms of operations on file objects, and then uses a single, well-documented protocol to exchange information between computers. Unlike NFS, 9P treats files as a sequence of bytes rather than blocks. Also unlike NFS, 9P is stateful: clients perform remote procedure calls to establish pointers to objects in the remote file server. These pointers are called file identifiers or *fids*. All operations on files supply a fid to identify an object in the remote file system.

The 9P protocol defines 17 messages, providing means to authenticate users, navigate fids around a file system hierarchy, copy fids, perform I/O, change file attributes, and create and delete files. Its complete specification is in Section 5 of the Programmer s Manual [9man]. Here is the procedure to gain access to the name hierarchy supplied by a server. A file server connection is established via a pipe or network connection. An initial session message performs a bilateral authentication between client and server. An attach message then connects a fid suggested by the client to the root of the server file tree. The attach message includes the identity of the user performing the attach; henceforth all fids derived from that fid will have permissions associated with that user. Multiple users may share the connection, but each must perform an attach to establish his or her identity.

The walk message moves a fid through a single level of the file system hierarchy. The clone message takes an established fid and produces a copy that points to the same file as the original. Its purpose is to enable walking to a file in a directory without losing the fid on the directory. The open message locks a fid to a specific file in the hierarchy, checks access permissions, and prepares the fid for I/O. The read and write messages allow I/O at arbitrary offsets in the file; the maximum size transferred is defined by the protocol. The clunk message indicates the client has no further use for a fid. The remove message behaves like clunk but causes the file associated with the fid to be removed and any associated resources on the server to be deallocated.

9P has two forms: RPC messages sent on a pipe or network connection and a procedural interface within the kernel. Since kernel device drivers are directly addressable, there is no need to pass messages to communicate with them; instead each 9P transaction is implemented by a direct procedure call. For each fid, the kernel maintains a local representation in a data structure called a *channel*, so all operations on files performed by the kernel involve a channel connected to that fid. The simplest example is a user process s file descriptors, which are indexes into an array of channels. A table in the kernel provides a list of entry points corresponding one to one with the 9P messages for each device. A system call such as read from the user translates into one or more procedure calls through that table, indexed by the type character stored in the channel: procread, eiaread, etc. Each call takes at least one channel as an argument. A special kernel driver, called the *mount* driver, translates procedure calls to messages, that is, it converts local procedure calls to remote ones. In effect, this special driver becomes a local proxy for the files served by a remote file server. The channel pointer in the local call is translated to the associated fid in the transmitted message.

The mount driver is the sole RPC mechanism employed by the system. The semantics of the supplied files, rather than the operations performed upon them, create a particular service such as the cpu command. The mount driver demultiplexes protocol messages between clients sharing a communication channel with a file server. For each outgoing RPC message, the mount driver allocates a buffer labeled by a small unique integer, called a *tag*. The reply to the RPC is labeled with the same tag, which is used by the mount driver to match the reply with the request.

The kernel representation of the name space is called the *mount table*, which stores a list of bindings between channels. Each entry in the mount table contains a pair of channels: a *from* channel and a *to* channel. Every time a walk succeeds in moving a channel to a new location in the name space, the mount table is consulted to see if a from channel matches the new name; if so the to channel is cloned and substituted for the original. Union directories are implemented by converting the to channel into a list of channels: a successful walk to a union directory returns a to channel that forms the head of a list of channels, each representing a component directory of the union. If a walk fails to find a file in the first directory of the union, the list is followed, the next component cloned, and walk tried on that directory.

Each file in Plan 9 is uniquely identified by a set of integers: the type of the channel (used as the index of the function call table), the server or device number distinguishing the server from others of the same type (decided locally by the driver), and a *qid* formed from two 32-bit numbers called *path* and *version*. The path is a unique file number assigned by a device driver or file server when a file is created. The version number is updated whenever the file is modified; as described in the next section, it can be used to maintain cache coherency between clients and servers.

The type and device number are analogous to UNIX major and minor device numbers; the qid is analogous to the i-number. The device and type connect the channel to a device driver and the qid identifies the file within that device. If the file recovered from a walk has the same type, device, and qid path as an entry in the mount table, they are the same file and the corresponding substitution from the mount table is made. This is how the name space is implemented.

File Caching

The 9P protocol has no explicit support for caching files on a client. The large memory of the central file server acts as a shared cache for all its clients, which reduces the total amount of memory needed across all machines in the network. Nonetheless, there are sound reasons to cache files on the client, such as a slow connection to the file server.

The version field of the qid is changed whenever the file is modified, which makes it possible to do some weakly coherent forms of caching. The most important is client caching of text and data segments of executable files. When a process execs a program, the file is re-opened and the qid's version is compared with that in the cache; if they match, the local copy is used. The same method can be used to build a local caching file server. This user-level server interposes on the 9P connection to the remote server and monitors the traffic, copying data to a local disk. When it sees a read of known data, it answers directly, while writes are passed on immediately—the cache is write-through—to keep the central copy up to date. This is transparent to processes on the terminal and requires no change to 9P; it works well on home machines connected over serial lines. A similar method can be applied to build a general client cache in unused local memory, but this has not been done in Plan 9.

Networks and Communication Devices

Network interfaces are kernel-resident file systems, analogous to the EIA device described earlier. Call setup and shutdown are achieved by writing text strings to the control file associated with the device; information is sent and received by reading and writing the data file. The structure and semantics of the devices is common to all networks so, other than a file name substitution, the same procedure makes a call using TCP over Ethernet as URP over Datakit [Fra80].

This example illustrates the structure of the TCP device:

```
% ls -lp /net/tcp
d-r-xr-xr-x   I 0 bootes  bootes  0 Feb 23 20:20 0
d-r-xr-xr-x   I 0 bootes  bootes  0 Feb 23 20:20 1
--rw-rw-rw-   I 0 bootes  bootes  0 Feb 23 20:20 clone
% ls -lp /net/tcp/0
--rw-rw----   I 0 rob     bootes  0 Feb 23 20:20 ctl
--rw-rw----   I 0 rob     bootes  0 Feb 23 20:20 data
--rw-rw----   I 0 rob     bootes  0 Feb 23 20:20 listen
--r--r--r--   I 0 bootes  bootes  0 Feb 23 20:20 local
--r--r--r--   I 0 bootes  bootes  0 Feb 23 20:20 remote
--r--r--r--   I 0 bootes  bootes  0 Feb 23 20:20 status
%
```

The top directory, /net/tcp, contains a clone file and a directory for each connection, numbered 0 to *n*. Each connection directory corresponds to an TCP/IP connection. Opening clone reserves an unused connection and returns its control file. Reading the control file returns the textual connection number, so the user process can construct the full name of the newly allocated connection directory. The local, remote, and status files are diagnostic; for example, remote contains the address (for TCP, the IP address and port number) of the remote side.

A call is initiated by writing a connect message with a network-specific address as its argument; for example, to open a Telnet session (port 23) to a remote machine with IP address 135.104.9.52, the string is:

```
connect 135.104.9.52!23
```

The write to the control file blocks until the connection is established; if the destination is unreachable, the write returns an error. Once the connection is established, the telnet application reads and writes the data file to talk to the remote Telnet daemon. On the other end, the Telnet daemon would start by writing

```
announce 23
```

to its control file to indicate its willingness to receive calls to this port. Such a daemon is called a *listener* in Plan 9.

A uniform structure for network devices cannot hide all the details of addressing and communication for dissimilar networks. For example, Datakit uses textual, hierarchical addresses unlike IP's 32-bit addresses, so an application given a control file must still know what network it represents. Rather than make every application know the addressing of every network, Plan 9 hides these details in a *connection server*, called cs. Cs is a file system mounted in a known place. It supplies a single control file that an application uses to discover how to connect to a host. The application writes the symbolic address and service name for the connection it wishes to make, and reads back the name of the clone file to open and the address to present to it. If there are multiple networks between the machines, cs presents a list of possible networks and addresses to be tried in sequence; it uses heuristics to decide the order. For instance, it presents the highest-bandwidth choice first.

A single library function called dial talks to cs to establish the connection. An application that uses dial needs no changes, not even recompilation, to adapt to new networks; the interface to cs hides the details.

The uniform structure for networks in Plan 9 makes the import command all that is needed to construct gateways.

Kernel structure for networks

The kernel plumbing used to build Plan 9 communications channels is called *streams* [Rit84][Presotto]. A stream is a bidirectional channel connecting a physical or pseudo-device to a user process. The user process inserts and removes data at one end of the stream; a kernel process acting on behalf of a device operates at the other end. A stream comprises a linear list of *processing modules*. Each module has both an upstream (toward the process) and downstream (toward the device) *put routine*. Calling the put routine of the module on either end of the stream inserts data into the stream. Each module calls the succeeding one to send data up or down the stream. Like UNIX streams [Rit84], Plan 9 streams can be dynamically configured.

The IL Protocol

The 9P protocol must run above a reliable transport protocol with delimited messages. 9P has no mechanism to recover from transmission errors and the system assumes that each read from a communication channel will return a single 9P message; it does not parse the data stream to discover message boundaries. Pipes and some network protocols already have these properties but the standard IP protocols do not. TCP does not delimit messages, while UDP [RFC768] does not provide reliable in-order delivery.

We designed a new protocol, called IL (Internet Link), to transmit 9P messages over IP. It is a connection-based protocol that provides reliable transmission of sequenced messages between machines. Since a process can have only a single outstanding 9P request, there is no need for flow control in IL. Like TCP, IL has adaptive timeouts: it scales acknowledge and retransmission times to match the network speed. This allows the protocol to perform well on both the Internet and on local Ethernets. Also, IL does no blind retransmission, to avoid adding to the congestion of busy networks. Full details are in another paper [PrWi95].

In Plan 9, the implementation of IL is smaller and faster than TCP. IL is our main Internet transport protocol.

Overview of authentication

Authentication establishes the identity of a user accessing a resource. The user requesting the resource is called the *client* and the user granting access to the resource is called the *server*. This is usually done under the auspices of a 9P attach message. A user may be a client in one authentication exchange and a server in another. Servers always act on behalf of some user, either a normal client or some administrative entity, so authentication is defined to be between users, not machines.

Each Plan 9 user has an associated DES [NBS77] authentication key; the users identity is verified by the ability to encrypt and decrypt special messages called challenges. Since knowledge of a users key gives access to that users resources, the Plan 9 authentication protocols never transmit a message containing a cleartext key.

Authentication is bilateral: at the end of the authentication exchange, each side is convinced of the others identity. Every machine begins the exchange with a DES key in memory. In the case of CPU and file servers, the key, user name, and domain name for the server are read from permanent storage, usually non-volatile RAM. In the case of terminals, the key is derived from a password typed by the user at boot time. A special machine, known as the *authentication server*, maintains a database of keys for all users in its administrative domain and participates in the authentication protocols.

The authentication protocol is as follows: after exchanging challenges, one party contacts the authentication server to create permission-granting *tickets* encrypted with each partys secret key and containing a new conversation key. Each party decrypts its own ticket and uses the conversation key to encrypt the other partys challenge.

This structure is somewhat like Kerberos [MBSS87], but avoids its reliance on synchronized clocks. Also unlike Kerberos, Plan 9 authentication supports a speaks for relation [LABW91] that enables one user to have the authority of another; this is how a CPU server runs processes on behalf of its clients.

Plan 9 s authentication structure builds secure services rather than depending on firewalls. Whereas firewalls require special code for every service penetrating the wall, the Plan 9 approach permits authentication to be done in a single place 9P for all services. For example, the cpu command works securely across the Internet.

Authenticating external connections

The regular Plan 9 authentication protocol is not suitable for text-based services such as Telnet or FTP. In such cases, Plan 9 users authenticate with hand-held DES calculators called *authenticators*. The authenticator holds a key for the user, distinct from the users normal authentication key. The user logs on to the authenticator using a 4-digit PIN. A correct PIN enables the authenticator for a challenge/response exchange with the server. Since a correct challenge/response exchange is valid only once and keys are never sent over the network, this procedure is not susceptible to replay attacks, yet is compatible with protocols like Telnet and FTP.

Special users

Plan 9 has no super-user. Each server is responsible for maintaining its own security, usually permitting access only from the console, which is protected by a password. For example, file servers have a unique administrative user called adm, with special privileges that apply only to commands typed at the server's physical console. These privileges concern the day-to-day maintenance of the server, such as adding new users and configuring disks and networks. The privileges do *not* include the ability to modify, examine, or change the permissions of any files. If a file is read-protected by a user, only that user may grant access to others.

CPU servers have an equivalent user name that allows administrative access to resources on that server such as the control files of user processes. Such permission is necessary, for example, to kill rogue processes, but does not extend beyond that server. On the other hand, by means of a key held in protected non-volatile RAM, the identity of the administrative user is proven to the authentication server. This allows the CPU server to authenticate remote users, both for access to the server itself and when the CPU server is acting as a proxy on their behalf.

Finally, a special user called none has no password and is always allowed to connect; anyone may claim to be none. None has restricted permissions; for example, it is not allowed to examine dump files and can read only world-readable files.

The idea behind none is analogous to the anonymous user in FTP services. On Plan 9, guest FTP servers are further confined within a special restricted name space. It disconnects guest users from system programs, such as the contents of /bin, but makes it possible to make local files available to guests by binding them explicitly into the space. A restricted name space is more secure than the usual technique of exporting an ad hoc directory tree; the result is a kind of cage around untrusted users.

The cpu command and proxied authentication

When a call is made to a CPU server for a user, say Peter, the intent is that Peter wishes to run processes with his own authority. To implement this property, the CPU server does the following when the call is received. First, the listener forks off a process to handle the call. This process changes to the user none to avoid giving away permissions if it is compromised. It then performs the authentication protocol to verify that the calling user really is Peter, and to prove to Peter that the machine is itself trustworthy. Finally, it reattaches to all relevant file servers using the authentication protocol to identify itself as Peter. In this case, the CPU server is a client of the file server and performs the client portion of the authentication exchange on behalf of Peter. The authentication server will give the process tickets to accomplish this only if the CPU server's administrative user name is allowed to *speak for* Peter.

The *speaks for* relation [LABW91] is kept in a table on the authentication server. To simplify the management of users computing in different authentication domains, it also contains mappings between user names in different domains, for example saying that user rtm in one domain is the same person as user rtmorris in another.

File Permissions

One of the advantages of constructing services as file systems is that the solutions to ownership and permission problems fall out naturally. As in UNIX, each file or directory has separate read, write, and execute/search permissions for the file owner, the file group, and anyone else. The idea of group is unusual: any user name is potentially a group name. A group is just a user with a list of other users in the group. Conventions make the distinction: most people have user names without group members, while groups have long lists of attached names. For example, the sys group traditionally has all the system programmers, and system files are accessible by group sys. Consider the following two lines of a user database stored on a server:

```
pjw:pjw:
sys::pjw,ken,philw,presotto
```

The first establishes user pjw as a regular user. The second establishes user sys as a group and lists four users who are *members* of that group. The empty colon-separated field is space for a user to be named as the *group leader*. If a group has a leader, that user has special permissions for the group, such as freedom to change the group permissions of files in that group. If no leader is specified, each member of the group is considered equal, as if each were the leader. In our example, only pjw can add members to his group, but all of sys members are equal partners in that group.

Regular files are owned by the user that creates them. The group name is inherited from the directory holding the new file. Device files are treated specially; the kernel may arrange the ownership and permissions of a file appropriate to the user accessing the file.

A good example of the generality this offers is process files, which are owned and read-protected by the owner of the process. If the owner wants to let someone else access the memory of a process, for example to let the author of a program debug a broken image, the standard chmod command applied to the process files does the job.

Another unusual application of file permissions is the dump file system, which is not only served by the same file server as the original data, but represented by the same user database. Files in the dump are therefore given identical protection as files in the regular file system; if a file is owned by pjw and read-protected, once it is in the dump file system it is still owned by pjw and read-protected. Also, since the dump file system is immutable, the file cannot be changed; it is read-protected forever. Drawbacks are that if the file is readable but should have been read-protected, it is readable forever, and that user names are hard to re-use.

Performance

As a simple measure of the performance of the Plan 9 kernel, we compared the time to do some simple operations on Plan 9 and on SGIs IRIX Release 5.3 running on an SGI Challenge M with a 100MHz MIPS R4400 and a 1-megabyte secondary cache. The test program was written in Alef, compiled with the same compiler, and run on identical hardware, so the only variables are the operating system and libraries.

The program tests the time to do a context switch (rendezvous on Plan 9, blockproc on IRIX); a trivial system call (rfork(0) and nap(0)); and lightweight fork (rfork(RFPROC) and sproc(PR_SFDS|PR_SADDR)). It also measures the time to send a byte on a pipe from one process to another and the throughput on a pipe between two processes. The results appear in Table 1.

Test	Plan 9	IRIX
Context switch	39 µs	150 µs
System call	6 µs	36 µs
Light fork	1300 µs	2200 µs
Pipe latency	110 µs	200 µs
Pipe bandwidth	11678 KB/s	14545 KB/s

Table 1. Performance comparison.

Although the Plan 9 times are not spectacular, they show that the kernel is competitive with commercial systems.

Discussion

Plan 9 has a relatively conventional kernel; the system's novelty lies in the pieces outside the kernel and the way they interact. When building Plan 9, we considered all aspects of the system together, solving problems where the solution fit best. Sometimes the solution spanned many components. An example is the problem of heterogeneous instruction architectures, which is addressed by the compilers (different code characters, portable object code), the environment ($cputype and $objtype), the name space (binding in /bin), and other components. Sometimes many issues could be solved in a single place. The best example is 9P, which centralizes naming, access, and authentication. 9P is really the core of the system; it is fair to say that the Plan 9 kernel is primarily a 9P multiplexer.

Plan 9's focus on files and naming is central to its expressiveness. Particularly in distributed computing, the way things are named has profound influence on the system [Nee89]. The combination of local name spaces and global conventions to interconnect networked resources avoids the difficulty of maintaining a global uniform name space, while naming everything like a file makes the system easy to understand, even for novices. Consider the dump file system, which is trivial to use for anyone familiar with hierarchical file systems. At a deeper level, building all the resources above a single uniform interface makes interoperability easy. Once a resource exports a 9P interface, it can combine transparently with any other part of the system to build unusual applications; the details are hidden. This may sound object-oriented, but there are distinctions. First, 9P defines a fixed set of methods; it is not an extensible protocol. More important, files are well-defined and well-understood and come prepackaged with familiar methods of access, protection, naming, and networking. Objects, despite their generality, do not come with these attributes defined. By reducing object to file, Plan 9 gets some technology for free.

Nonetheless, it is possible to push the idea of file-based computing too far. Converting every resource in the system into a file system is a kind of metaphor, and metaphors can be abused. A good example of restraint is /proc, which is only a view of a process, not a representation. To run processes, the usual fork and exec calls are still necessary, rather than doing something like

```
cp /bin/date /proc/clone/mem
```

The problem with such examples is that they require the server to do things not under its control. The ability to assign meaning to a command like this does not imply the meaning will fall naturally out of the structure of answering the 9P requests it generates. As a related example, Plan 9 does not put machine's network names in the file name space. The network interfaces provide a very different model of naming, because using open, create, read, and write on such files would not offer a suitable place to encode all the details of call setup for an arbitrary network. This does not mean that the network interface cannot be file-like, just that it must have a more tightly defined structure.

What would we do differently next time? Some elements of the implementation are unsatisfactory. Using streams to implement network interfaces in the kernel allows protocols to be connected together dynamically, such as to attach the same TTY driver to TCP, URP, and IL connections, but Plan 9 makes no use of this configurability. (It was exploited, however, in the research UNIX system for which streams were invented.) Replacing streams by static I/O queues would simplify the code and make it faster.

Although the main Plan 9 kernel is portable across many machines, the file server is implemented separately. This has caused several problems: drivers that must be written twice, bugs that must be fixed twice, and weaker portability of the file system code. The solution is easy: the file server kernel should be maintained as a variant of the regular operating system, with no user processes and special compiled-in kernel processes to implement file service. Another improvement to the file system would be a change of internal structure. The WORM jukebox is the least reliable piece of the hardware, but because it holds the metadata of the file system, it must be present in order to serve files. The system could be restructured so the WORM is a backup device only, with the file system proper residing on magnetic disks. This would require no change to the external interface.

Although Plan 9 has per-process name spaces, it has no mechanism to give the description of a process's name space to another process except by direct inheritance. The cpu command, for example, cannot in general reproduce the terminal's name space; it can only re-interpret the user's login profile and make substitutions for things like the name of the binary directory to load. This misses any local modifications made before running cpu. It should instead be possible to capture the terminal's name space and transmit its description to a remote process.

Despite these problems, Plan 9 works well. It has matured into the system that supports our research, rather than being the subject of the research itself. Experimental new work includes developing interfaces to faster networks, file caching in the client kernel, encapsulating and exporting name spaces, and the ability to re-establish the client state after a server crash. Attention is now focusing on using the system to build distributed applications.

One reason for Plan 9's success is that we use it for our daily work, not just as a research tool. Active use forces us to address shortcomings as they arise and to adapt the system to solve our problems. Through this process, Plan 9 has become a comfortable, productive programming environment, as well as a vehicle for further systems research.

References

[9man] *Plan 9 Programmer's Manual, Volume 1*, AT&T Bell Laboratories, Murray Hill, NJ, 1995.

[ANSIC] *American National Standard for Information Systems Programming Language C*, American National Standards Institute, Inc., New York, 1990.

[Duff90] Tom Duff, "Rc - A Shell for Plan 9 and UNIX systems", *Proc. of the Summer 1990 UKUUG Conf.*, London, July, 1990, pp. 21-33, reprinted, in a different form, in this volume.

[Fra80] A.G. Fraser, "Datakit A Modular Network for Synchronous and Asynchronous Traffic", *Proc. Int. Conf. on Commun.*, June 1980, Boston, MA.

[FSSUTF] *File System Safe UCS Transformation Format (FSS-UTF)*, X/Open Preliminary Specification, 1993. ISO designation is ISO/IEC JTC1/SC2/WG2 N 1036, dated 1994-08-01.

[ISO10646] ISO/IEC DIS 10646-1:1993 *Information technology Universal Multiple-Octet Coded Character Set (UCS) Part 1: Architecture and Basic Multilingual Plane.*

[Kill84] T. J Killian, "Processes as Files", *USENIX Summer 1984 Conf. Proc.*, June 1984, Salt Lake City, UT.

[LABW91] Butler Lampson, Martín Abadi, Michael Burrows, and Edward Wobber, "Authentication in Distributed Systems: Theory and Practice", *Proc. 13th ACM Symp. on Op. Sys. Princ.*, Asilomar, 1991, pp. 165-182.

[MBSS87] S. P. Miller, B. C. Neumann, J. I. Schiller, and J. H. Saltzer, Kerberos Authentication and Authorization System, Massachusetts Institute of Technology, 1987.

[NBS77] National Bureau of Standards (U.S.), *Federal Information Processing Standard 46*, National Technical Information Service, Springfield VA, 1977.

[Nee89] R. Needham, Names, in *Distributed systems*, S. Mullender, ed., Addison Wesley, 1989

[NeHe82] R. M. Needham and A. J. Herbert, *The Cambridge Distributed Computing System*, Addison-Wesley, London, 1982

[Neu92] B. Clifford Neuman, The Prospero File System, *USENIX File Systems Workshop Proc.*, Ann Arbor, 1992, pp. 13-28.

[OCDNW88] John Ousterhout, Andrew Cherenson, Fred Douglis, Mike Nelson, and Brent Welch, The Sprite Network Operating System, *IEEE Computer*, 21(2), 23-38, Feb. 1988.

[Pike87] Rob Pike, The Text Editor sam, *Software – Practice and Experience*, Nov 1987, **17** (11), pp. 813-845; reprinted in this volume.

[Pike91] Rob Pike, 8½, the Plan 9 Window System, *USENIX Summer Conf. Proc.*, Nashville, June, 1991, pp. 257-265, reprinted in this volume.

[Pike93] Rob Pike and Ken Thompson, Hello World or °±»·¼-Á± °ÌÃ¼µ or こんにちは 世界, *USENIX Winter Conf. Proc.*, San Diego, 1993, pp. 43-50, reprinted in this volume.

[Pike94] Rob Pike, Acme: A User Interface for Programmers, *USENIX Proc. of the Winter 1994 Conf.*, San Francisco, CA,

[Pike95] Rob Pike, How to Use the Plan 9 C Compiler, *Plan 9 Programmer's Manual, Volume 2*, AT&T Bell Laboratories, Murray Hill, NJ, 1995.

[POSIX] *Information Technology Portable Operating System Interface (POSIX) Part 1: System Application Program Interface (API) [C Language]*, IEEE, New York, 1990.

[PPTTW93] Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, and Phil Winterbottom, The Use of Name Spaces in Plan 9, *Op. Sys. Rev.*, Vol. 27, No. 2, April 1993, pp. 72-76, reprinted in this volume.

[Presotto] Dave Presotto, Multiprocessor Streams for Plan 9, *UKUUG Summer 1990 Conf. Proc.*, July 1990, pp. 11-19.

[PrWi93] Dave Presotto and Phil Winterbottom, The Organization of Networks in Plan 9, *USENIX Proc. of the Winter 1993 Conf.*, San Diego, CA, pp. 43-50, reprinted in this volume.

[PrWi95] Dave Presotto and Phil Winterbottom, The IL Protocol, *Plan 9 Programmer's Manual, Volume 2*, AT&T Bell Laboratories, Murray Hill, NJ, 1995.

[RFC768] J. Postel, RFC768, *User Datagram Protocol, DARPA Internet Program Protocol Specification*, August 1980.

[RFC793] RFC793, *Transmission Control Protocol, DARPA Internet Program Protocol Specification*, September 1981.

[Rao91] Herman Chung-Hwa Rao, *The Jade File System*, (Ph. D. Dissertation), Dept. of Comp. Sci, University of Arizona, TR 91-18.

[Rit84] D. M. Ritchie, A Stream Input-Output System, *AT&T Bell Laboratories Technical Journal*, **63** (8), October, 1984.

[Tric95] Howard Trickey, APE The ANSI/POSIX Environment, *Plan 9 Programmer's Manual, Volume 2*, AT&T Bell Laboratories, Murray Hill, NJ, 1995.

[Unicode] *The Unicode Standard, Worldwide Character Encoding, Version 1.0, Volume 1*, The Unicode Consortium, Addison Wesley, New York, 1991.

[UNIX85] *UNIX Time-Sharing System Programmer's Manual, Research Version, Eighth Edition, Volume 1.* AT&T Bell Laboratories, Murray Hill, NJ, 1985.

[Welc94] Brent Welch, A Comparison of Three Distributed File System Architectures: Vnode, Sprite, and Plan 9, *Computing Systems*, 7(2), pp. 175-199, Spring, 1994.

[Wint95] Phil Winterbottom, Alef Language Reference Manual, *Plan 9 Programmer's Manual, Volume 2*, AT&T Bell Laboratories, Murray Hill, NJ, 1995.

The Use of Name Spaces in Plan 9

Rob Pike
Dave Presotto
Ken Thompson
Howard Trickey
Phil Winterbottom

Bell Laboratories
Murray Hill, New Jersey 07974
USA

ABSTRACT

Plan 9 is a distributed system built at the Computing Sciences Research Center of AT&T Bell Laboratories (now Lucent Technologies, Bell Labs) over the last few years. Its goal is to provide a production-quality system for software development and general computation using heterogeneous hardware and minimal software. A Plan 9 system comprises CPU and file servers in a central location connected together by fast networks. Slower networks fan out to workstation-class machines that serve as user terminals. Plan 9 argues that given a few carefully implemented abstractions it is possible to produce a small operating system that provides support for the largest systems on a variety of architectures and networks. The foundations of the system are built on two ideas: a per-process name space and a simple message-oriented file system protocol.

The operating system for the CPU servers and terminals is structured as a traditional kernel: a single compiled image containing code for resource management, process control, user processes, virtual memory, and I/O. Because the file server is a separate machine, the file system is not compiled in, although the management of the name space, a per-process attribute, is. The entire kernel for the multiprocessor SGI Power Series machine is 25000 lines of C, the largest part of which is code for four networks including the Ethernet with the Internet protocol suite. Fewer than 1500 lines are machine-specific, and a functional kernel with minimal I/O can be put together from source files totaling 6000 lines. [Pike90]

The system is relatively small for several reasons. First, it is all new: it has not had time to accrete as many fixes and features as other systems. Also, other than the network protocol, it adheres to no external interface; in particular, it is not Unix-compatible. Economy stems from careful selection of services and interfaces. Finally, wherever possible the system is built around two simple ideas: every resource in the system, either local or remote, is represented by a hierarchical file system; and a user or process assembles a private view of the system by constructing a file *name space* that connects these resources. [Needham]

Appeared in *Operating Systems Review*, Vol. 27, #2, April 1993, pp. 72-76 (reprinted from *Proceedings of the 5th ACM SIGOPS European Workshop*, Mont Saint-Michel, 1992, Paper n° 34).

File Protocol

All resources in Plan 9 look like file systems. That does not mean that they are repositories for permanent files on disk, but that the interface to them is file-oriented: finding files (resources) in a hierarchical name tree, attaching to them by name, and accessing their contents by read and write calls. There are dozens of file system types in Plan 9, but only a few represent traditional files. At this level of abstraction, files in Plan 9 are similar to objects, except that files are already provided with naming, access, and protection methods that must be created afresh for objects. Object-oriented readers may approach the rest of this paper as a study in how to make objects look like files.

The interface to file systems is defined by a protocol, called 9P, analogous but not very similar to the NFS protocol. The protocol talks about files, not blocks; given a connection to the root directory of a file server, the 9P messages navigate the file hierarchy, open files for I/O, and read or write arbitrary bytes in the files. 9P contains 17 message types: three for initializing and authenticating a connection and fourteen for manipulating objects. The messages are generated by the kernel in response to user- or kernel-level I/O requests. Here is a quick tour of the major message types. The auth and attach messages authenticate a connection, established by means outside 9P, and validate its user. The result is an authenticated *channel* that points to the root of the server. The clone message makes a new channel identical to an existing channel, which may be moved to a file on the server using a walk message to descend each level in the hierarchy. The stat and wstat messages read and write the attributes of the file pointed to by a channel. The open message prepares a channel for subsequent read and write messages to access the contents of the file, while create and remove perform, on the files, the actions implied by their names. The clunk message discards a channel without affecting the file. None of the 9P messages consider caching; file caches are provided, when needed, either within the server (centralized caching) or by implementing the cache as a transparent file system between the client and the 9P connection to the server (client caching).

For efficiency, the connection to local kernel-resident file systems, misleadingly called *devices*, is by regular rather than remote procedure calls. The procedures map one-to-one with 9P message types. Locally each channel has an associated data structure that holds a type field used to index a table of procedure calls, one set per file system type, analogous to selecting the method set for an object. One kernel-resident file system, the *mount device*, translates the local 9P procedure calls into RPC messages to remote services over a separately provided transport protocol such as TCP or IL, a new reliable datagram protocol, or over a pipe to a user process. Write and read calls transmit the messages over the transport layer. The mount device is the sole bridge between the procedural interface seen by user programs and remote and user-level services. It does all associated marshaling, buffer management, and multiplexing and is the only integral RPC mechanism in Plan 9. The mount device is in effect a proxy object. There is no RPC stub compiler; instead the mount driver and all servers just share a library that packs and unpacks 9P messages.

Examples

One file system type serves permanent files from the main file server, a stand-alone multiprocessor system with a 350-gigabyte optical WORM jukebox that holds the data, fronted by a two-level block cache comprising 7 gigabytes of magnetic disk and 128 megabytes of RAM. Clients connect to the file server using any of a variety of networks and protocols and access files using 9P. The file server runs a distinct operating system and has no support for user processes; other than a restricted set of commands available on the console, all it does is answer 9P messages from clients.

Once a day, at 5:00 AM, the file server sweeps through the cache blocks and marks dirty blocks copy-on-write. It creates a copy of the root directory and labels it with the current date, for example 1995/0314. It then starts a background process to copy the dirty blocks to the WORM. The result is that the server retains an image of the file system as it was early each morning. The set of old root directories is accessible using 9P, so a client may examine backup files using ordinary commands. Several advantages stem from having the backup service implemented as a plain file system. Most obviously, ordinary commands can access them. For example, to see when a bug was fixed

```
grep 'mouse bug fix' 1995/*/sys/src/cmd/8½/file.c
```

The owner, access times, permissions, and other properties of the files are also backed up. Because it is a file system, the backup still has protections; it is not possible to subvert security by looking at the backup.

The file server is only one type of file system. A number of unusual services are provided within the kernel as local file systems. These services are not limited to I/O devices such as disks. They include network devices and their associated protocols, the bitmap display and mouse, a representation of processes similar to /proc [Killian], the name/value pairs that form the environment passed to a new process, profiling services, and other resources. Each of these is represented as a file system — directories containing sets of files — but the constituent files do not represent permanent storage on disk. Instead, they are closer in properties to UNIX device files.

For example, the *console* device contains the file /dev/cons, similar to the UNIX file /dev/console: when written, /dev/cons appends to the console typescript; when read, it returns characters typed on the keyboard. Other files in the console device include /dev/time, the number of seconds since the epoch, /dev/cputime, the computation time used by the process reading the device, /dev/pid, the process id of the process reading the device, and /dev/user, the login name of the user accessing the device. All these files contain text, not binary numbers, so their use is free of byte-order problems. Their contents are synthesized on demand when read; when written, they cause modifications to kernel data structures.

The *process* device contains one directory per live local process, named by its numeric process id: /proc/1, /proc/2, etc. Each directory contains a set of files that access the process. For example, in each directory the file mem is an image of the virtual memory of the process that may be read or written for debugging. The text file is a sort of link to the file from which the process was executed; it may be opened to read the symbol tables for the process. The ctl file may be written textual messages such as stop or kill to control the execution of the process. The status file contains a fixed-format line of text containing information about the process: its name, owner, state, and so on. Text strings written to the note file are delivered to the process as *notes*, analogous to UNIX signals. By providing these services as textual I/O on files rather than as system calls (such as kill) or special-purpose operations (such as ptrace), the Plan 9 process device simplifies the implementation of debuggers and related programs. For example, the command

```
cat /proc/*/status
```

is a crude form of the ps command; the actual ps merely reformats the data so obtained.

The *bitmap* device contains three files, /dev/mouse, /dev/screen, and /dev/bitblt, that provide an interface to the local bitmap display (if any) and pointing device. The mouse file returns a fixed-format record containing 1 byte of button state and 4 bytes each of *x* and *y* position of the mouse. If the mouse has not moved since the file was last read, a subsequent read will block. The screen file contains a memory image of the contents of the display; the bitblt file provides a procedural interface. Calls to the graphics library are translated into messages that are written to the bitblt file to perform bitmap graphics operations. (This is essentially a nested RPC protocol.)

The various services being used by a process are gathered together into the processs *name space*, a single rooted hierarchy of file names. When a process forks, the child process shares the name space with the parent. Several system calls manipulate name spaces. Given a file descriptor fd that holds an open communications channel to a service, the call

```
mount(int fd, char *old, int flags)
```

authenticates the user and attaches the file tree of the service to the directory named by old . The flags specify how the tree is to be attached to old : replacing the current contents or appearing before or after the current contents of the directory. A directory with several services mounted is called a *union* directory and is searched in the specified order.The call

```
bind(char *new, char *old, int flags)
```

takes the portion of the existing name space visible at new , either a file or a directory, and makes it also visible at old . For example,

```
bind("1995/0301/sys/include", "/sys/include", REPLACE)
```

causes the directory of include files to be overlaid with its contents from the dump on March first.

A process is created by the rfork system call, which takes as argument a bit vector defining which attributes of the process are to be shared between parent and child instead of copied. One of the attributes is the name space: when shared, changes made by either process are visible in the other; when copied, changes are independent.

Although there is no global name space, for a process to function sensibly the local name spaces must adhere to global conventions. Nonetheless, the use of local name spaces is critical to the system. Both these ideas are illustrated by the use of the name space to handle heterogeneity. The binaries for a given architecture are contained in a directory named by the architecture, for example /mips/bin ; in use, that directory is bound to the conventional location /bin . Programs such as shell scripts need not know the CPU type they are executing on to find binaries to run. A directory of private binaries is usually unioned with /bin . (Compare this to the *ad hoc* and special-purpose idea of the PATH variable, which is not used in the Plan 9 shell.) Local bindings are also helpful for debugging, for example by binding an old library to the standard place and linking a program to see if recent changes to the library are responsible for a bug in the program.

The window system, 8½ [Pike91], is a server for files such as /dev/cons and /dev/bitblt . Each client sees a distinct copy of these files in its local name space: there are many instances of /dev/cons , each served by 8½ to the local name space of a window. Again, 8½ implements services using local name spaces plus the use of I/O to conventionally named files. Each client just connects its standard input, output, and error files to /dev/cons , with analogous operations to access bitmap graphics. Compare this to the implementation of /dev/tty on UNIX, which is done by special code in the kernel that overloads the file, when opened, with the standard input or output of the process. Special arrangement must be made by a UNIX window system for /dev/tty to behave as expected; 8½ instead uses the provision of the corresponding file as its central idea, which to succeed depends critically on local name spaces.

The environment 8½ provides its clients is exactly the environment under which it is implemented: a conventional set of files in /dev . This permits the window system to be run recursively in one of its own windows, which is handy for debugging. It also means that if the files are exported to another machine, as described below, the window system or client applications may be run transparently on remote machines, even ones without graphics hardware. This mechanism is used for Plan 9's implementation of the X window system: X is run as a client of 8½, often on a remote machine with lots of memory. In this configuration, using Ethernet to connect MIPS machines, we measure only a 10% degradation in graphics performance relative to running X on a bare Plan 9 machine.

An unusual application of these ideas is a statistics-gathering file system implemented by a command called iostats. The command encapsulates a process in a local name space, monitoring 9P requests from the process to the outside world the name space in which iostats is itself running. When the command completes, iostats reports usage and performance figures for file activity. For example

```
iostats 8½
```

can be used to discover how much I/O the window system does to the bitmap device, font files, and so on.

The import command connects a piece of name space from a remote system to the local name space. Its implementation is to dial the remote machine and start a process there that serves the remote name space using 9P. It then calls mount to attach the connection to the name space and finally dies; the remote process continues to serve the files. One use is to access devices not available locally. For example, to write a floppy one may say

```
import lab.pc /a: /n/dos
cp foo /n/dos/bar
```

The call to import connects the file tree from /a: on the machine lab.pc (which must support 9P) to the local directory /n/dos. Then the file foo can be written to the floppy just by copying it across.

Another application is remote debugging:

```
import helix /proc
```

makes the process file system on machine helix available locally; commands such as ps then see helix's processes instead of the local ones. The debugger may then look at a remote process:

```
db /proc/27/text /proc/27/mem
```

allows breakpoint debugging of the remote process. Since db infers the CPU type of the process from the executable header on the text file, it supports cross-architecture debugging, too. Care is taken within db to handle issues of byte order and floating point; it is possible to breakpoint debug a big-endian MIPS process from a little-endian i386.

Network interfaces are also implemented as file systems [Presotto]. For example, /net/tcp is a directory somewhat like /proc: it contains a set of numbered directories, one per connection, each of which contains files to control and communicate on the connection. A process allocates a new connection by accessing /net/tcp/clone, which evaluates to the directory of an unused connection. To make a call, the process writes a textual message such as 'connect 135.104.53.2!512' to the ctl file and then reads and writes the data file. An rlogin service can be implemented in a few of lines of shell code.

This structure makes network gatewaying easy to provide. We have machines with Datakit interfaces but no Internet interface. On such a machine one may type

```
import  helix  /net
telnet  tcp!ai.mit.edu
```

The import uses Datakit to pull in the TCP interface from helix, which can then be used directly; the tcp! notation is necessary because we routinely use multiple networks and protocols on Plan 9 — it identifies the network in which ai.mit.edu is a valid name.

In practice we do not use rlogin or telnet between Plan 9 machines. Instead a command called cpu in effect replaces the CPU in a window with that on another machine, typically a fast multiprocessor CPU server. The implementation is to recreate the name space on the remote machine, using the equivalent of import to connect pieces of the terminal's name space to that of the process (shell) on the CPU server, making the terminal a file server for the CPU. CPU-local devices such as fast file system connections are still local; only terminal-resident devices are imported. The result is unlike UNIX rlogin, which moves into a distinct name space on the remote machine, or file sharing with NFS, which keeps the name space the same but forces processes to execute locally. Bindings in /bin may change because of a change in CPU architecture, and the networks involved may be different because of differing hardware, but the effect feels like simply speeding up the processor in the current name space.

Position

These examples illustrate how the ideas of representing resources as file systems and per-process name spaces can be used to solve problems often left to more exotic mechanisms. Nonetheless there are some operations in Plan 9 that are not mapped into file I/O. An example is process creation. We could imagine a message to a control file in /proc that creates a process, but the details of constructing the environment of the new process — its open files, name space, memory image, etc. — are too intricate to be described easily in a simple I/O operation. Therefore new processes on Plan 9 are created by fairly conventional fork and exec system calls; /proc is used only to represent and control existing processes.

Plan 9 does not attempt to map network name spaces into the file system name space, for several reasons. The different addressing rules for various networks and protocols cannot be mapped uniformly into a hierarchical file name space. Even if they could be, the various mechanisms to authenticate, select a service, and control the connection would not map consistently into operations on a file.

Shared memory is another resource not adequately represented by a file name space. Plan 9 takes care to provide mechanisms to allow groups of local processes to share and map memory. Memory is controlled by system calls rather than special files, however, since a representation in the file system would imply that memory could be imported from remote machines.

Despite these limitations, file systems and name spaces offer an effective model around which to build a distributed system. Used well, they can provide a uniform, familiar, transparent interface to a diverse set of distributed resources. They carry well-understood properties of access, protection, and naming. The integration of devices into the hierarchical file system was the best idea in UNIX. Plan 9 pushes the concept much further and shows that file systems, when used inventively, have plenty of scope for productive research.

References

[Killian] T. Killian, Processes as Files, USENIX Summer Conf. Proc., Salt Lake City, 1984
[Needham] R. Needham, Names, in *Distributed systems*, S. Mullender, ed., Addison Wesley, 1989
[Pike90] R. Pike, D. Presotto, K. Thompson, H. Trickey, Plan 9 from Bell Labs, UKUUG

Proc. of the Summer 1990 Conf., London, England, 1990
[Presotto] D. Presotto, Multiprocessor Streams for Plan 9, UKUUG Proc. of the Summer 1990 Conf., London, England, 1990
[Pike91] Pike, R., 8.5, The Plan 9 Window System, USENIX Summer Conf. Proc., Nashville, 1991

Security in Plan 9

Russ Cox, MIT LCS
Eric Grosse, Bell Labs
Rob Pike, Bell Labs
Dave Presotto, Avaya Labs and Bell Labs
Sean Quinlan, Bell Labs
{rsc,ehg,rob,presotto,seanq}@plan9.bell-labs.com

ABSTRACT

The security architecture of the Plan 9™ operating system has recently been redesigned to address some technical shortcomings. This redesign provided an opportunity also to make the system more convenient to use securely. Plan 9 has thus improved in two ways not usually seen together: it has become more secure *and* easier to use.

The central component of the new architecture is a per-user self-contained agent called *factotum*. *Factotum* securely holds a copy of the users keys and negotiates authentication protocols, on behalf of the user, with secure services around the network. Concentrating security code in a single program offers several advantages including: ease of update or repair to broken security software and protocols; the ability to run secure services at a lower privilege level; uniform management of keys for all services; and an opportunity to provide single sign on, even to unchanged legacy applications. *Factotum* has an unusual architecture: it is implemented as a Plan 9 file server.

1. Introduction

Secure computing systems face two challenges: first, they must employ sophisticated technology that is difficult to design and prove correct; and second, they must be easy for regular people to use. The question of ease of use is sometimes neglected, but it is essential: weak but easy-to-use security can be more effective than strong but difficult-to-use security if it is more likely to be used. People lock their front doors when they leave the house, knowing full well that a burglar is capable of picking the lock (or avoiding the door altogether); yet few would accept the cost and awkwardness of a bank vault door on the house even though that might reduce the probability of a robbery. A related point is that users need a clear model of how the security operates (if not how it actually provides security) in order to use it well; for example, the clarity of a lock icon on a web browser is offset by the confusing and typically insecure steps for installing X.509 certificates.

The security architecture of the Plan 9 operating system [Pike95] has recently been redesigned to make it both more secure and easier to use. By *security* we mean three things: first, the business of authenticating users and services; second, the safe handling, deployment, and use of keys and other secret information; and third, the use of

---

Appeared, in a slightly different form, in *Proc. of the 2002 Usenix Security Symposium*, San Francisco.

encryption and integrity checks to safeguard communications from prying eyes.

The old security architecture of Plan 9 had several engineering problems in common with other operating systems. First, it had an inadequate notion of security domain. Once a user provided a password to connect to a local file store, the system required that the same password be used to access all the other file stores. That is, the system treated all network services as belonging to the same security domain.

Second, the algorithms and protocols used in authentication, by nature tricky and difficult to get right, were compiled into the various applications, kernel modules, and file servers. Changes and fixes to a security protocol required that all components using that protocol needed to be recompiled, or at least relinked, and restarted.

Third, the file transport protocol, 9P [Pike93], that forms the core of the Plan 9 system, had its authentication protocol embedded in its design. This meant that fixing or changing the authentication used by 9P required deep changes to the system. If someone were to find a way to break the protocol, the system would be wide open and very hard to fix.

These and a number of lesser problems, combined with a desire for more widespread use of encryption in the system, spurred us to rethink the entire security architecture of Plan 9.

The centerpiece of the new architecture is an agent, called *factotum*, that handles the user's keys and negotiates all security interactions with system services and applications. Like a trusted assistant with a copy of the owner's keys, *factotum* does all the negotiation for security and authentication. Programs no longer need to be compiled with cryptographic code; instead they communicate with *factotum* agents that represent distinct entities in the cryptographic exchange, such as a user and server of a secure service. If a security protocol needs to be added, deleted, or modified, only *factotum* needs to be updated for all system services to be kept secure.

Building on *factotum*, we modified secure services in the system to move user authentication code into *factotum*; made authentication a separable component of the file server protocol; deployed new security protocols; designed a secure file store, called *secstore*, to protect our keys but make them easy to get when they are needed; designed a new kernel module to support transparent use of Transport Layer Security (TLS) [RFC2246]; and began using encryption for all communications within the system. The overall architecture is illustrated in Figure 1a.

Secure protocols and algorithms are well understood and are usually not the weakest link in a system's security. In practice, most security problems arise from buggy servers, confusing software, or administrative oversights. It is these practical problems that we are addressing. Although this paper describes the algorithms and protocols we are using, they are included mainly for concreteness. Our main intent is to present a simple security architecture built upon a small trusted code base that is easy to verify (whether by manual or automatic means), easy to understand, and easy to use.

Although it is a subjective assessment, we believe we have achieved our goal of ease of use. That we have achieved our goal of improved security is supported by our plan to move our currently private computing environment onto the Internet outside the corporate firewall. The rest of this paper explains the architecture and how it is used, to explain why a system that is easy to use securely is also safe enough to run in the open network.

2. An Agent for Security

One of the primary reasons for the redesign of the Plan 9 security infrastructure was to remove the authentication method both from the applications and from the kernel. Cryptographic code is large and intricate, so it should be packaged as a separate component that can be repaired or modified without altering or even relinking applications

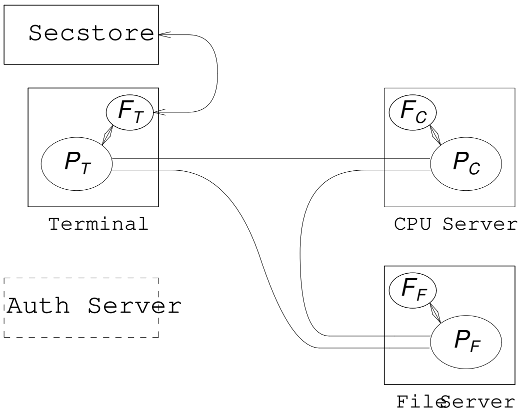

Figure 1a. Components of the security architecture. Each box is a (typically) separate machine; each ellipse a process. The ellipses labeled F_X are factotum processes; those labeled P_X are the pieces and proxies of a distributed program. The authentication server is one of several repositories for users security information that factotum processes consult as required. Secstore is a shared resource for storing private information such as keys; factotum consults it for the user during bootstrap.

and services that depend on it. If a security protocol is broken, it should be trivial to repair, disable, or replace it on the fly. Similarly, it should be possible for multiple programs to use a common security protocol without embedding it in each program.

Some systems use dynamically linked libraries (DLLs) to address these configuration issues. The problem with this approach is that it leaves security code in the same address space as the program using it. The interactions between the program and the DLL can therefore accidentally or deliberately violate the interface, weakening security. Also, a program using a library to implement secure services must run at a privilege level necessary to provide the service; separating the security to a different program makes it possible to run the services at a weaker privilege level, isolating the privileged code to a single, more trustworthy component.

Following the lead of the SSH agent [Ylon96], we give each user an agent process responsible for holding and using the users keys. The agent program is called factotum because of its similarity to the proverbial servant with the power to act on behalf of his master because he holds the keys to all the masters possessions. It is essential that factotum keep the keys secret and use them only in the owner s interest. Later we ll discuss some changes to the kernel to reduce the possibility of factotum leaking information inadvertently.

Factotum is implemented, like most Plan 9 services, as a file server. It is conventionally mounted upon the directory /mnt/factotum, and the files it serves there are analogous to virtual devices that provide access to, and control of, the services of the factotum. The next few sections describe the design of factotum and how it operates with the other pieces of Plan 9 to provide security services.

2.1. Logging in

To make the discussions that follow more concrete, we begin with a couple of examples showing how the Plan 9 security architecture appears to the user. These examples both involve a user gre logging in after booting a local machine. The user may or may not have a secure store in which all his keys are kept. If he does, factotum will prompt him for the password to the secure store and obtain keys from it, prompting only when a key isn t found in the store. Otherwise, factotum must prompt for each key.

In the typescripts, \n represents a literal newline character typed to force a default response. User input is in italics, and long lines are folded and indented to fit.

This first example shows a user logging in without help from the secure store. First, factotum prompts for a user name that the local kernel will use:

```
user[none]: gre
```

(Default responses appear in square brackets.) The kernel then starts accessing local resources and requests, through factotum, a user/password pair to do so:

```
!Adding key: dom=cs.bell-labs.com
     proto=p9sk1
user[gre]: \n
password: ****
```

Now the user is logged in to the local system, and the mail client starts up:

```
!Adding key: proto=apop
     server=plan9.bell-labs.com
user[gre]: \n
password: ****
```

Factotum is doing all the prompting and the applications being started are not even touching the keys. Note that it is always clear which key is being requested.

Now consider the same login sequence, but in the case where gre has a secure store account:

```
user[none]: gre
secstore password: *********
STA PIN+SecurID: *********
```

That's the last gre will hear from factotum unless an attempt is made to contact a system for which no key is kept in the secure store.

2.2. The factotum

Each computer running Plan 9 has one user id that owns all the resources on that system — the scheduler, local disks, network interfaces, etc. That user, the *host owner*, is the closest analogue in Plan 9 to a Unix root account (although it is far weaker; rather than having special powers, as its name implies the host owner is just a regular user that happens to own the resources of the local machine). On a single-user system, which we call a terminal, the host owner is the id of the terminal's user. Shared servers such as CPU servers normally have a pseudo-user that initially owns all resources. At boot time, the Plan 9 kernel starts a factotum executing as, and therefore with the privileges of, the host owner.

New processes run as the same user as the process which created them. When a process must take on the identity of a new user, such as to provide a login shell on a shared CPU server, it does so by proving to the host owner's factotum that it is authorized to do so. This is done by running an authentication protocol with factotum to prove that the process has access to secret information which only the new user should possess. For example, consider the setup in Figure 1a. If a user on the terminal wants to log in to the CPU server using the Plan 9 cpu service [Pike93], then P_T might be the cpu client program and P_C the cpu server. Neither P_C nor P_T knows the details of the authentication. They do need to be able to shuttle messages back and forth between the two factotums, but this is a generic function easily performed without knowing, or being able to extract, secrets in the messages. P_T will make a network connection to P_C. P_T and P_C will then relay messages between the factotum owned by the user, F_T, and the one owned by the CPU server, F_C, until mutual authentication has been established. Later sections describe the RPC between factotum and applications and the library functions to support proxy operations.

The kernel always uses a single local instance of factotum, running as the host owner, for its authentication purposes, but a regular user may start other factotum agents. In fact, the factotum representing the user need not be running on the same machine as its client. For instance, it is easy for a user on a CPU server, through standard Plan 9 operations, to replace the /mnt/factotum in the users private file name space on the server with a connection to the factotum running on the terminal. (The usual file system permissions prevent interlopers from doing so maliciously.) This permits secure operations on the CPU server to be transparently validated by the user factotum, so secrets need never leave the users terminal. The SSH agent [Ylon96] does much the same with special SSH protocol messages, but an advantage to making our agent a file system is that we need no new mechanism to access our remote agent; remote file access is sufficient.

Within factotum, each protocol is implemented as a state machine with a generic interface, so protocols are in essence pluggable modules, easy to add, modify, or drop. Writing a message to and reading a message from factotum each require a separate RPC and result in a single state transition. Therefore factotum always runs to completion on every RPC and never blocks waiting for input during any authentication. Moreover, the number of simultaneous authentications is limited only by the amount of memory we re willing to dedicate to representing the state machines.

Authentication protocols are implemented only within factotum, but adding and removing protocols does require relinking the binary, so factotum processes (but no others) need to be restarted in order to take advantage of new or repaired protocols.

At the time of writing, factotum contains authentication modules for the Plan 9 shared key protocol (p9sk1), SSH s RSA authentication, passwords in the clear, APOP, CRAM, PPPs CHAP, Microsoft PPPs MSCHAP, and VNCs challenge/response.

2.3. Local capabilities

A capability system, managed by the kernel, is used to empower factotum to grant permission to another process to change its user id. A kernel device driver implements two files, /dev/caphash and /dev/capuse. The write-only file /dev/caphash can be opened only by the host owner, and only once. Factotum opens this file immediately after booting.

To use the files, factotum creates a string of the form *userid1*@*userid2*@*random-string*, uses SHA1 HMAC to hash *userid1*@*userid2* with key *random-string*, and writes that hash to /dev/caphash. Factotum then passes the original string to another process on the same machine, running as user *userid1*, which writes the string to /dev/capuse. The kernel hashes the string and looks for a matching hash in its list. If it finds one, the writing processs user id changes from *userid1* to *userid2*. Once used, or if a timeout expires, the capability is discarded by the kernel.

The capabilities are local to the machine on which they are created. Hence a factotum running on one machine cannot pass capabilities to processes on another and expect them to work.

2.4. Keys

We define the word *key* to mean not only a secret, but also a description of the context in which that secret is to be used: the protocol, server, user, etc. to which it applies. That is, a key is a combination of secret and descriptive information used to authenticate the identities of parties transmitting or receiving information. The set of keys used in any authentication depends both on the protocol and on parameters passed by the program requesting the authentication.

Taking a tip from SDSI [RiLa], which represents security information as textual S-expressions, keys in Plan 9 are represented as plain UTF-8 text. Text is easily understood and manipulated by users. By contrast, a binary or other cryptic format can actually reduce overall security. Binary formats are difficult for users to examine and can only be cracked by special tools, themselves poorly understood by most users. For example, very few people know or understand what's inside their X.509 certificates. Most don't even know where in the system to find them. Therefore, they have no idea what they are trusting, and why, and are powerless to change their trust relationships. Textual, centrally stored and managed keys are easier to use and safer.

Plan 9 has historically represented databases as attribute/value pairs, since they are a good foundation for selection and projection operations. Factotum therefore represents the keys in the format *attribute* = *value*, where *attribute* is an identifier, possibly with a single-character prefix, and *value* is an arbitrary quoted string. The pairs themselves are separated by whitespace. For example, a Plan 9 key and an APOP key might be represented like this:

```
dom=bell-labs.com proto=p9sk1 user=gre
        !password='don''t tell'
proto=apop server=x.y.com user=gre
        !password='open sesame'
```

If a value is empty or contains whitespace or single quotes, it must be quoted; quotes are represented by doubled single quotes. Attributes that begin with an exclamation mark (!) are considered *secret*. Factotum will never let a secret value escape its address space and will suppress keyboard echo when asking the user to type one.

A program requesting authentication selects a key by providing a *query*, a list of elements to be matched by the key. Each element in the list is either an *attribute* = *value* pair, which is satisfied by keys with exactly that pair, or an attribute followed by a question mark, *attribute* ?, which is satisfied by keys with some pair specifying the attribute. A key matches a query if every element in the list is satisfied. For instance, to select the APOP key in the previous example, an APOP client process might specify the query

```
server=x.y.com proto=apop
```

Internally, factotum's APOP module would add the requirements of having user and !password attributes, forming the query

```
server=x.y.com proto=apop user? !password?
```

when searching for an appropriate key.

Factotum modules expect keys to have some well-known attributes. For instance, the proto attribute specifies the protocol module responsible for using a particular key, and protocol modules may expect other well-known attributes (many expect keys to have !password attributes, for example). Additional attributes can be used as comments or for further discrimination without intervention by factotum; for example, the APOP and IMAP mail clients conventionally include a server attribute to select an appropriate key for authentication.

Unlike in SDSI, keys in Plan 9 have no nested structure. This design keeps the representation simple and straightforward. If necessary, we could add a nested attribute or, in the manner of relational databases, an attribute that selects another tuple, but so far the simple design has been sufficient.

A simple common structure for all keys makes them easy for users to administer, but the set of attributes and their interpretation is still protocol-specific and can be subtle. Users may still need to consult a manual to understand all details. Many attributes (proto, user, password, server) are self-explanatory and our short experience has not uncovered any particular difficulty in handling keys. Things will likely get messier, however, when we grapple with public keys and their myriad components.

2.5. Protecting keys

Secrets must be prevented from escaping factotum. There are a number of ways they could leak: another process might be able to debug the agent process, the agent might swap out to disk, or the process might willingly disclose the key. The last is the easiest to avoid: secret information in a key is marked as such, and whenever factotum prints keys or queries for new ones, it is careful to avoid displaying secret information. (The only exception to this is the plaintext password protocol, which consists of sending the values of the user and !password attributes. Only keys tagged with proto=pass can have their passwords disclosed by this mechanism.)

Preventing the first two forms of leakage requires help from the kernel. In Plan 9, every process is represented by a directory in the /proc file system. Using the files in this directory, other processes could (with appropriate access permission) examine factotum's memory and registers. Factotum is protected from processes of other users by the default access bits of its /proc directory. However, we'd also like to protect the agent from other processes owned by the same user, both to avoid honest mistakes and to prevent an unattended terminal being exploited to discover secret passwords. To do this, we added a control message to /proc called private. Once the factotum process has written private to its /proc/*pid*/ctl file, no process can access factotum's memory through /proc. (Plan 9 has no other mechanism, such as /dev/kmem, for accessing a process's memory.)

Similarly, the agent's address space should not be swapped out, to prevent discovering unencrypted keys on the swapping media. The noswap control message in /proc prevents this scenario. Neither private nor noswap is specific to factotum. User-level file servers such as dossrv, which interprets FAT file systems, could use noswap to keep their buffer caches from being swapped to disk.

Despite our precautions, attackers might still find a way to gain access to a process running as the host owner on a machine. Although they could not directly access the keys, attackers could use the local factotum to perform authentications for them. In the case of some keys, for example those locking bank accounts, we want a way to disable or at least detect such access. That is the role of the confirm attribute in a key. Whenever a key with a confirm attribute is accessed, the local user must confirm use of the key via a local GUI. The next section describes the actual mechanism.

We have not addressed leaks possible as a result of someone rebooting or resetting a machine running factotum. For example, someone could reset a machine and reboot it with a debugger instead of a kernel, allowing them to examine the contents of memory and find keys. We have not found a satisfactory solution to this problem.

2.6. Factotum transactions

External programs manage factotum's internal key state through its file interface, writing textual key and delkey commands to the /mnt/factotum/ctl file. Both commands take a list of attributes as an argument. Key creates a key with the given attributes, replacing any extant key with an identical set of public attributes. Delkey deletes all keys that match the given set of attributes. Reading the ctl file returns a list of keys, one per line, displaying only public attributes. The following example illustrates these interactions.

```
% cd /mnt/factotum
% ls -l
-lrw-------   gre  gre  0  Jan  30  22:17  confirm
--rw-------   gre  gre  0  Jan  30  22:17  ctl
-lr--------   gre  gre  0  Jan  30  22:17  log
-lrw-------   gre  gre  0  Jan  30  22:17  needkey
--r--r--r--   gre  gre  0  Jan  30  22:17  proto
--rw-rw-rw-   gre  gre  0  Jan  30  22:17  rpc
% cat >ctl
key dom=bell-labs.com proto=p9sk1 user=gre
    !password='don''t tell'
key proto=apop server=x.y.com user=gre
    !password='bite me'
^D
% cat ctl
key dom=bell-labs.com proto=p9sk1 user=gre
key proto=apop server=x.y.com user=gre
% echo 'delkey proto=apop' >ctl
% cat ctl
key dom=bell-labs.com proto=p9sk1 user=gre
%
```

(A file with the `l` bit set can be opened by only one process at a time.)

The heart of the interface is the `rpc` file. Programs authenticate with `factotum` by writing a request to the `rpc` file and reading back the reply; this sequence is called an RPC *transaction*. Requests and replies have the same format: a textual verb possibly followed by arguments, which may be textual or binary. The most common reply verb is `ok`, indicating success. An RPC session begins with a `start` transaction; the argument is a key query as described earlier. Once started, an RPC conversation usually consists of a sequence of `read` and `write` transactions. If the conversation is successful, an `authinfo` transaction will return information about the identities learned during the transaction. The `attr` transaction returns a list of attributes for the current conversation; the list includes any attributes given in the `start` query as well as any public attributes from keys being used.

As an example of the `rpc` file in action, consider a mail client connecting to a mail server and authenticating using the POP3 protocol's APOP challenge-response command. There are four programs involved: the mail client P_C, the client `factotum` F_C, the mail server P_S, and the server `factotum` F_S. All authentication computations are handled by the `factotum` processes. The mail programs' role is just to relay messages.

At startup, the mail server at `x.y.com` begins an APOP conversation with its `factotum` to obtain the banner greeting, which includes a challenge:

$P_S \rightarrow F_S$: `start proto=apop role=server`
$F_S \rightarrow P_S$: `ok`
$P_S \rightarrow F_S$: `read`
$F_S \rightarrow P_S$: `ok +OK POP3` *challenge*

Having obtained the challenge, the server greets the client:

$P_S \rightarrow P_C$: `+OK POP3` *challenge*

The client then uses an APOP conversation with its `factotum` to obtain a response:

$P_C \rightarrow F_C$: `start proto=apop role=client server=x.y.com`
$F_C \rightarrow P_C$: `ok`
$P_C \rightarrow F_C$: `write +OK POP3` *challenge*
$F_C \rightarrow P_C$: `ok`
$P_C \rightarrow F_C$: `read`
$F_C \rightarrow P_C$: `ok APOP gre` *response*

Factotum requires that start requests include a proto attribute, and the APOP module requires an additional role attribute, but the other attributes are optional and only restrict the key space. Before responding to the start transaction, the client factotum looks for a key to use for the rest of the conversation. Because of the arguments in the start request, the key must have public attributes proto=apop and server=x.y.com; as mentioned earlier, the APOP module additionally requires that the key have user and !password attributes. Now that the client has obtained a response from its factotum, it echoes that response to the server:

$P_C \rightarrow P_S$: `APOP gre` *response*

Similarly, the server passes this message to its factotum and obtains another to send back.

$P_S \rightarrow F_S$: `write APOP gre` *response*
$F_S \rightarrow P_S$: `ok`
$P_S \rightarrow F_S$: `read`
$F_S \rightarrow P_S$: `ok +OK welcome`

$P_S \rightarrow P_C$: `+OK welcome`

Now the authentication protocol is done, and the server can retrieve information about what the protocol established.

$P_S \rightarrow F_S$: `authinfo`
$F_S \rightarrow P_S$: `ok client=gre capability=` *capability*

The authinfo data is a list of *attr* = *value* pairs, here a client user name and a capability. (Protocols that establish shared secrets or provide mutual authentication indicate this by adding appropriate *attr* = *value* pairs.) The capability can be used by the server to change its identity to that of the client, as described earlier. Once it has changed its identity, the server can access and serve the client's mailbox.

Two more files provide hooks for a graphical factotum control interface. The first, confirm, allows the user detailed control over the use of certain keys. If a key has a confirm= attribute, then the user must approve each use of the key. A separate program with a graphical interface reads from the confirm file to see when a confirmation is necessary. The read blocks until a key usage needs to be approved, whereupon it will return a line of the form

`confirm tag=1` *attributes*

requesting permission to use the key with those public attributes. The graphical interface then prompts the user for approval and writes back

`tag=1 answer=yes`

(or answer=no).

The second file, needkey, diverts key requests. In the APOP example, if a suitable key had not been found during the start transaction, factotum would have indicated failure by returning a response indicating what key was needed:

$F_C \rightarrow P_C$: `needkey proto=apop server=x.y.com user? !password?`

A typical client would then prompt the user for the desired key information, create a new key via the ctl file, and then reissue the start request. If the needkey file is open, then instead of failing, the transaction will block, and the next read from the /mnt/factotum/needkey file will return a line of the form

needkey tag=1 *attributes*

The graphical interface then prompts the user for the needed key information, creates the key via the ctl file, and writes back tag=1 to resume the transaction.

The remaining files are informational and used for debugging. The proto file contains a list of supported protocols (to see what protocols the system supports, cat /mnt/factotum/proto), and the log file contains a log of operations and debugging output enabled by a debug control message.

The next few sections explain how factotum is used by system services.

3. Authentication in 9P

Plan 9 uses a remote file access protocol, 9P [Pike93], to connect to resources such as the file server and remote processes. The original design for 9P included special messages at the start of a conversation to authenticate the user. Multiple users can share a single connection, such as when a CPU server runs processes for many users connected to a single file server, but each must authenticate separately. The authentication protocol, similar to that of Kerberos [Stei88], used a sequence of messages passed between client, file server, and authentication server to verify the identities of the user, calling machine, and serving machine. One major drawback to the design was that the authentication method was defined by 9P itself and could not be changed. Moreover, there was no mechanism to relegate authentication to an external (trusted) agent, so a process implementing 9P needed, besides support for file service, a substantial body of cryptographic code to implement a handful of startup messages in the protocol.

A recent redesign of 9P addressed a number of file service issues outside the scope of this paper. On issues of authentication, there were two goals: first, to remove details about authentication from the protocol itself; second, to allow an external program to execute the authentication part of the protocol. In particular, we wanted a way to quickly incorporate ideas found in other systems such as SFS [Mazi99].

Since 9P is a file service protocol, the solution involved creating a new type of file to be served: an *authentication file*. Connections to a 9P service begin in a state that allows no general file access but permits the client to open an authentication file by sending a special message, generated by the new fauth system call:

```
afd = fauth(int fd, char *servicename);
```

Here fd is the user's file descriptor for the established network connection to the 9P server and servicename is the name of the desired service offered on that server, typically the file subsystem to be accessed. The returned file descriptor, afd , is a unique handle representing the authentication file created for this connection to authenticate to this service; it is analogous to a capability. The authentication file represented by afd is not otherwise addressable on the server, such as through the file name hierarchy. In all other respects, it behaves like a regular file; most important, it accepts standard read and write operations.

To prove its identity, the user process (via factotum) executes the authentication protocol, described in the next section of this paper, over the afd file descriptor with ordinary reads and writes. When client and server have successfully negotiated, the authentication file changes state so it can be used as evidence of authority in

Once identity is established, the process presents the (now verified) `afd` as proof of identity to the `mount` system call:

```
mount(int fd, int afd, char *mountpoint,
      int flag, char *servicename)
```

If the `mount` succeeds, the user now has appropriate permissions for the file hierarchy made visible at the mount point.

This sequence of events has several advantages. First, the actual authentication protocol is implemented using regular reads and writes, not special 9P messages, so they can be processed, forwarded, proxied, and so on by any 9P agent without special arrangement. Second, the business of negotiating the authentication by reading and writing the authentication file can be delegated to an outside agent, in particular `factotum`; the programs that implement the client and server ends of a 9P conversation need no authentication or cryptographic code. Third, since the authentication protocol is not defined by 9P itself, it is easy to change and can even be negotiated dynamically. Finally, since `afd` acts like a capability, it can be treated like one: handed to another process to give it special permissions; kept around for later use when authentication is again required; or closed to make sure no other process can use it.

All these advantages stem from moving the authentication negotiation into reads and writes on a separate file. As is often the case in Plan 9, making a resource (here authentication) accessible with a file-like interface reduces *a priori* the need for special interfaces.

3.1. Plan 9 shared key protocol

In addition to the various standard protocols supported by `factotum`, we use a shared key protocol for native Plan 9 authentication. This protocol provides backward compatibility with older versions of the system. One reason for the new architecture is to let us replace such protocols in the near future with more cryptographically secure ones.

P9sk1 is a shared key protocol that uses tickets much like those in the original Kerberos. The difference is that we've replaced the expiration time in Kerberos tickets with a random nonce parameter and a counter. We summarize it here:

$$
\begin{array}{ll}
C \rightarrow S: & nonce_C \\
S \rightarrow C: & nonce_S, uid_S, domain_S \\
 & \\
C \rightarrow A: & nonce_S, uid_S, domain_S, uid_C, \\
 & \quad factotum_C \\
A \rightarrow C: & K_C\{nonce_S, uid_C, uid_S, K_n\}, \\
 & \quad K_S\{nonce_S, uid_C, uid_S, K_n\} \\
 & \\
C \rightarrow S: & K_S\{nonce_S, uid_C, uid_S, K_n\}, \\
 & \quad K_n\{nonce_S, counter\} \\
S \rightarrow C: & K_n\{nonce_C, counter\}
\end{array}
$$

(Here $K\{x\}$ indicates x encrypted with DES key K.) The first two messages exchange nonces and server identification. After this initial exchange, the client contacts the authentication server to obtain a pair of encrypted tickets, one encrypted with the client key and one with the server key. The client relays the server ticket to the server. The server believes that the ticket is new because it contains $nonce_S$ and that the ticket is from the authentication server because it is encrypted in the server key K_S. The ticket is basically a statement from the authentication server that now uid_C and uid_S share a secret K_n. The authenticator $K_n\{nonce_S, counter\}$ convinces the server that the client knows K_n and thus must be uid_C. Similarly, authenticator $K_n\{nonce_C, counter\}$ convinces the client that the server knows K_n and thus must be uid_S. Tickets can be reused, without contacting the authentication server again, by incrementing the counter before each authenticator is generated.

In the future we hope to introduce a public key version of p9sk1, which would allow authentication even when the authentication server is not available.

3.2. The authentication server

Each Plan 9 security domain has an authentication server (AS) that all users trust to keep the complete set of shared keys. It also offers services for users and administrators to manage the keys, create and disable accounts, and so on. It typically runs on a standalone machine with few other services. The AS comprises two services, keyfs and authsrv.

Keyfs is a user-level file system that manages an encrypted database of user accounts. Each account is represented by a directory containing the files key, containing the Plan 9 key for p9sk1; secret for the challenge/response protocols (APOP, VNC, CHAP, MSCHAP, CRAM); log for authentication outcomes; expire for an expiration time; and status. If the expiration time passes, if the number of successive failed authentications exceeds 50, or if disabled is written to the status file, any attempt to access the key or secret files will fail.

Authsrv is a network service that brokers shared key authentications for the protocols p9sk1, APOP, VNC, CHAP, MSCHAP, and CRAM. Remote users can also call authsrv to change their passwords.

The p9sk1 protocol was described in the previous section. The challenge/response protocols differ in detail but all follow the general structure:

$$
\begin{array}{ll}
C \rightarrow S: & nonce_C \\
S \rightarrow C: & nonce_S, uid_S, domain_S \\
C \rightarrow A: & nonce_S, uid_S, domain_S, \\
 & \quad hostid_C, uid_C \\
A \rightarrow C: & K_C\{nonce_S, uid_C, uid_S, K_n\}, \\
 & \quad K_S\{nonce_S, uid_C, uid_S, K_n\} \\
C \rightarrow S: & K_S\{nonce_S, uid_C, uid_S, K_n\}, \\
 & \quad K_n\{nonce_S\} \\
S \rightarrow C: & K_n\{nonce_C\}
\end{array}
$$

The password protocol is:

$$
\begin{array}{ll}
C \rightarrow A: & uid_C \\
A \rightarrow C: & K_c\{K_n\} \\
C \rightarrow A: & K_n\{password_{old}, password_{new}\} \\
A \rightarrow C: & OK
\end{array}
$$

To avoid replay attacks, the pre-encryption clear text for each of the protocols (as well as for p9sk1) includes a tag indicating the encryption's role in the protocol. We elided them in these outlines.

3.3. Protocol negotiation

Rather than require particular protocols for particular services, we implemented a negotiation metaprotocol, *p9any*, which chooses the actual authentication protocol to use. P9any is used now by all native services on Plan 9.

The metaprotocol is simple. The callee sends a null-terminated string of the form:

v.n $proto_1$@$domain_1$ $proto_2$@$domain_2$ ...

where n is a decimal version number, $proto_k$ is the name of a protocol for which the factotum has a key, and $domain_k$ is the name of the domain in which the key is valid. The caller then responds *proto*@*domain*

indicating its choice. Finally the callee responds

OK

Any other string indicates failure. At this point the chosen protocol commences. The final fixed-length reply is used to make it easy to delimit the I/O stream should the chosen protocol require the caller rather than the callee to send the first message.

With this negotiation metaprotocol, the underlying authentication protocols used for Plan 9 services can be changed under any application just by changing the keys known by the factotum agents at each end.

P9any is vulnerable to man in the middle attacks to the extent that the attacker may constrain the possible choices by changing the stream. However, we believe this is acceptable since the attacker cannot force either side to choose algorithms that it is unwilling to use.

4. Library Interface to Factotum

Although programs can access factotum's services through its file system interface, it is more common to use a C library that packages the interaction. There are a number of routines in the library, not all of which are relevant here, but a few examples should give their flavor.

First, consider the problem of mounting a remote file server using 9P. An earlier discussion showed how the fauth and mount system calls use an authentication file, afd, as a capability, but not how factotum manages afd. The library contains a routine, amount (authenticated mount), that is used by most programs in preference to the raw fauth and mount calls. Amount engages factotum to validate afd; here is the complete code:

```
int
amount(int fd, char *mntpt,
	int flags, char *aname)
{
	int afd, ret;
	AuthInfo *ai;

	afd = fauth(fd, aname);
	if(afd >= 0){
		ai = auth_proxy(afd, amount_getkey,
			"proto=p9any role=client");
		if(ai != NULL)
			auth_freeAI(ai);
	}
	ret = mount(fd, afd, mntpt,
		flags, aname);
	if(afd >= 0)
		close(afd);
	return ret;
}
```

where parameter fd is a file descriptor returned by open or dial for a new connection to a file server. The conversation with factotum occurs in the call to auth_proxy, which specifies, as a key query, which authentication protocol to use (here the metaprotocol p9any) and the role being played (client). Auth_proxy will read and write the factotum files, and the authentication file descriptor afd, to validate the user's right to access the service. If the call is successful, any auxiliary data, held in an AuthInfo structure, is freed. In any case, the mount is then called with the (perhaps validated) afd. A 9P server can cause the fauth system call to fail, as an indication that authentication is not required to access the service.

The second argument to auth_proxy is a function, here amount_getkey, to be called if secret information such as a password or response to a challenge is required as part of the authentication. This function, of course, will provide this data to factotum as a key message on the /mnt/factotum/ctl file.

Although the final argument to auth_proxy in this example is a simple string, in general it can be a formatted-print specifier in the manner of printf, to enable the construction of more elaborate key queries.

As another example, consider the Plan 9 cpu service, which exports local devices to a shell process on a remote machine, typically to connect the local screen and keyboard to a more powerful computer. At heart, cpu is a superset of a service called exportfs [Pike93], which allows one machine to see an arbitrary portion of the file name space of another machine, such as to export the network device to another machine for gatewaying. However, cpu is not just exportfs because it also delivers signals such as interrupt and negotiates the initial environment for the remote shell.

To authenticate an instance of cpu requires factotum processes on both ends: the local, client end running as the user on a terminal and the remote, server end running as the host owner of the server machine. Here is schematic code for the two ends:

```
/* client */
int
p9auth(int fd)
{
  AuthInfo *ai;

  ai = auth_proxy(fd, auth_getkey,
    "proto=p9any role=client");
  if(ai == NULL)
    return -1;

  /* start cpu protocol here */
}

/* server */
int
srvp9auth(int fd, char *user)
{
  AuthInfo *ai;

  ai = auth_proxy(fd, NULL,
    "proto=p9any role=server");
  if(ai == NULL)
    return -1;
  /* set user id for server process */
  if(auth_chuid(ai, NULL) < 0)
    return -1;

  /* start cpu protocol here */
}
```

Auth_chuid encapsulates the negotiation to change a user id using the caphash and capuse files of the (server) kernel. Note that although the client process may ask the user for new keys, using auth_getkey, the server machine, presumably a shared machine with a pseudo-user for the host owner, sets the key-getting function to NULL.

5. Secure Store

Factotum keeps its keys in volatile memory, which must somehow be initialized at boot time. Therefore, factotum must be supplemented by a persistent store, perhaps a floppy disk containing a key file of commands to be copied into /mnt/factotum/ctl during bootstrap. But removable media are a nuisance to carry and are vulnerable to theft. Keys could be stored encrypted on a shared file system, but only if those keys are not necessary for authenticating to the file system in the first place. Even if the keys are encrypted under a user password, a thief might well succeed with a dictionary attack. Other risks of local storage are loss of the contents through mechanical mishap or dead batteries. Thus for convenience and safety we provide a secstore (secure store) server in the network to hold each users permanent list of keys, *key file*.

Secstore is a file server for encrypted data, used only during bootstrapping. It must provide strong authentication and resistance to passive and active protocol attacks while assuming nothing more from the client than a password. Once factotum has loaded the key file, further encrypted or authenticated file storage can be accomplished by standard mechanisms.

The cryptographic technology that enables secstore is a form of encrypted key exchange called PAK [Boyk00], analogous to EKE [Bell93], SRP [Wu98], or SPEKE [Jabl]. PAK was chosen because it comes with a proof of equivalence in strength to Diffie-Hellman; subtle flaws in some earlier encrypted key exchange protocols and implementations have encouraged us to take special care. In outline, the PAK protocol is:

$$
\begin{aligned}
C \rightarrow S&: \; C, g^x H \\
S \rightarrow C&: \; S, g^y, hash(g^{xy}, C, S) \\
C \rightarrow S&: \; hash(g^{xy}, S, C)
\end{aligned}
$$

where H is a preshared secret between client C and server S. There are several variants of PAK, all presented in papers mainly concerned with proofs of cryptographic properties. To aid implementers, we have distilled a description of the specific version we use into an Appendix to this paper. The Plan 9 open source license provides for use of Lucents encrypted key exchange patents in this context.

As a further layer of defense against password theft, we provide (within the encrypted channel $C \rightarrow S$) information that is validated at a RADIUS server, such as the digits from a hardware token [RFC2138]. This provides two-factor authentication, which potentially requires tricking two independent administrators in any attack by social engineering.

The key file stored on the server is encrypted with AES (Rijndael) using CBC with a 10-byte initialization vector and trailing authentication padding. All this is invisible to the user of secstore. For that matter, it is invisible to the secstore server as well; if the AES Modes of Operation are standardized and a new encryption format designed, it can be implemented by a client without change to the server. The secstore is deliberately not backed up; the user is expected to use more than one secstore or save the key file on removable media and lock it away. The users password is hashed to create the H used in the PAK protocol; a different hash of the password is used as the file encryption key. Finally, there is a command (inside the authenticated, encrypted channel between client and secstore) to change passwords by sending a new H; for consistency, the client process must at the same time fetch and re-encrypt all files.

When factotum starts, it dials the local secstore and checks whether the user has an account. If so, it prompts for the users secstore password and fetches the key file. The PAK protocol ensures mutual authentication and prevents dictionary attacks on the password by passive wiretappers or active intermediaries. Passwords saved in the key file can be long random strings suitable for simpler challenge/response authentication protocols. Thus the user need only remember a single, weaker password to enable strong, single sign on authentication to unchanged legacy applications scattered across multiple authentication domains.

6. Transport Layer Security

Since the Plan 9 operating system is designed for use in network elements that must withstand direct attack, unguarded by firewall or VPN, we seek to ensure that all applications use channels with appropriate mutual authentication and encryption. A principal tool for this is TLS 1.0 [RFC2246]. (TLS 1.0 is nearly the same as SSL 3.0, and our software is designed to interoperate with implementations of either standard.)

TLS defines a record layer protocol for message integrity and privacy through the use of message digesting and encryption with shared secrets. We implement this service as a kernel device, though it could be performed at slightly higher cost by invoking a separate program. The library interface to the TLS kernel device is:

```
int pushtls(int fd, char *hashalg,
     char *cryptalg, int isclient,
     char *secret, char *dir);
```

Given a file descriptor, the names of message digest and encryption algorithms, and the shared secret, pushtls returns a new file descriptor for the encrypted connection. (The final argument dir receives the name of the directory in the TLS device that is associated with the new connection.) The function is named by analogy with the push operation supported by the stream I/O system of Research Unix and the first two editions of Plan 9. Because adding encryption is as simple as replacing one file descriptor with another, adding encryption to a particular network service is usually trivial.

The Plan 9 shared key authentication protocols establish a shared 56-bit secret as a side effect. Native Plan 9 network services such as cpu and exportfs use these protocols for authentication and then invoke pushtls with the shared secret.

Above the record layer, TLS specifies a handshake protocol using public keys to establish the session secret. This protocol is widely used with HTTP and IMAP4 to provide server authentication, though with client certificates it could provide mutual authentication. The library function

```
int tlsClient(int fd, TLSconn *conn)
```

handles the initial handshake and returns the result of pushtls. On return, it fills the conn structure with the session ID used and the X.509 certificate presented by the server, but makes no effort to verify the certificate. Although the original design intent of X.509 certificates expected that they would be used with a Public Key Infrastructure, reliable deployment has been so delayed and problematic that we have adopted the simpler policy of just using the X.509 certificate as a representation of the public key, depending on a locally-administered directory of SHA1 thumbprints to allow applications to decide which public keys to trust for which purposes.

7. Related Work and Discussion

Kerberos, one of the earliest distributed authentication systems, keeps a set of authentication tickets in a temporary file called a ticket cache. The ticket cache is protected by Unix file permissions. An environment variable containing the file name of the ticket cache allows for different ticket caches in different simultaneous login sessions. A user logs in by typing his or her Kerberos password. The login program uses the Kerberos password to obtain a temporary ticket-granting ticket from the authentication server, initializes the ticket cache with the ticket-granting ticket, and then forgets the password. Other applications can use the ticket-granting ticket to sign tickets for themselves on behalf of the user during the login session. The ticket cache is removed when the user logs out [Stei88]. The ticket cache relieves the user from typing a password every time authentication is needed.

The secure shell SSH develops this idea further, replacing the temporary file with a named Unix domain socket connected to a user-level program, called an agent. Once the SSH agent is started and initialized with one or more RSA private keys, SSH clients can employ it to perform RSA authentications on their behalf. In the absence of an agent, SSH typically uses RSA keys read from encrypted disk files or uses passphrase-based authentication, both of which would require prompting the user for a passphrase whenever authentication is needed [Ylon96]. The self-certifying file system SFS uses a similar agent [Kami00], not only for moderating the use of client authentication keys but also for verifying server public keys [Mazi99].

Factotum is a logical continuation of this evolution, replacing the program-specific SSH or SFS agents with a general agent capable of serving a wide variety of programs. Having one agent for all programs removes the need to have one agent for each program. It also allows the programs themselves to be protocol-agnostic, so that, for example, one could build an SSH workalike capable of using any protocol supported by factotum, without that program knowing anything about the protocols. Traditionally each program needs to implement each authentication protocol for itself, an $O(n^2)$ coding problem that factotum reduces to $O(n)$.

Previous work on agents has concentrated on their use by clients authenticating to servers. Looking in the other direction, Sun Microsystems' pluggable authentication module (PAM) is one of the earliest attempts to provide a general authentication mechanism for Unix-like operating systems [Sama96]. Without a central authority like PAM, system policy is tied up in the various implementations of network services. For example, on a typical Unix, if a system administrator decides not to allow plaintext passwords for authentication, the configuration files for a half dozen different servers rlogind, telnetd, ftpd, sshd, and so on need to be edited. PAM solves this problem by hiding the details of a given authentication mechanism behind a common library interface. Directed by a system-wide configuration file, an application selects a particular authentication mechanism by dynamically loading the appropriate shared library. PAM is widely used on Sun's Solaris and some Linux distributions.

Factotum achieves the same goals using the agent approach. Factotum is the only process that needs to create capabilities, so all the network servers can run as untrusted users (e.g., Plan 9's none or Unix's nobody), which greatly reduces the harm done if a server is buggy and is compromised. In fact, if factotum were implemented on Unix along with an analogue to the Plan 9 capability device, venerable programs like su and login would no longer need to be installed setuid root.

Several other systems, such as Password Safe [Schn], store multiple passwords in an encrypted file, so that the user only needs to remember one password. Our secstore solution differs from these by placing the storage in a hardened location in the network, so that the encrypted file is less liable to be stolen for offline dictionary attack and so that it is available even when a user has several computers. In contrast, Microsoft's Passport system [Micr] keeps credentials in the network, but centralized at one extremely-high-value target. The important feature of Passport, setting up trust relationships with e-merchants, is outside our scope. The secstore architecture is almost identical to Perlman and Kaufman's [Perl99] but with newer EKE technology. Like them, we chose to defend mainly against outside attacks on secstore; if additional defense of the files on the server itself is desired, one can use distributed techniques [Ford00].

We made a conscious choice of placing encryption, message integrity, and key management at the application layer (TLS, just above layer 4) rather than at layer 3, as in IPsec. This leads to a simpler structure for the network stack, easier integration with applications and, most important, easier network administration since we can recognize which applications are misbehaving based on TCP port numbers. TLS does suffer (relative to IPsec) from the possibility of forged TCP Reset, but we feel that this is adequately dealt with by randomized TCP sequence numbers. In contrast with other TLS libraries, Plan 9 does not require the application to change write calls to sslwrite but simply to add a few lines of code at startup [Resc01].

8. Conclusion

Writing safe code is difficult. Stack attacks, mistakes in logic, and bugs in compilers and operating systems can each make it possible for an attacker to subvert the intended execution sequence of a service. If the server process has the privileges of a powerful user, such as root on Unix, then so does the attacker. Factotum allows us to constrain the privileged execution to a single process whose core is a few thousand lines of code. Verifying such a process, both through manual and automatic means, is much easier and less error prone than requiring it of all servers.

An implementation of these ideas is in Plan 9 from Bell Labs, Fourth Edition, freely available from http://plan9.bell-labs.com/plan9 .

Acknowledgments

William Josephson contributed to the implementation of password changing in secstore . We thank Phil MacKenzie and Martín Abadi for helpful comments on early parts of the design. Chuck Blake, Peter Bosch, Frans Kaashoek, Sape Mullender, and Lakshman Y. N., predominantly Dutchmen, gave helpful comments on the paper. Russ Cox is supported by a fellowship from the Fannie and John Hertz Foundation.

References

[Bell93] S. M. Bellovin and M. Merritt, Augmented Encrypted Key Exchange, Proceedings of the 1st ACM Conference on Computer and Communications Security, 1993, pp. 244 - 250.

[Boyk00] Victor Boyko, Philip MacKenzie, and Sarvar Patel, Provably Secure Password-Authenticated Key Exchange using Diffie-Hellman, Eurocrypt 2000, 156-171.

[RFC2246] T. Dierks and C. Allen, The TLS Protocol, Version 1.0, RFC 2246.

[Ford00] Warwick Ford and Burt S. Kaliski, Jr., Server-Assisted Generation of a Strong Secret from a Password, IEEE Fifth International Workshop on Enterprise Security, National Institute of Standards and Technology (NIST), Gaithersburg MD, June 14 - 16, 2000.

[Jabl] David P. Jablon, Strong Password-Only Authenticated Key Exchange, http://integritysciences.com/speke97.html .

[Kami00] Michael Kaminsky, Flexible Key Management with SFS Agents, Masters Thesis, MIT, May 2000.

[Mack] Philip MacKenzie, private communication.

[Mazi99] David Mazières, Michael Kaminsky, M. Frans Kaashoek and Emmett Witchel, Separating key management from file system security, Symposium on Operating Systems Principles, 1999, pp. 124-139.

[Micr] Microsoft Passport, http://www.passport.com/ .

[Perl99] Radia Perlman and Charlie Kaufman, Secure Password-Based Protocol for Downloading a Private Key, Proc. 1999 Network and Distributed System Security Symposium, Internet Society, January 1999.

[Pike95] Rob Pike, Dave Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, and Phil Winterbottom, Plan 9 from Bell Labs, Computing Systems, **8**, 3, Summer 1995, pp. 221-254.

[Pike93] Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, Phil Winterbottom, The Use of Name Spaces in Plan 9, Operating Systems Review, **27**, 2, April 1993, pp.

72-76 (reprinted from Proceedings of the 5th ACM SIGOPS European Workshop, Mont Saint-Michel, 1992, Paper n° 34).

[Resc01] Eric Rescorla, SSL and TLS: Designing and Building Secure Systems, Addison-Wesley, 2001. ISBN 0-201-61598-3, p. 387.

[RFC2138] C. Rigney, A. Rubens, W. Simpson, S. Willens, Remote Authentication Dial In User Service (RADIUS), RFC2138, April 1997.

[RiLa] Ronald L. Rivest and Butler Lampson, SDSI A Simple Distributed Security Infrastructure, http://theory.lcs.mit.edu/~rivest/sdsi10.ps .

[Schn] Bruce Schneier, Password Safe, http://www.counterpane.com/-passsafe.html .

[Sama96] Vipin Samar, Unified Login with Pluggable Authentication Modules (PAM), Proceedings of the Third ACM Conference on Computer Communications and Security, March 1996, New Delhi, India.

[Stei88] Jennifer G. Steiner, Clifford Neumann, and Jeffrey I. Schiller, *Kerberos*: An Authentication Service for Open Network Systems, Proceedings of USENIX Winter Conference, Dallas, Texas, February 1988, pp. 191-202.

[Wu98] T. Wu, The Secure Remote Password Protocol, Proceedings of the 1998 Internet Society Network and Distributed System Security Symposium, San Diego, CA, March 1998, pp. 97-111.

[Ylon96] Ylonen, T., SSH Secure Login Connections Over the Internet, 6th USENIX Security Symposium, pp. 37-42. San Jose, CA, July 1996.

Appendix: Summary of the PAK protocol

Let $q > 2^{160}$ and $p > 2^{1024}$ be primes such that $p = rq + 1$ with r not a multiple of q. Take $h \in Z_p^*$ such that $g = h^r$ is not 1. These parameters may be chosen by the NIST algorithm for DSA, and are public, fixed values. The client C knows a secret π and computes $H = (H_1(C, \pi))^r$ and H^{-1}, where H_1 is a hash function yielding a random element of Z_p^*, and H^{-1} may be computed by gcd. (All arithmetic is modulo p.) The client gives H^{-1} to the server S ahead of time by a private channel. To start a new connection, the client generates a random value x, computes $m = g^x H$, then calls the server and sends C and m. The server checks $m \neq 0 \bmod p$, generates random y, computes $\mu = g^y$, $\sigma = (mH^{-1})^y$, and sends S, μ, $k = sha1(\text{"server"}, C, S, m, \mu, \sigma, H^{-1})$. Next the client computes $\sigma = \mu^x$, verifies k, and sends $k' = sha1(\text{"client"}, C, S, m, \mu, \sigma, H^{-1})$. The server then verifies k' and both sides begin using session key $K = sha1(\text{"session"}, C, S, m, \mu, \sigma, H^{-1})$. In the published version of PAK, the server name S is included in the initial hash H, but doing so is inconvenient in our application, as the server may be known by various equivalent names.

MacKenzie has shown [Mack] that the equivalence proof [Boyk00] can be adapted to cover our version.

How to Use the Plan 9 C Compiler*

Rob Pike
rob@plan9.bell-labs.com

Introduction

The C compiler on Plan 9 is a wholly new program; in fact it was the first piece of software written for what would eventually become Plan 9 from Bell Labs. Programmers familiar with existing C compilers will find a number of differences in both the language the Plan 9 compiler accepts and in how the compiler is used.

The compiler is really a set of compilers, one for each architecture MIPS, SPARC, Intel 386, Power PC, ARM, etc. that accept a dialect of ANSI C and efficiently produce fairly good code for the target machine. There is a packaging of the compiler that accepts strict ANSI C for a POSIX environment, but this document focuses on the native Plan 9 environment, that in which all the system source and almost all the utilities are written.

Source

The language accepted by the compilers is the core 1989 ANSI C language with some modest extensions, a greatly simplified preprocessor, a smaller library that includes system calls and related facilities, and a completely different structure for include files.

Official ANSI C accepts the old (K&R) style of declarations for functions; the Plan 9 compilers are more demanding. Without an explicit run-time flag (-B) whose use is discouraged, the compiler insists on new-style function declarations, that is, prototypes for function arguments. The function declarations in the libraries' include files are all in the new style so the interfaces are checked at compile time. For C programmers who have not yet switched to function prototypes the clumsy syntax may seem repellent but the payoff in stronger typing is substantial. Those who wish to import existing software to Plan 9 are urged to use the opportunity to update their code.

The compilers include an integrated preprocessor that accepts the familiar #include, #define for macros both with and without arguments, #undef, #line, #ifdef, #ifndef, and #endif. It supports neither #if nor ##, although it does honor a few #pragmas. The #if directive was omitted because it greatly complicates the preprocessor, is never necessary, and is usually abused. Conditional compilation in general makes code hard to understand; the Plan 9 source uses it sparingly. Also, because the compilers remove dead code, regular if statements with constant conditions are more readable equivalents to many #ifs. To compile imported code ineluctably fouled by #if there is a separate command, /bin/cpp, that implements the complete ANSI C preprocessor specification.

Include files fall into two groups: machine-dependent and machine-independent. The machine-independent files occupy the directory /sys/include; the others are placed in a directory appropriate to the machine, such as /mips/include. The

* This paper has been revised to reflect the move to 21-bit Unicode.

compiler searches for include files first in the machine-dependent directory and then in the machine-independent directory. At the time of writing there are thirty-one machine-independent include files and two (per machine) machine-dependent ones: <ureg.h> and <u.h>. The first describes the layout of registers on the system stack, for use by the debugger. The second defines some architecture-dependent types such as jmp_buf for setjmp and the va_arg and va_list macros for handling arguments to variadic functions, as well as a set of typedef abbreviations for unsigned short and so on.

Here is an excerpt from /386/include/u.h:

```
#define   nil               ((void*)0)
typedef   unsigned   short   ushort;
typedef   unsigned   char    uchar;
typedef   unsigned   long    ulong;
typedef   unsigned   int     uint;
typedef     signed   char    schar;
typedef   long  long         vlong;

typedef   long      jmp_buf[2];
#define   JMPBUFSP          0
#define   JMPBUFPC          1
#define   JMPBUFDPC         0
```

Plan 9 programs use nil for the name of the zero-valued pointer. The type vlong is the largest integer type available; on most architectures it is a 64-bit value. A couple of other types in <u.h> are u32int, which is guaranteed to have exactly 32 bits (a possibility on all the supported architectures) and mpdigit, which is used by the multiprecision math package <mp.h>. The #define constants permit an architecture-independent (but compiler-dependent) implementation of stack-switching using setjmp and longjmp.

Every Plan 9 C program begins

```
#include   <u.h>
```

because all the other installed header files use the typedefs declared in <u.h>.

In strict ANSI C, include files are grouped to collect related functions in a single file: one for string functions, one for memory functions, one for I/O, and none for system calls. Each include file is protected by an #ifdef to guarantee its contents are seen by the compiler only once. Plan 9 takes a different approach. Other than a few include files that define external formats such as archives, the files in /sys/include correspond to *libraries.* If a program is using a library, it includes the corresponding header. The default C library comprises string functions, memory functions, and so on, largely as in ANSI C, some formatted I/O routines, plus all the system calls and related functions. To use these functions, one must #include the file <libc.h>, which in turn must follow <u.h>, to define their prototypes for the compiler. Here is the complete source to the traditional first C program:

```
#include   <u.h>
#include   <libc.h>

void
main(void)
{
         print("hello   world\n");
         exits(0);
}
```

The print routine and its relatives fprint and sprint resemble the similarly-named functions in Standard I/O but are not attached to a specific I/O library. In Plan 9 `main` is not integer-valued; it should call `exits`, which takes a string argument (or null; here ANSI C promotes the 0 to a `char*`). All these functions are, of course, documented in the Programmer's Manual.

To use `printf`, `<stdio.h>` must be included to define the function prototype for `printf`:

```
#include  <u.h>
#include  <libc.h>
#include  <stdio.h>

void
main(int  argc,  char  *argv[])
{
        printf("%s:  hello  world;  argc  =  %d\n",  argv[0],  argc);
        exits(0);
}
```

In practice, Standard I/O is not used much in Plan 9. I/O libraries are discussed in a later section of this document.

There are libraries for handling regular expressions, raster graphics, windows, and so on, and each has an associated include file. The manual for each library states which include files are needed. The files are not protected against multiple inclusion and themselves contain no nested `#includes`. Instead the programmer is expected to sort out the requirements and to `#include` the necessary files once at the top of each source file. In practice this is trivial: this way of handling include files is so straightforward that it is rare for a source file to contain more than half a dozen `#includes`.

The compilers do their own register allocation so the `register` keyword is ignored. For different reasons, `volatile` and `const` are also ignored.

To make it easier to share code with other systems, Plan 9 has a version of the compiler, `pcc`, that provides the standard ANSI C preprocessor, headers, and libraries with POSIX extensions. `Pcc` is recommended only when broad external portability is mandated. It compiles slower, produces slower code (it takes extra work to simulate POSIX on Plan 9), eliminates those parts of the Plan 9 interface not related to POSIX, and illustrates the clumsiness of an environment designed by committee. `Pcc` is described in more detail in *APE—The ANSI/POSIX Environment*, by Howard Trickey.

Process

Each CPU architecture supported by Plan 9 is identified by a single, arbitrary, alphanumeric character: `k` for SPARC, `q` for 32-bit Power PC, `v` for MIPS, `0` for little-endian MIPS, `5` for ARM v5 and later 32-bit architectures, `6` for AMD64, `8` for Intel 386, and `9` for 64-bit Power PC. The character labels the support tools and files for that architecture. For instance, for the 386 the compiler is `8c`, the assembler is `8a`, the link editor/loader is `8l`, the object files are suffixed `.8`, and the default name for an executable file is `8.out`. Before we can use the compiler we therefore need to know which machine we are compiling for. The next section explains how this decision is made; for the moment assume we are building 386 binaries and make the mental substitution for `8` appropriate to the machine you are actually using.

To convert source to an executable binary is a two-step process. First run the compiler, `8c`, on the source, say `file.c`, to generate an object file `file.8`. Then run the loader, `8l`, to generate an executable `8.out` that may be run (on a 386 machine):

```
8c file.c
8l file.8
8.out
```

The loader automatically links with whatever libraries the program needs, usually including the standard C library as defined by <libc.h>. Of course the compiler and loader have lots of options, both familiar and new; see the manual for details. The compiler does not generate an executable automatically; the output of the compiler must be given to the loader. Since most compilation is done under the control of mk (see below), this is rarely an inconvenience.

The distribution of work between the compiler and loader is unusual. The compiler integrates preprocessing, parsing, register allocation, code generation and some assembly. Combining these tasks in a single program is part of the reason for the compilers efficiency. The loader does instruction selection, branch folding, instruction scheduling, and writes the final executable. There is no separate C preprocessor and no assembler in the usual pipeline. Instead the intermediate object file (here a .8 file) is a type of binary assembly language. The instructions in the intermediate format are not exactly those in the machine. For example, on the 68020 the object file may specify a MOVE instruction but the loader will decide just which variant of the MOVE instruction MOVE immediate, MOVE quick, MOVE address, etc. is most efficient.

The assembler, 8a, is just a translator between the textual and binary representations of the object file format. It is not an assembler in the traditional sense. It has limited macro capabilities (the same as the integral C preprocessor in the compiler), clumsy syntax, and minimal error checking. For instance, the assembler will accept an instruction (such as memory-to-memory MOVE on the MIPS) that the machine does not actually support; only when the output of the assembler is passed to the loader will the error be discovered. The assembler is intended only for writing things that need access to instructions invisible from C, such as the machine-dependent part of an operating system; very little code in Plan 9 is in assembly language.

The compilers take an option -S that causes them to print on their standard output the generated code in a format acceptable as input to the assemblers. This is of course merely a formatting of the data in the object file; therefore the assembler is just an ASCII-to-binary converter for this format. Other than the specific instructions, the input to the assemblers is largely architecture-independent; see A Manual for the Plan 9 Assembler, by Rob Pike, for more information.

The loader is an integral part of the compilation process. Each library header file contains a #pragma that tells the loader the name of the associated archive; it is not necessary to tell the loader which libraries a program uses. The C run-time startup is found, by default, in the C library. The loader starts with an undefined symbol, _main, that is resolved by pulling in the run-time startup code from the library. (The loader undefines _mainp when profiling is enabled, to force loading of the profiling start-up instead.)

Unlike its counterpart on other systems, the Plan 9 loader rearranges data to optimize access. This means the order of variables in the loaded program is unrelated to its order in the source. Most programs don t care, but some assume that, for example, the variables declared by

```
int a;
int b;
```

will appear at adjacent addresses in memory. On Plan 9, they won

Heterogeneity

When the system starts or a user logs in the environment is configured so the appropriate binaries are available in /bin. The configuration process is controlled by an environment variable, $cputype, with values such as mips, 386, arm, or sparc. For each architecture there is a directory in the root, with the appropriate name, that holds the binary and library files for that architecture. Thus /mips/lib contains the object code libraries for MIPS programs, /mips/include holds MIPS-specific include files, and /mips/bin has the MIPS binaries. These binaries are attached to /bin at boot time by binding /$cputype/bin to /bin, so /bin always contains the correct files.

The MIPS compiler, vc, by definition produces object files for the MIPS architecture, regardless of the architecture of the machine on which the compiler is running. There is a version of vc compiled for each architecture: /mips/bin/vc, /arm/bin/vc, /sparc/bin/vc, and so on, each capable of producing MIPS object files regardless of the native instruction set. If one is running on a SPARC, /sparc/bin/vc will compile programs for the MIPS; if one is running on machine $cputype, /$cputype/bin/vc will compile programs for the MIPS.

Because of the bindings that assemble /bin, the shell always looks for a command, say date, in /bin and automatically finds the file /$cputype/bin/date. Therefore the MIPS compiler is known as just vc; the shell will invoke /bin/vc and that is guaranteed to be the version of the MIPS compiler appropriate for the machine running the command. Regardless of the architecture of the compiling machine, /bin/vc is *always* the MIPS compiler.

Also, the output of vc and vl is completely independent of the machine type on which they are executed: .v files compiled (with vc) on a SPARC may be linked (with vl) on a 386. (The resulting v.out will run, of course, only on a MIPS.) Similarly, the MIPS libraries in /mips/lib are suitable for loading with vl on any machine; there is only one set of MIPS libraries, not one set for each architecture that supports the MIPS compiler.

Heterogeneity and mk

Most software on Plan 9 is compiled under the control of mk, a descendant of make that is documented in the Programmer's Manual. A convention used throughout the mkfiles makes it easy to compile the source into binary suitable for any architecture.

The variable $cputype is advisory: it reports the architecture of the current environment, and should not be modified. A second variable, $objtype, is used to set which architecture is being *compiled* for. The value of $objtype can be used by a mkfile to configure the compilation environment.

In each machine's root directory there is a short mkfile that defines a set of macros for the compiler, loader, etc. Here is /mips/mkfile:

```
</sys/src/mkfile.proto

CC=vc
LD=vl
O=v
AS=va
```

The line

```
</sys/src/mkfile.proto
```

causes mk to include the file /sys/src/mkfile.proto, which contains general definitions:

```
#
# common mkfile parameters shared by all architectures
#

OS=5689qv
CPUS=arm amd64 386 power mips
CFLAGS=-FTVw
LEX=lex
YACC=yacc
MK=/bin/mk
```

CC is obviously the compiler, AS the assembler, and LD the loader. O is the suffix for the object files and CPUS and OS are used in special rules described below.

Here is a mkfile to build the installed source for sam:

```
</$objtype/mkfile
OBJ=sam.$O address.$O buffer.$O cmd.$O disc.$O error.$O \
        file.$O io.$O list.$O mesg.$O moveto.$O multi.$O \
        plan9.$O rasp.$O regexp.$O string.$O sys.$O xec.$O

$O.out: $OBJ
        $LD $OBJ

install:        $O.out
        cp $O.out /$objtype/bin/sam

installall:
        for(objtype in $CPUS) mk install

%.$O:   %.c
        $CC $CFLAGS $stem.c

$OBJ:   sam.h errors.h mesg.h
address.$O cmd.$O parse.$O xec.$O unix.$O:      parse.h

clean:V:
        rm -f [$OS].out *.[$OS] y.tab.?
```

(The actual mkfile imports most of its rules from other secondary files, but this example works and is not misleading.) The first line causes mk to include the contents of /$objtype/mkfile in the current mkfile. If $objtype is mips, this inserts the MIPS macro definitions into the mkfile. In this case the rule for $O.out uses the MIPS tools to build v.out. The %.$O rule in the file uses mk's pattern matching facilities to convert the source files to the object files through the compiler. (The text of the rules is passed directly to the shell, rc, without further translation. See the mk manual if any of this is unfamiliar.) Because the default rule builds $O.out rather than sam, it is possible to maintain binaries for multiple machines in the same source directory without conflict. This is also, of course, why the output files from the various compilers and loaders have distinct names.

The rest of the mkfile should be easy to follow; notice how the rules for clean and installall (that is, install versions for all architectures) use other macros defined in /$objtype/mkfile. In Plan 9, mkfiles for commands conventionally contain rules to install (compile and install the version for $objtype), installall (compile and install for all $objtypes), and clean (remove all object files, binaries, etc.).

The mkfile is easy to use. To build a MIPS binary, v.out:

```
% objtype=mips
% mk
```

To build and install a MIPS binary:

```
% objtype=mips
% mk install
```

To build and install all versions:

```
% mk installall
```

These conventions make cross-compilation as easy to manage as traditional native compilation. Plan 9 programs compile and run without change on machines from large multiprocessors to laptops. For more information about this process, see Plan 9 Mkfiles, by Bob Flandrena.

Portability

Within Plan 9, it is painless to write portable programs, programs whose source is independent of the machine on which they execute. The operating system is fixed and the compiler, headers and libraries are constant so most of the stumbling blocks to portability are removed. Attention to a few details can avoid those that remain.

Plan 9 is a heterogeneous environment, so programs must *expect* that external files will be written by programs on machines of different architectures. The compilers, for instance, must handle without confusion object files written by other machines. The traditional approach to this problem is to pepper the source with #ifdefs to turn byte-swapping on and off. Plan 9 takes a different approach: of the handful of machine-dependent #ifdefs in all the source, almost all are deep in the libraries. Instead programs read and write files in a defined format, either (for low volume applications) as formatted text, or (for high volume applications) as binary in a known byte order. If the external data were written with the most significant byte first, the following code reads a 4-byte integer correctly regardless of the architecture of the executing machine (assuming an unsigned long holds 4 bytes):

```
ulong
getlong(void)
{
        ulong l;

        l = (getchar()&0xFF)<<24;
        l |= (getchar()&0xFF)<<16;
        l |= (getchar()&0xFF)<<8;
        l |= (getchar()&0xFF)<<0;
        return l;
}
```

Note that this code does not swap the bytes; instead it just reads them in the correct order. Variations of this code will handle any binary format and also avoid problems involving how structures are padded, how words are aligned, and other impediments to portability. Be aware, though, that extra care is needed to handle floating point data.

Efficiency hounds will argue that this method is unnecessarily slow and clumsy when the executing machine has the same byte order (and padding and alignment) as the data. The CPU cost of I/O processing is rarely the bottleneck for an application, however, and the gain in simplicity of porting and maintaining the code greatly outweighs the minor speed loss from handling data in this general way. This method is how the Plan 9 compilers, the window system, and even the file servers transmit data between programs.

To port programs beyond Plan 9, where the system interface is more variable, it is probably necessary to use pcc and hope that the target machine supports ANSI C and POSIX.

I/O

The default C library, defined by the include file <libc.h>, contains no buffered I/O package. It does have several entry points for printing formatted text: print outputs text to the standard output, fprint outputs text to a specified integer file descriptor, and sprint places text in a character array. To access library routines for buffered I/O, a program must explicitly include the header file associated with an appropriate library.

The recommended I/O library, used by most Plan 9 utilities, is bio (buffered I/O), defined by <bio.h>. There also exists an implementation of ANSI Standard I/O, stdio.

Bio is small and efficient, particularly for buffer-at-a-time or line-at-a-time I/O. Even for character-at-a-time I/O, however, it is significantly faster than the Standard I/O library, stdio. Its interface is compact and regular, although it lacks a few conveniences. The most noticeable is that one must explicitly define buffers for standard input and output; bio does not predefine them. Here is a program to copy input to output a byte at a time using bio:

```
#include  <u.h>
#include  <libc.h>
#include  <bio.h>

Biobuf   bin;
Biobuf   bout;

main(void)
{
        int  c;

        Binit(&bin,  0, OREAD);
        Binit(&bout, 1, OWRITE);

        while((c=Bgetc(&bin)) != Beof)
                Bputc(&bout, c);
        exits(0);
}
```

For peak performance, we could replace Bgetc and Bputc by their equivalent in-line macros BGETC and BPUTC but the performance gain would be modest. For more information on bio, see the Programmer's Manual.

Perhaps the most dramatic difference in the I/O interface of Plan 9 from other systems is that text is not ASCII. The format for text in Plan 9 is a byte-stream encoding of 21-bit characters. The character set is based on the Unicode Standard and is backward compatible with ASCII: characters with value 0 through 127 are the same in both sets. The 21-bit characters, called *runes* in Plan 9, are encoded using a representation called UTF, an encoding that is becoming accepted as a standard. (ISO calls it UTF-8; throughout Plan 9 it's just called UTF.) UTF defines multibyte sequences to represent character values from 0 to 1,114,111. In UTF, character values up to 127 decimal, 7F hexadecimal, represent themselves, so straight ASCII files are also valid UTF. Also, UTF guarantees that bytes with values 0 to 127 (NUL to DEL, inclusive) will appear only when they represent themselves, so programs that read bytes looking for plain ASCII characters will continue to work. Any program that expects a one-to-one correspondence between bytes and characters will, however, need to be modified. An example is parsing file names. File names, like all text, are in UTF, so it is incorrect to search for a character in a string by strchr(filename, c) because the character might have a multi-byte encoding. The correct method is to call utfrune(filename, c), defined in *rune*(2), which interprets the file name as a sequence of encoded characters rather than bytes. In fact, even when you know the character is a single byte that can represent only itself, it is safer to use utfrune because that assumes nothing about the character set and its representation.

The library defines several symbols relevant to the representation of characters. Any byte with unsigned value less than Runesync will not appear in any multi-byte encoding of a character. Utfrune compares the character being searched against Runesync to see if it is sufficient to call strchr or if the byte stream must be interpreted. Any byte with unsigned value less than Runeself is represented by a single byte with the same value. Finally, when errors are encountered converting to runes from a byte stream, the library returns the rune value Runeerror and advances a single byte. This permits programs to find runes embedded in binary data.

Bio includes routines Bgetrune and Bputrune to transform the external byte stream UTF format to and from internal 21-bit runes. Also, the %s format to print accepts UTF; %c prints a character after narrowing it to 8 bits. The %S format prints a null-terminated sequence of runes; %C prints a character after narrowing it to 21 bits. For more information, see the Programmer s Manual, in particular *utf*(6) and *rune*(2), and the paper, Hello world, or ˚±»·¼–Á± °ÌÃ¼µ, or こんにちは世界, by Rob Pike and Ken Thompson; there is not room for the full story here.

These issues affect the compiler in several ways. First, the C source is in UTF. ANSI says C variables are formed from ASCII alphanumerics, but comments and literal strings may contain any characters encoded in the native encoding, here UTF. The declaration

```
char *cp = "abcÿ";
```

initializes the variable cp to point to an array of bytes holding the UTF representation of the characters abcÿ. The type Rune is defined in <u.h> to be ushort, which is also the wide character type in the compiler. Therefore the declaration

```
Rune *rp = L"abcÿ";
```

initializes the variable rp to point to an array of unsigned long integers holding the 21-bit values of the characters abcÿ. Note that in both these declarations the characters in the source that represent abcÿ are the same; what changes is how those characters are represented in memory in the program. The following two lines:

```
print("%s\n", "abcÿ");
print("%S\n", L"abcÿ");
```

produce the same UTF string on their output, the first by copying the bytes, the second by converting from runes to bytes.

In C, character constants are integers but narrowed through the char type. The Unicode character ÿ has value 255, so if the char type is signed, the constant 'ÿ' has value 1 (which is equal to EOF). On the other hand, L'ÿ' narrows through the wide character type, ushort, and therefore has value 255.

Finally, although it s not ANSI C, the Plan 9 C compilers assume any character with value above Runeself is an alphanumeric, so ± is a legal, if non-portable, variable name.

Arguments

Some macros are defined in <libc.h> for parsing the arguments to main(). They are described in *ARG*(2) but are fairly self-explanatory. There are four macros: ARGBEGIN and ARGEND are used to bracket a hidden switch statement within which ARGC returns the current option character (rune) being processed and ARGF returns the argument to the option, as in the loader option -o file. Here, for example, is the code at the beginning of main() in ramfs.c (see *ramfs*(1)) that cracks its arguments:

```
void
main(int argc, char *argv[])
{
	char *defmnt;
	int p[2];
	int mfd[2];
	int stdio = 0;

	defmnt = "/tmp";
	ARGBEGIN{
	case 'i':
		defmnt = 0;
		stdio = 1;
		mfd[0] = 0;
		mfd[1] = 1;
		break;
	case 's':
		defmnt = 0;
		break;
	case 'm':
		defmnt = ARGF();
		break;
	default:
		usage();
	}ARGEND
```

Extensions

The compiler has several extensions to 1989 ANSI C, all of which are used extensively in the system source. Some of these have been adopted in later ANSI C standards. First, *structure displays* permit struct expressions to be formed dynamically. Given these declarations:

```
typedef struct Point Point;
typedef struct Rectangle Rectangle;

struct Point
{
	int x, y;
};

struct Rectangle
{
	Point min, max;
};

Point     p, q, add(Point, Point);
Rectangle r;
int       x, y;
```

this assignment may appear anywhere an assignment is legal:

```
r = (Rectangle){add(p, q), (Point){x, y+3}};
```

The syntax is the same as for initializing a structure but with a leading cast.

If an *anonymous structure* or *union* is declared within another structure or union, the members of the internal structure or union are addressable without prefix in the outer structure. This feature eliminates the clumsy naming of nested structures and, particularly, unions. For example, after these declarations,

```
struct Lock
{
        int     locked;
};

struct Node
{
        int     type;
        union{
                double  dval;
                double  fval;
                long    lval;
        };              /* anonymous union */
        struct Lock;    /* anonymous structure */
} *node;

void    lock(struct Lock*);
```

one may refer to node->type, node->dval, node->fval, node->lval, and node->locked. Moreover, the address of a struct Node may be used without a cast anywhere that the address of a struct Lock is used, such as in argument lists. The compiler automatically promotes the type and adjusts the address. Thus one may invoke lock(node).

Anonymous structures and unions may be accessed by type name if (and only if) they are declared using a typedef name. For example, using the above declaration for Point, one may declare

```
struct
{
        int     type;
        Point;
} p;
```

and refer to p.Point.

In the initialization of arrays, a number in square brackets before an element sets the index for the initialization. For example, to initialize some elements in a table of function pointers indexed by ASCII character,

```
void    percent(void), slash(void);

void    (*func[128])(void) =
{
        ['%']   percent,
        ['/']   slash,
};
```

A similar syntax allows one to initialize structure elements:

```
Point p =
{
        .y 100,
        .x 200
};
```

These initialization syntaxes were later added to ANSI C, with the addition of an equals sign between the index or tag and the value. The Plan 9 compiler accepts either form.

Finally, the declaration

```
extern register reg;
```

(*this* appearance of the register keyword is not ignored) allocates a global register to hold the variable reg. External registers must be used carefully: they need to be declared in *all* source files and libraries in the program to guarantee the register is not allocated temporarily for other purposes. Especially on machines with few registers, such as the i386, it is easy to link accidentally with code that has already usurped the global registers and there is no diagnostic when this happens. Used wisely, though, external registers are powerful. The Plan 9 operating system uses them to access per-process and per-machine data structures on a multiprocessor. The storage class they provide is hard to create in other ways.

The compile-time environment

The code generated by the compilers is optimized by default: variables are placed in registers and peephole optimizations are performed. The compiler flag -N disables these optimizations. Registerization is done locally rather than throughout a function: whether a variable occupies a register or the memory location identified in the symbol table depends on the activity of the variable and may change throughout the life of the variable. The -N flag is rarely needed; its main use is to simplify debugging. There is no information in the symbol table to identify the registerization of a variable, so -N guarantees the variable is always where the symbol table says it is.

Another flag, -w, turns *on* warnings about portability and problems detected in flow analysis. Most code in Plan 9 is compiled with warnings enabled; these warnings plus the type checking offered by function prototypes provide most of the support of the Unix tool lint more accurately and with less chatter. Two of the warnings, used and not set and set and not used, are almost always accurate but may be triggered spuriously by code with invisible control flow, such as in routines that call longjmp. The compiler statements

```
SET(v1);
USED(v2);
```

decorate the flow graph to silence the compiler. Either statement accepts a comma-separated list of variables. Use them carefully: they may silence real errors. For the common case of unused parameters to a function, leaving the name off the declaration silences the warnings. That is, listing the type of a parameter but giving it no associated variable name does the trick.

Debugging

There are two debuggers available on Plan 9. The first, and older, is db, a revision of Unix adb. The other, acid, is a source-level debugger whose commands are statements in a true programming language. Acid is the preferred debugger, but since it borrows some elements of db, notably the formats for displaying values, it is worth knowing a little bit about db.

Both debuggers support multiple architectures in a single program; that is, the programs are db and acid, not for example vdb and vacid. They also support cross-architecture debugging comfortably: one may debug a 386 binary on a MIPS.

Imagine a program has crashed mysteriously:

```
% X11/X
Fatal server bug!
failed to create default stipple
X 106: suicide: sys: trap: fault read addr=0x0 pc=0x00105fb8
%
```

When a process dies on Plan 9 it hangs in the broken state for debugging. Attach a debugger to the process by naming its process id:

```
% acid 106
/proc/106/text:mips plan 9 executable

/sys/lib/acid/port
/sys/lib/acid/mips
acid:
```

The acid function stk() reports the stack traceback:

```
acid: stk()
At pc:0x105fb8:abort+0x24 /sys/src/ape/lib/ap/stdio/abort.c:6
abort() /sys/src/ape/lib/ap/stdio/abort.c:4
        called from FatalError+#4e
                /sys/src/X/mit/server/dix/misc.c:421
FatalError(s9=#e02, s8=#4901d200, s7=#2, s6=#72701, s5=#1,
     s4=#7270d, s3=#6, s2=#12, s1=#ff37f1c, s0=#6, f=#7270f)
     /sys/src/X/mit/server/dix/misc.c:416
        called from gnotscreeninit+#4ce
                /sys/src/X/mit/server/ddx/gnot/gnot.c:792
gnotscreeninit(snum=#0, sc=#80db0)
     /sys/src/X/mit/server/ddx/gnot/gnot.c:766
        called from AddScreen+#16e
                /n/bootes/sys/src/X/mit/server/dix/main.c:610
AddScreen(pfnInit=0x0000129c,argc=0x00000001,argv=0x7fffffe4)
     /sys/src/X/mit/server/dix/main.c:530
        called from InitOutput+0x80
                /sys/src/X/mit/server/ddx/brazil/brddx.c:522
InitOutput(argc=0x00000001,argv=0x7fffffe4)
     /sys/src/X/mit/server/ddx/brazil/brddx.c:511
        called from main+0x294
                /sys/src/X/mit/server/dix/main.c:225
main(argc=0x00000001,argv=0x7fffffe4)
     /sys/src/X/mit/server/dix/main.c:136
        called from _main+0x24
                /sys/src/ape/lib/ap/mips/main9.s:8
```

The function lstk() is similar but also reports the values of local variables. Note that the traceback includes full file names; this is a boon to debugging, although it makes the output much noisier.

To use acid well you will need to learn its input language; see the Acid Manual, by Phil Winterbottom, for details. For simple debugging, however, the information in the manual page is sufficient. In particular, it describes the most useful functions for examining a process.

The compiler does not place information describing the types of variables in the executable, but a compile-time flag provides crude support for symbolic debugging. The -a flag to the compiler suppresses code generation and instead emits source text in the *acid* language to format and display data structure types defined in the program. The easiest way to use this feature is to put a rule in the **mkfile**:

```
syms:    main.$O
         $CC -a main.c > syms
```

Then from within *acid*,

```
acid: include("sourcedirectory/syms")
```

to read in the relevant definitions. (For multi-file source, you need to be a little fancier; see *8c*(1)). This text includes, for each defined compound type, a function with that name that may be called with the address of a structure of that type to display its contents. For example, if *rect* is a global variable of type *Rectangle*, one may execute

```
Rectangle(*rect)
```

to display it. The * (indirection) operator is necessary because of the way *acid* works: each global symbol in the program is defined as a variable by *acid*, with value equal to the *address* of the symbol.

Another common technique is to write by hand special *acid* code to define functions to aid debugging, initialize the debugger, and so on. Conventionally, this is placed in a file called *acid* in the source directory; it has a line

```
include("sourcedirectory/syms");
```

to load the compiler-produced symbols. One may edit the compiler output directly but it is wiser to keep the hand-generated *acid* separate from the machine-generated.

To make things simple, the default rules in the system mkfiles include entries to make *foo.acid* from *foo.c*, so one may use *mk* to automate the production of *acid* definitions for a given C source file.

There is much more to say here. See *acid* manual page, the reference manual, or the paper *Acid: A Debugger Built From A Language*, also by Phil Winterbottom.

Changes to the Programming Environment in the Fourth Release of Plan 9

Rob Pike

rob@plan9.bell-labs.com

Introduction

The fourth release of Plan 9 includes changes at many levels of the system, with repercussions in the libraries and program interfaces. This document summarizes the changes and describes how existing programs must be modified to run in the new release. It is not exhaustive, of course; for further detail about any of the topics refer to the manual pages, as always.

Programmers new to Plan 9 may find valuable tidbits here, but the real audience for this paper is those with a need to update applications and servers written in C for earlier releases of the Plan 9 operating system.

9P, NAMELEN, and strings

The underlying file service protocol for Plan 9, 9P, retains its basic form but has had a number of adjustments to deal with longer file names and error strings, new authentication mechanisms, and to make it more efficient at evaluating file names. The change to file names affects a number of system interfaces; because file name elements are no longer of fixed size, they can no longer be stored as arrays.

9P used to be a fixed-format protocol with NAMELEN-sized byte arrays representing file name elements. Now, it is a variable-format protocol, as described in *intro*(5), in which strings are represented by a count followed by that many bytes. Thus, the string ken would previously have occupied 28 (NAMELEN) bytes in the message; now it occupies 5: a two-byte count followed by the three bytes of ken and no terminal zero. (And of course, a name could now be much longer.) A similar format change has been made to stat buffers: they are no longer DIRLEN bytes long but instead have variable size prefixed by a two-byte count. And in fact the entire 9P message syntax has changed: every message now begins with a message length field that makes it trivial to break the string into messages without parsing them, so aux/fcall is gone. A new library entry point, read9pmsg, makes it easy for user-level servers to break the client data stream into 9P messages. All servers should switch from using read (or the now gone getS) to using read9pmsg.

This change to 9P affects the way strings are handled by the kernel and throughout the system. The consequences are primarily that fixed-size arrays have been replaced by pointers and counts in a variety of system interfaces. Most programs will need at least some adjustment to the new style. In summary: NAMELEN is gone, except as a vestige in the authentication libraries, where it has been rechristened ANAMELEN. DIRLEN and ERRLEN are also gone. All programs that mention these constants will need to be fixed.

The simplest place to see this change is in the errstr system call, which no longer assumes a buffer of length ERRLEN but now requires a byte-count argument:

```
char buf[...];

errstr(buf, sizeof buf);
```

The buffer can be any size you like. For convenience, the kernel stores error strings internally as 256-byte arrays, so if you like — but it's not required — you can use the defined constant ERRMAX= 256 as a good buffer size. Unlike the old ERRLEN (which had value 64), ERRMAX is advisory, not mandatory, and is not part of the 9P specification.

With names, stat buffers, and directories, there isn't even an echo of a fixed-size array any more.

Directories and wait messages

With strings now variable-length, a number of system calls needed to change: errstr, stat, fstat, wstat, fwstat, and wait are all affected, as is read when applied to directories.

As far as directories are concerned, most programs don't use the system calls directly anyway, since they operate on the machine-independent form, but instead call the machine-dependent Dir routines dirstat, dirread, etc. These used to fill user-provided fixed-size buffers; now they return objects allocated by malloc (which must therefore be freed after use). To 'stat' a file:

```
Dir *d;

d = dirstat(filename);
if(d == nil){
	fprint(2, "can't stat %s: %r\n", filename);
	exits("stat");
}
use(d);
free(d);
```

A common new bug is to forget to free the Dir returned by dirstat.

Dirfstat and Dirfwstat work pretty much as before, but changes to 9P make it possible to exercise finer-grained control on what fields of the Dir are to be changed; see *stat*(2) and *stat*(5) for details.

Reading a directory works in a similar way to dirstat, with dirread allocating and filling in an array of Dir structures. The return value is the number of elements of the array. The arguments to dirread now include a pointer to a Dir* to be filled in with the address of the allocated array:

```
Dir *d;
int i, n;

while((n = dirread(fd, &d)) > 0){
	for(i=0; i<n; i++)
		use(&d[i]);
	free(d);
}
```

A new library function, dirreadall, has the same form as dirread but returns the entire directory in one call:

```
n = dirreadall(fd, &d)
for(i=0; i<n; i++)
        use(&d[i]);
free(d);
```

If your program insists on using the underlying stat system call or its relatives, or wants to operate directly on the machine-independent format returned by stat or read, it will need to be modified. Such programs are rare enough that we'll not discuss them here beyond referring to the man page *stat* (2) for details. Be aware, though, that it used to be possible to regard the buffer returned by stat as a byte array that began with the zero-terminated name of the file; this is no longer true. With very rare exceptions, programs that call stat would be better recast to use the dir routines or, if their goal is just to test the existence of a file, access.

Similar changes have affected the wait system call. In fact, wait is no longer a system call but a library routine that calls the new await system call and returns a newly allocated machine-dependent Waitmsg structure:

```
Waitmsg *w;

w = wait();
if(w == nil)
        error("wait: %r");
print("pid is %d; exit string %s\n", w->pid, w->msg);
free(w);
```

The exit string w->msg may be empty but it will never be a nil pointer. Again, don't forget to free the structure returned by wait. If all you need is the pid, you can call waitpid, which reports just the pid and doesn't return an allocated structure:

```
int pid;

pid = waitpid();
if(pid < 0)
        error("wait: %r");
print("pid is %d\n", pid);
```

Quoted strings and tokenize

Wait gives us a good opportunity to describe how the system copes with all this free-format data. Consider the text returned by the await system call, which includes a set of integers (pids and times) and a string (the exit status). This information is formatted free-form; here is the statement in the kernel that generates the message:

```
n = snprint(a, n, "%d %lud %lud %lud %q",
        wq->w.pid,
        wq->w.time[TUser], wq->w.time[TSys], wq->w.time[TReal],
        wq->w.msg);
```

Note the use of %q to produce a quoted-string representation of the exit status. The %q format is like %s but will wrap rc-style single quotes around the string if it contains whitespace or is otherwise ambiguous. The library routine tokenize can be used to parse data formatted this way: it splits white-space-separated fields but understands the %q quoting conventions. Here is how the wait library routine builds its Waitmsg from the data returned by await:

```
Waitmsg*
wait(void)
{
	int n, l;
	char buf[512], *fld[5];
	Waitmsg *w;

	n = await(buf, sizeof buf-1);
	if(n < 0)
		return nil;
	buf[n] = ' ';
	if(tokenize(buf, fld, nelem(fld)) != nelem(fld)){
		werrstr("couldn't parse wait message");
		return nil;
	}
	l = strlen(fld[4])+1;
	w = malloc(sizeof(Waitmsg)+l);
	if(w == nil)
		return nil;
	w->pid = atoi(fld[0]);
	w->time[0] = atoi(fld[1]);
	w->time[1] = atoi(fld[2]);
	w->time[2] = atoi(fld[3]);
	w->msg = (char*)&w[1];
	memmove(w->msg, fld[4], l);
	return w;
}
```

This style of quoted-string and tokenize is used all through the system now. In particular, devices now tokenize the messages written to their ctl files, which means that you can send messages that contain white space, by quoting them, and that you no longer need to worry about whether or not the device accepts a newline. In other words, you can say

```
echo message > /dev/xx/ctl
```

instead of echo -n because tokenize treats the newline character as white space and discards it.

While we're on the subject of quotes and strings, note that the implementation of await used snprint rather than sprint. We now deprecate sprint because it has no protection against buffer overflow. We prefer snprint or seprint, to constrain the output. The %q format is cleverer than most in this regard: if the string is too long to be represented in full, %q is smart enough to produce a truncated but correctly quoted string within the available space.

Mount

Although strings in 9P are now variable-length and not zero-terminated, this has little direct effect in most of the system interfaces. File and user names are still zero-terminated strings as always; the kernel does the work of translating them as necessary for transport. And of course, they are now free to be as long as you might want; the only hard limit is that their length must be represented in 16 bits.

One example where this matters is that the file system specification in the mount system call can now be much longer. Programs like rio that used the specification string in creative ways were limited by the NAMELEN restriction; now they can use the string more freely. Rio now accepts a simple but less cryptic specification language for the window to be created by the mount call, e.g.:

```
% mount $wsys /mnt/wsys 'new -dx 250 -dy 250 -pid 1234'
```

In the old system, this sort of control was impossible through the mount interface.

While we're on the subject of mount, note that with the new security architecture (see *factotum*(4)), 9P has moved its authentication outside the protocol proper. (For a full description of this change to 9P, see *fauth*(2), *attach*(5), and the paper *Security in Plan 9*.) The most explicit effect of this change is that mount now takes another argument, afd, a file descriptor for the authentication file through which the authentication will be made. For most user-level file servers, which do not require authentication, it is sufficient to provide -1 as the value of afd:

```
if(mount(fd, -1, "/mnt/wsys", MREPL,
    "new -dx 250 -dy 250 -pid 1234") < 0)
        error("mount failed: %r");
```

To connect to servers that require authentication, use the new fauth system call or the reimplemented amount (authenticated mount) library call. In fact, since amount handles both authenticating and non-authenticating servers, it is often easiest just to replace calls to mount by calls to amount; see *auth*(2) for details.

Print

The C library has been heavily reworked in places. Besides the changes mentioned above, it now has a much more complete set of routines for handling Rune strings (that is, zero-terminated arrays of 16-bit character values). The most sweeping changes, however, are in the way formatted I/O is performed.

The print routine and all its relatives have been reimplemented to offer a number of improvements:

(1) Better buffer management, including the provision of an internal flush routine, makes it unnecessary to provide large buffers. For example, print uses a much smaller buffer now (reducing stack load) while simultaneously removing the need to truncate the output string if it doesn't fit in the buffer.

(2) Global variables have been eliminated so no locking is necessary.

(3) The combination of (1) and (2) means that the standard implementation of print now works fine in threaded programs, and threadprint is gone.

(4) The new routine smprint prints into, and returns, storage allocated on demand by malloc.

(5) It is now possible to print into a Rune string; for instance, runesmprint is the Rune analog of smprint.

(6) There is improved support for custom print verbs and custom output routines such as error handlers. The routine doprint is gone, but vseprint can always be used instead. However, the new routines fmtfdinit, fmtstrinit, fmtprint, and friends are often a better replacement. The details are too long for exposition here; *fmtinstall*(2) explains the new interface and provides examples.

(7) Two new format flags, space and comma, close somewhat the gap between Plan 9 and ANSI C.

Despite these changes, most programs will be unaffected; print is still print. Don't forget, though, that you should eliminate calls to sprint and use the %q format when appropriate.

Binary compatibility

The discussion so far has been about changes at the source level. Existing binaries will probably run without change in the new environment, since the kernel provides backward-compatible system calls for `errstr`, `stat`, `wait`, etc. The only exceptions are programs that do either a `mount` system call, because of the security changes and because the file descriptor in `mount` must point to a new 9P connection; or a `read` system call on a directory, since the returned data will be in the new format. A moment's reflection will discover that this means old user-level file servers will need to be fixed to run on the new system.

File servers

A full description of what user-level servers must do to provide service with the new 9P is beyond the scope of this paper. Your best source of information is section 5 of the manual, combined with study of a few examples. `/sys/src/cmd/ramfs.c` is a simple example; it has a counterpart `/sys/src/lib9p/ramfs.c` that implements the same service using the new *9p*(2) library.

That said, it's worth summarizing what to watch for when converting a file server. The `session` message is gone, and there is a now a `version` message that is exchanged at the start of a connection to establish the version of the protocol to use (there's only one at the moment, identified by the string `9P2000`) and what the maximum message size will be. This negotiation makes it easier to handle 9P encapsulation, such as with `exportfs`, and also permits larger message sizes when appropriate.

If your server wants to authenticate, it will need to implement an authentication file and implement the `auth` message; otherwise it should return a helpful error string to the `Tauth` request to signal that authentication is not required.

The handling of `stat` and directory reads will require some changes but they should not be fundamental. Be aware that seeking on directories is forbidden, so it is fine if you disregard the file offset when implementing directory reads; this makes it a little easier to handle the variable-length entries. You should still never return a partial directory entry; if the I/O count is too small to return even one entry, you should return two bytes containing the byte count required to represent the next entry in the directory. User code can use this value to formulate a retry if it desires. See the DIAGNOSTICS section of *stat*(2) for a description of this process.

The trickiest part of updating a file server is that the `clone` and `walk` messages have been merged into a single message, a sort of clone-multiwalk. The new message, still called `walk`, proposes a sequence of file name elements to be evaluated using a possibly cloned fid. The return message contains the qids of the files reached by walking to the sequential elements. If all the elements can be walked, the fid will be cloned if requested. If a non-zero number of elements are requested, but none can be walked, an error should be returned. If only some can be walked, the fid is not cloned, the original fid is left where it was, and the returned `Rwalk` message should contain the partial list of successfully reached qids. See *walk*(5) for a full description.

APE — The ANSI/POSIX Environment

Howard Trickey
howard@plan9.bell-labs.com

Introduction

When a large or frequently-updated program must be ported to or from Plan 9, the ANSI/POSIX environment known as APE can be useful. APE combines the set of headers and object code libraries specified by the ANSI C standard (ANSI X3.159-1989) with the POSIX operating system interface standard (IEEE 1003.1-1990, ISO 9945-1), the part of POSIX defining the basic operating system functions. Using APE will cause slower compilation and marginally slower execution speeds, so if the importing or exporting happens only infrequently, due consideration should be given to using the usual Plan 9 compilation environment instead. Another factor to consider is that the Plan 9 header organization is much simpler to remember and use.

There are some aspects of required POSIX behavior that are impossible or very hard to simulate in Plan 9. They are described below. Experience has shown, however, that the simulation is adequate for the vast majority of programs. A much more common problem is that many programs use functions or headers not defined by POSIX. APE has some extensions to POSIX to help in this regard. Extensions must be explicitly enabled with an appropriate #define, in order that the APE environment be a good aid for testing ANSI/POSIX compliance of programs.

Pcc

The pcc command acts as a front end to the Plan 9 C compilers and loaders. It runs an ANSI C preprocessor over source files, using the APE headers to satisfy #include <*file*> directives; then it runs a Plan 9 C compiler; finally, it may load with APE libraries to produce an executable program. The document *How to Use the Plan 9 C Compiler* explains how environment variables are used by convention to handle compilation for differing architectures. The environment variable $objtype controls which Plan 9 compiler and loader are used by pcc, as well as the location of header and library files. For example, if $objtype is mips, then pcc has cpp look for headers in /mips/include/ape followed by /sys/include/ape; then pcc uses vc to create .v object files; finally, vl is used to create an executable using libraries in /mips/lib/ape.

Psh and Cc

The pcc command is intended for uses where the source code is ANSI/POSIX, but the programs are built in the usual Plan 9 manner — with mk and producing object files with names ending in .v, etc. Sometimes it is best to use the standard POSIX make and cc (which produces object files with names ending in .o, and automatically calls the loader unless -c is specified). Under these circumstances, execute the command:

```
ape/psh
```

This starts a POSIX shell, with an environment that includes the POSIX commands ar89, c89, cc, basename, dirname, expr, false, grep, kill, make, rmdir, sed, sh, stty, true, uname, and yacc. There are also a few placeholders for commands that cannot be implemented in Plan 9: chown, ln, and umask.

The cc command accepts the options mandated for the POSIX command c89, as specified in the C-Language Development Utilities Option annex of the POSIX Shell and Utilities standard. It also accepts the following nonstandard options: -v for echoing the commands for each pass to stdout; -A to turn on ANSI prototype warnings; -S to leave assembly language in *file*.s; -Wp, *args* to pass *args* to the cpp; -W0, *args* to pass *args* to 2c, etc.; and -Wl, *args* to pass *args* to 2l, etc.

The sh command is pdksh, a mostly POSIX-compliant public domain Korn Shell. The Plan 9 implementation does not include the emacs and vi editing modes.

The stty command only has effect if the ape/ptyfs command has been started to interpose a pseudo-tty interface between /dev/cons and the running command. None of the distributed commands do this automatically.

Symbols

The C and POSIX standards require that certain symbols be defined in headers. They also require that certain other classes of symbols not be defined in the headers, and specify certain other symbols that may be defined in headers at the discretion of the implementation. POSIX defines *feature test macros*, which are preprocessor symbols beginning with an underscore and then a capital letter; if the program #defines a feature test macro before the inclusion of any headers, then it is requesting that certain symbols be visible in the headers. The most important feature test macro is _POSIX_SOURCE: when it is defined, exactly the symbols required by POSIX are visible in the appropriate headers. Consider <signal.h> for example: ANSI defines some names that must be defined in <signal.h>, but POSIX defines others, such as sigset_t, which are not allowed according to ANSI. The solution is to make the additional symbols visible only when _POSIX_SOURCE is defined.

To export a program, it helps to know whether it fits in one of the following categories:

1. Strictly conforming ANSI C program. It only uses features of the language, libraries, and headers explicitly required by the C standard. It does not depend on unspecified, undefined, or implementation-dependent behavior, and does not exceed any minimum implementation limit.
2. Strictly conforming POSIX program. Similar, but for the POSIX standard as well.
3. Some superset of POSIX, with extensions. Each extension is selected by a feature test macro, so it is clear which extensions are being used.

With APE, if headers are always included to declare any library functions used, then the set of feature test macros defined by a program will show which of the above categories the program is in. To accomplish this, no symbol is defined in a header if it is not required by the C or POSIX standard, and those required by the POSIX standard are protected by #ifdef _POSIX_SOURCE. For example, <errno.h> defines EDOM, ERANGE, and errno, as required by the C standard. The C standard allows more names beginning with E, but our header defines only those unless _POSIX_SOURCE is defined, in which case the symbols required by POSIX are also defined. This means that a program that uses ENAMETOOLONG cannot masquerade as a strictly conforming ANSI C program.

Pcc and cc do not predefine any preprocessor symbols except those required by the ANSI C standard: __STDC__, __LINE__, __FILE__, __DATE__, and __TIME__. Any others must be defined in the program itself or by using -D on the command line.

Extensions

The discipline enforced by putting only required names in the headers is useful for exporting programs, but it gets in the way when importing programs. The compromise is to allow additional symbols in headers, additional headers, and additional library functions, but only under control of extension feature test macros. The following extensions are provided; unless otherwise specified, the additional library functions are in the default APE library.

_LIBG_EXTENSION. This allows the use of the Plan 9 graphics library. The functions are as described in the Plan 9 manual (see *graphics* (2)) except that div had to be renamed ptdiv. Include the <libg.h> header to declare the needed types and functions.

_LIMITS_EXTENSION. POSIX does not require that names such as PATH_MAX and OPEN_MAX be defined in <limits.h>, but many programs assume they are defined there. If _LIMITS_EXTENSION is defined, those names will all be defined when <limits.h> is included.

_BSD_EXTENSION. This extension includes not only Berkeley Unix routines, but also a grab bag of other miscellaneous routines often found in Unix implementations. The extension allows the inclusion of any of: <bsd.h> for bcopy(), bcmp(), and similar Berkeley functions; <netdb.h> for gethostbyname(), etc., and associated structures; <select.h> for the Berkeley select function and associated types and macros for dealing with multiple input sources; <sys/ioctl.h> for the ioctl function (minimally implemented); <sys/param.h> for NOFILES_MAX; <sys/pty.h> for pseudo-tty support via the ptsname(int) and ptmname(int) functions; <sys/resource.h>; <sys/socket.h> for socket structures, constants, and functions; <sys/time.h> for definitions of the timeval and timezone structures; and <sys/uio.h> for the iovec structure and the writev and readv functions used for scatter/gather I/O. Defining _BSD_EXTENSION also enables various extra definitions in <ctype.h>, <signal.h>, <stdio.h>, <unistd.h>, <sys/stat.h>, and <sys/times.h>.

_NET_EXTENSION. This extension allows inclusion of <libnet.h>, which defines the networking functions described in the Plan 9 manual page *dial* (2).

_PLAN9_EXTENSION. This extension allows inclusion of <u.h>, <lock.h>, <qlock.h>, <utf.h>, <fmt.h>, and <draw.h>. These are pieces of Plan 9 source code ported into APE, mostly from <libc.h>.

_REGEXP_EXTENSION. This extension allows inclusion of <regexp.h>, which defines the regular expression matching functions described in the Plan 9 manual page *regexp* (2).

_RESEARCH_SOURCE. This extension enables a small library of functions from the Tenth Edition Unix Research System (V10). These functions and the types needed to use them are all defined in the <libv.h> header. The provided functions are: srand, rand, nrand, lrand, and frand (better random number generators); getpass, tty_echoon, tty_echooff (for dealing with the common needs for mucking with terminal characteristics); min and max; nap; and setfields, getfields, and getmfields (for parsing a line into fields). See the Research Unix System Programmer’s Manual, Tenth Edition, for a description of these functions.

_C99_SNPRINTF_EXTENSION. This extension permits the use of the return values of *snprintf* and *vsnprintf*. Before C99, the 1999 C standard, these functions usually returned the number of bytes, excluding terminating NUL, actually stored in the target string. (GNU, as usual, had to be different and returned -1 if the target string was too small.) C99 requires them to instead return the number of bytes, excluding terminating NUL, that would have been written into the target string if it were infinitely large or a negative value if an encoding error occurs, so old programs compiled under C99 rules will be prone to overrunning their buffers. This extension is a way for the programmer to declare that he or she understands the situation and has adjusted the code being compiled to compensate.

Common Problems

Some large systems, including X11, have been ported successfully to Plan 9 using APE (the X11 port is not included in the distribution, however, because supporting it properly is too big a job). The problems encountered fall into three categories: (1) non-ANSI C/POSIX features used; (2) inadequate simulation of POSIX functions; and (3) compiler/loader bugs. By far the majority of problems are in the first category.

POSIX is just starting to be a target for programmers. Most existing code is written to work with one or both of a BSD or a System V Unix. System V is fairly close to POSIX, but there are some differences. Also, many System V systems have imported some BSD features that are not part of POSIX. A good strategy for porting external programs is to first try using CFLAGS=-D_POSIX_SOURCE; if that doesn't work, try adding _D_BSD_EXTENSION and perhaps include <bsd.h> in source files. Here are some solutions to problems that might remain:

Third (environment) argument to main. Use the environ global instead.

OPEN_MAX, PATH_MAX, etc., assumed in <limits.h>. Rewrite to call sysconf or define _LIMITS_EXTENSION.

<varargs.h>. Rewrite to use <stdarg.h>.

The second class of problems has to do with inadequacies in the Plan 9 simulation of POSIX functions. These shortcomings have rarely gotten in the way (except perhaps, for the link problem).

Functions for setting the userid, groupid, effective userid and effective groupid do not do anything useful. The concept is impossible to simulate in Plan 9. Chown also does nothing.

execlp and the related functions do not look at the PATH environment variable. They just try the current directory and /bin if the pathname is not absolute.

Advisory locking via fcntl is not implemented.

isatty is hard to do correctly. The approximation used is only sometimes correct.

link always fails.

With open, the O_NOCTTY option has no effect. The concept of a controlling tty is foreign to Plan 9.

setsid forks the name space and note group, which is only approximately the right behavior.

The functions dealing with stacking signals, sigpending, sigprocmask and sigsuspend, do not work.

umask has no effect, as there is no such concept in Plan 9.

code that does getenv("HOME") should be changed to getenv("home") on Plan 9.

Acid: A Debugger Built From A Language

Phil Winterbottom
philw@plan9.bell-labs.com

ABSTRACT

Acid is an unusual source-level symbolic debugger for Plan 9. It is implemented as a language interpreter with specialized primitives that provide debugger support. Programs written in the language manipulate one or more target processes; variables in the language represent the symbols, state, and resources of those processes. This structure allows complex interaction between the debugger and the target program and provides a convenient method of parameterizing differences between machine architectures. Although some effort is required to learn the debugging language, the richness and flexibility of the debugging environment encourages new ways of reasoning about the way programs run and the conditions under which they fail.

1. Introduction

The size and complexity of programs have increased in proportion to processor speed and memory but the interface between debugger and programmer has changed little. Graphical user interfaces have eased some of the tedious aspects of the interaction. A graphical interface is a convenient means for navigating through source and data structures but provides little benefit for process control. The introduction of a new concurrent language, Alef [Win93], emphasized the inadequacies of the existing Plan 9 [Pike90] debugger *db*, a distant relative of *adb*, and made it clear that a new debugger was required.

Current debuggers like *dbx*, *sdb*, and *gdb* are limited to answering only the questions their authors envisage. As a result, they supply a plethora of specialized commands, each attempting to anticipate a specific question a user may ask. When a debugging situation arises that is beyond the scope of the command set, the tool is useless. Further, it is often tedious or impossible to reproduce an anomalous state of the program, especially when the state is embedded in the program's data structures.

Acid applies some ideas found in CAD software used for hardware test and simulation. It is based on the notion that the state and resources of a program are best represented and manipulated by a language. The state and resources, such as memory, registers, variables, type information and source code are represented by variables in the language. Expressions provide a computation mechanism and control statements allow repetitive or selective interpretation based on the result of expression evaluation. The heart of the Acid debugger is an interpreter for a small typeless language whose operators mirror the operations of C and Alef, which in turn correspond well to the basic operations of the machine. The interpreter itself knows nothing of the underlying hardware; it deals with the program state and resources in the abstract. Fundamental routines to control processes, read files, and interface to the system are implemented as builtin

---

Originally appeared in *Proc. of the Winter 1994 USENIX Conf.*, pp. 211-222, San Francisco, CA

functions available to the interpreter. The actual debugger functionality is coded in Acid; commands are implemented as Acid functions.

This language-based approach has several advantages. Most importantly, programs written in Acid, including most of the debugger itself, are inherently portable. Furthermore, Acid avoids the limitations other debuggers impose when debugging parallel programs. Instead of embedding a fixed process model in the debugger, Acid allows the programmer to adapt the debugger to handle an arbitrary process partitioning or program structure. The ability to interact dynamically with an executing process provides clear advantages over debuggers constrained to probe a static image. Finally, the Acid language is a powerful vehicle for expressing assertions about logic, process state, and the contents of data structures. When combined with dynamic interaction it allows a limited form of automated program verification without requiring modification or recompilation of the source code. The language is also an excellent vehicle for preserving a test suite for later regression testing.

The debugger may be customized by its users; standard functions may be modified or extended to suit a particular application or preference. For example, the kernel developers in our group require a command set supporting assembler-level debugging while the application programmers prefer source-level functionality. Although the default library is biased toward assembler-level debugging, it is easily modified to provide a convenient source-level interface. The debugger itself does not change; the user combines primitives and existing Acid functions in different ways to implement the desired interface.

2. Related Work

DUEL [Gol93], an extension to *gdb* [Stal91], proposes using a high level expression evaluator to solve some of these problems. The evaluator provides iterators to loop over data structures and conditionals to control evaluation of expressions. The author shows that complex state queries can be formulated by combining concise expressions but this only addresses part of the problem. A program is a dynamic entity; questions asked when the program is in a static state are meaningful only after the program has been caught in that state. The framework for manipulating the program is still as primitive as the underlying debugger. While DUEL provides a means to probe data structures it entirely neglects the most beneficial aspect of debugging languages: the ability to control processes. Acid is structured around a thread of control that passes between the interpreter and the target program.

The NeD debugger [May92] is a set of extensions to TCL [Ous90] that provide debugging primitives. The resulting language, NeDtcl, is used to implement a portable interface between a conventional debugger, pdb [May90], and a server that executes NeDtcl programs operating on the target program. Execution of the NeDtcl programs implements the debugging primitives that pdb expects. NeD is targeted at multi-process debugging across a network, and proves the flexibility of a language as a means of communication between debugging tools. Whereas NeD provides an interface between a conventional debugger and the process it debugs, Acid is the debugger itself. While NeD has some of the ideas found in Acid it is targeted toward a different purpose. Acid seeks to integrate the manipulation of a program s resources into the debugger while NeD provides a flexible interconnect between components of the debugging environment. The choice of TCL is appropriate for its use in NeD but is not suitable for Acid. Acid relies on the coupling of the type system with expression evaluation, which are the root of its design, to provide the debugging primitives.

Dalek [Ols90] is an event based language extension to gdb. State transitions in the target program cause events to be queued for processing by the debugging language.

Acid has many of the advantages of same processor *local agent* debuggers, like Parasight [Aral], without the need for dynamic linking or shared memory. Acid improves on the ideas of these other systems by completely integrating all aspects of the debugging process into the language environment. Of particular importance is the relationship between Acid variables, program symbols, source code, registers and type information. This integration is made possible by the design of the Acid language.

Interpreted languages such as Lisp and Smalltalk are able to provide richer debugging environments through more complete information than their compiled counterparts. Acid is a means to gather and represent similar information about compiled programs through cooperation with the compilation tools and library implementers.

3. Acid the Language

Acid is a small interpreted language targeted to its debugging task. It focuses on representing program state and addressing data rather than expressing complex computations. Program state is *addressable* from an Acid program. In addition to parsing and executing expressions and providing an architecture-independent interface to the target process, the interpreter supplies a mark-and-scan garbage collector to manage storage.

Every Acid session begins with the loading of the Acid libraries. These libraries contain functions, written in Acid, that provide a standard debugging environment including breakpoint management, stepping by instruction or statement, stack tracing, and access to variables, memory, and registers. The library contains 600 lines of Acid code and provides functionality similar to *dbx*. Following the loading of the system library, Acid loads user-specified libraries; this load sequence allows the user to augment or override the standard commands to customize the debugging environment. When all libraries are loaded, Acid issues an interactive prompt and begins evaluating expressions entered by the user. The Acid commands are actually invocations of builtin primitives or previously defined Acid functions. Acid evaluates each expression as it is entered and prints the result.

4. Types and Variables

Acid variables are of four basic types: *integer*, *string*, *float*, and *list*. The type of a variable is inferred by the type of the right-hand side of an assignment expression. Many of the operators can be applied to more than one type; for these operators the action of the operator is determined by the type of its operands. For example, the + operator adds *integer* and *float* operands, and concatenates *string* and *list* operands. Lists are the only complex type in Acid; there are no arrays, structures or pointers. Operators provide head, tail, append and delete operations. Lists can also be indexed like arrays.

Acid has two levels of scope: global and local. Function parameters and variables declared in a function body using the local keyword are created at entry to the function and exist for the lifetime of a function. Global variables are created by assignment and need not be declared. All variables and functions in the program being debugged are entered in the Acid symbol table as global variables during Acid initialization. Conflicting variable names are resolved by prefixing enough $ characters to make them unique. Syntactically, Acid variables and target program symbols are referenced identically. However, the variables are managed differently in the Acid symbol table and the user must be aware of this distinction. The value of an Acid variable is stored in the symbol table; a reference returns the value. The symbol table entry for a variable or function in the target program contains the address of that symbol in the image of the program. Thus, the value of a program variable is accessed by indirect reference through the Acid variable that has the same name; the value of an Acid variable is the address of the corresponding program variable.

5. Control Flow

The while and loop statements implement looping. The former is similar to the same statement in C. The latter evaluates starting and ending expressions yielding integers and iterates while an incrementing loop index is within the bounds of those expressions.

```
acid: i = 0; loop 1,5 do print(i=i+1)
0x00000001
0x00000002
0x00000003
0x00000004
0x00000005
acid:
```

The traditional if-then-else statement implements conditional execution.

6. Addressing

Two indirection operators allow Acid to access values in the program being debugged. The * operator fetches a value from the memory image of an executing process; the @ operator fetches a value from the text file of the process. When either operator appears on the left side of an assignment, the value is written rather than read.

The indirection operator must know the size of the object referenced by a variable. The Plan 9 compilers neglect to include this information in the program symbol table, so Acid cannot derive this information implicitly. Instead Acid variables have formats. The format is a code letter specifying the printing style and the effect of some of the operators on that variable. The indirection operators look at the format code to determine the number of bytes to read or write. The format codes are derived from the format letters used by *db*. By default, symbol table variables and numeric constants are assigned the format code 'X' which specifies 32-bit hexadecimal. Printing such a variable yields output of the form 0x00123456 . An indirect reference through the variable fetches 32 bits of data at the address indicated by the variable. Other formats specify various data types, for example i an instruction, D a signed 32 bit decimal, s a null-terminated string. The fmt function allows the user to change the format code of a variable to control the printing format and operator side effects. This function evaluates the expression supplied as the first argument, attaches the format code supplied as the second argument to the result and returns that value. If the result is assigned to a variable, the new format code applies to that variable. For convenience, Acid provides the \ operator as a shorthand infix form of fmt. For example:

```
acid: x=10
acid: x                              // print x in hex
0x0000000a
acid: x = fmt(x, 'D')                // make x type decimal
acid: print(x, fmt(x, 'X'), x\X)     // print x in decimal & hex
10 0x0000000a 0x0000000a
acid: x                              // print x in decimal
10
acid: x\o                            // print x in octal
000000000012
```

The ++ and -- operators increment or decrement a variable by an amount determined by its format code. Some formats imply a non-fixed size. For example, the i format code disassembles an instruction into a string. On a 68020, which has variable length instructions:

```
acid: p=main\i                          // p=addr(main), type INST
acid: loop 1,5 do print(p\X, @p++)      // disassemble 5 instr's
0x0000222e   LEA   0xffffe948(A7),A7
0x00002232   MOVL  s+0x4(A7),A2
0x00002236   PEA   0x2f($0)
0x0000223a   MOVL  A2,-(A7)
0x0000223c   BSR   utfrrune
acid:
```

Here, main is the address of the function of the same name in the program under test. The loop retrieves the five instructions beginning at that address and then prints the address and the assembly language representation of each. Notice that the stride of the increment operator varies with the size of the instruction: the MOVL at 0x0000223a is a two byte instruction while all others are four bytes long.

Registers are treated as normal program variables referenced by their symbolic assembler language names. When a process stops, the register set is saved by the kernel at a known virtual address in the process memory map. The Acid variables associated with the registers point to the saved values and the * indirection operator can then be used to read and write the register set. Since the registers are accessed via Acid variables they may be used in arbitrary expressions.

```
acid: PC                                // addr of saved PC
0xc0000f60
acid: *PC
0x0000623c                              // contents of PC
acid: *PC\a
main
acid: *R1=10                            // modify R1
acid: asm(*PC+4)                        // disassemble @ PC+4
main+0x4   0x00006240      MOVW     R31,0x0(R29)
main+0x8   0x00006244      MOVW     $setR30(SB),R30
main+0x10   0x0000624c     MOVW     R1,_clock(SB)
```

Here, the saved PC is stored at address 0xc0000f60 ; its current content is 0x0000623c . The a format code converts this value to a string specifying the address as an offset beyond the nearest symbol. After setting the value of register R1, the example uses the asm command to disassemble a short section of code beginning at four bytes beyond the current value of the PC.

7. Process Interface

A program executing under Acid is monitored through the *proc* file system interface provided by Plan 9. Textual messages written to the ctl file control the execution of the process. For example writing waitstop to the control file causes the write to block until the target process enters the kernel and is stopped. When the process is stopped the write completes. The startstop message starts the target process and then does a waitstop action. Synchronization between the debugger and the target process is determined by the actions of the various messages. Some operate asynchronously to the target process and always complete immediately, others block until the action completes. The asynchronous messages allow Acid to control several processes simultaneously.

The interpreter has builtin functions named after each of the control messages. The functions take a process id as argument. Any time a control message causes the program to execute instructions the interpreter performs two actions when the control operation has completed. The Acid variables pointing at the register set are fixed up to point at the saved registers, and then the user defined function stopped is executed. The stopped function may print the current address, line of source or instruction and return to interactive mode. Alternatively it may traverse a complex data structure, gather statistics and then set the program running again.

Several Acid variables are maintained by the debugger rather than the programmer. These variables allow generic Acid code to deal with the current process, architecture specifics or the symbol table. The variable pid is the process id of the current process Acid is debugging. The variable symbols contains a list of lists where each sublist contains the symbol name, its type and the value of the symbol. The variable registers contains a list of the machine-specific register names. Global symbols in the target program can be referenced directly by name from Acid. Local variables are referenced using the colon operator as function:variable.

8. Source Level Debugging

Acid provides several builtin functions to manipulate source code. The file function reads a text file, inserting each line into a list. The pcfile and pcline functions each take an address as an argument. The first returns a string containing the name of the source file and the second returns an integer containing the line number of the source line containing the instruction at the address.

```
acid: pcfile(main)                    // file containing main
main.c
acid: pcline(main)                    // line # of main in source
11
acid: file(pcfile(main))[pcline(main)]          // print that line
main(int argc, char *argv[])
acid: src(*PC)                        // print statements nearby
 9
 10 void
>11 main(int argc, char *argv[])
 12 {
 13      int a;
```

In this example, the three primitives are combined in an expression to print a line of source code associated with an address. The src function prints a few lines of source around the address supplied as its argument. A companion routine, Bsrc, communicates with the external editor sam. Given an address, it loads the corresponding source file into the editor and highlights the line containing the address. This simple interface is easily extended to more complex functions. For example, the step function can select the current file and line in the editor each time the target program stops, giving the user a visual trace of the execution path of the program. A more complete interface allowing two way communication between Acid and the acme user interface [Pike93] is under construction. A filter between the debugger and the user interface provides interpretation of results from both sides of the interface. This allows the programming environment to interact with the debugger and vice-versa, a capability missing from the sam interface. The src and Bsrc functions are both written in Acid code using the file and line primitives. Acid provides library functions to step through source level statements and functions. Furthermore, addresses in Acid expressions can be specified by source file and line. Source code is manipulated in the Acid *list* data type.

9. The Acid Library

The following examples define some useful commands and illustrate the interaction of the debugger and the interpreter.

```
defn bpset(addr)                          // set breakpoint
{
        if match(addr, bplist) >= 0 then
                print("bkpoint already set:", addr\a, "\n");
        else {
                *fmt(addr, bpfmt) = bpinst;     // plant it
                bplist = append bplist, addr;   // add to list
        }
}
```

The bpset function plants a break point in memory. The function starts by using the match builtin to search the breakpoint list to determine if a breakpoint is already set at the address. The indirection operator, controlled by the format code returned by the fmt primitive, is used to plant the breakpoint in memory. The variables bpfmt and bpinst are Acid global variables containing the format code specifying the size of the breakpoint instruction and the breakpoint instruction itself. These variables are set by architecture-dependent library code when the debugger first attaches to the executing image. Finally the address of the breakpoint is appended to the breakpoint list, bplist .

```
defn step()                               // single step
{
        local lst, lpl, addr, bput;

        bput = 0;                         // sitting on bkpoint
        if match(*PC, bplist) >= 0 then {
                bput = fmt(*PC, bpfmt);   // save current addr
                *bput = @bput;            // replace it
        }

        lst = follow(*PC);                // get follow set

        lpl = lst;
        while lpl do {                    // place breakpoints
                *(head lpl) = bpinst;
                lpl = tail lpl;
        }

        startstop(pid);                   // do the step

        while lst do {                    // remove breakpoints
                addr = fmt(head lst, bpfmt);
                *addr = @addr;            // replace instr.
                lst = tail lst;
        }
        if bput != 0 then
                *bput = bpinst;           // restore breakpoint
}
```

The step function executes a single assembler instruction. If the PC is sitting on a breakpoint, the address and size of the breakpoint are saved. The breakpoint instruction is then removed using the @ operator to fetch bpfmt bytes from the text file and to place it into the memory of the executing process using the * operator. The follow function is an Acid builtin which returns a follow-set: a list of instruction addresses which could be executed next. If the instruction stored at the PC is a branch instruction, the list contains the addresses of the next instruction and the branch destination; otherwise, it contains only the address of the next instruction. The follow-set is then used to replace each possible following instruction with a breakpoint instruction. The original instructions need not be saved; they remain in their unaltered state in the text file. The startstop builtin writes the startstop message to the *proc* control file for the process named pid. The target process executes until some condition causes it to enter the kernel, in this case, the execution of a breakpoint. When the process blocks, the debugger regains control and invokes the Acid library function stopped which reports the address and cause of the blockage. The startstop function completes and returns to the step function where the follow-set is used to replace the breakpoints placed earlier. Finally, if the address of the original PC contained a breakpoint, it is replaced.

Notice that this approach to process control is inherently portable; the Acid code is shared by the debuggers for all architectures. Acid variables and builtin functions provide a transparent interface to architecture-dependent values and functions. Here the breakpoint value and format are referenced through Acid variables and the follow primitive masks the differences in the underlying instruction set.

The next function, similar to the *dbx* command of the same name, is a simpler example. This function steps through a single source statement but steps over function calls.

```
defn next()
{
        local sp, bound;

        sp = *SP;                           // save starting SP
        bound = fnbound(*PC);               // begin & end of fn.
        stmnt();                            // step 1 statement
        pc = *PC;
        if pc >= bound[0] && pc < bound[1] then
                return {};

        while (pc<bound[0] || pc>bound[1]) && sp>=*SP do {
                step();
                pc = *PC;
        }
        src(*PC);
}
```

The next function starts by saving the current stack pointer in a local variable. It then uses the Acid library function fnbound to return the addresses of the first and last instructions in the current function in a list. The stmnt function executes a single source statement and then uses src to print a few lines of source around the new PC. If the new value of the PC remains in the current function, next returns. When the executed statement is a function call or a return from a function, the new value of the PC is outside the bounds calculated by fnbound and the test of the while loop is evaluated. If the statement was a return, the new value of the stack pointer is greater than the original value and the loop completes without execution. Otherwise, the loop is entered and instructions are continually executed until the value of the PC is between the bounds calculated earlier. At that point, execution ceases and a few lines of source in the vicinity of the PC are printed.

Acid provides concise and elegant expression for control and manipulation of target programs. These examples demonstrate how a few well-chosen primitives can be combined to create a rich debugging environment.

10. Dealing With Multiple Architectures

A single binary of Acid may be used to debug a program running on any of the five processor architectures supported by Plan 9. For example, Plan 9 allows a user on a MIPS to import the *proc* file system from an i486-based PC and remotely debug a program executing on that processor.

Two levels of abstraction provide this architecture independence. On the lowest level, a Plan 9 library supplies functions to decode the file header of the program being debugged and select a table of system parameters and a jump vector of architecture-dependent functions based on the magic number. Among these functions are byte-order-independent access to memory and text files, stack manipulation, disassembly, and floating point number interpretation. The second level of abstraction is supplied by Acid. It consists of primitives and approximately 200 lines of architecture-dependent Acid library code that interface the interpreter to the architecture-dependent library. This layer performs functions such as mapping register names to memory locations, supplying breakpoint values and sizes, and converting processor specific data to Acid data types. An example of the latter is the stack trace function strace, which uses the stack traversal functions in the architecture-dependent library to construct a list of lists describing the context of a process. The first level of list selects each function in the trace; subordinate lists contain the names and values of parameters and local variables of the functions. Acid commands and library functions that manipulate and display process state information operate on the list representation and are independent of the underlying architecture.

11. Alef Runtime

Alef is a concurrent programming language, designed specifically for systems programming, which supports both shared variable and message passing paradigms. Alef borrows the C expression syntax but implements a substantially different type system. The language provides a rich set of exception handling, process management, and synchronization primitives, which rely on a runtime system. Alef program bugs are often deadlocks, synchronization failures, or non-termination caused by locks being held incorrectly. In such cases, a process stalls deep in the runtime code and it is clearly unreasonable to expect a programmer using the language to understand the detailed internal semantics of the runtime support functions.

Instead, there is an Alef support library, coded in Acid, that allows the programmer to interpret the program state in terms of Alef operations. Consider the example of a multi-process program stalling because of improper synchronization. A stack trace of the program indicates that it is waiting for an event in some obscure Alef runtime synchronization function. The function itself is irrelevant to the programmer; of greater importance is the identity of the unfulfilled event. Commands in the Alef support library decode the runtime data structures and program state to report the cause of the blockage in terms of the high-level operations available to the Alef programmer. Here, the Acid language acts as a communications medium between Alef implementer and Alef user.

12. Parallel Debugging

The central issue in parallel debugging is how the debugger is multiplexed between the processes comprising the program. Acid has no intrinsic model of process partitioning; it only assumes that parallel programs share a symbol table, though they need not share memory. The setproc primitive attaches the debugger to a running process associated with the process ID supplied as its argument and assigns that value to the global variable pid, thereby allowing simple rotation among a group of processes. Further, the stack trace primitive is driven by parameters specifying a unique process context, so it is possible to examine the state of cooperating processes without switching the debugger focus from the process of interest. Since Acid is inherently extensible and capable of dynamic interaction with subordinate processes, the programmer can define Acid commands to detect and control complex interactions between processes. In short, the programmer is free to specify how the debugger reacts to events generated in specific threads of the program.

The support for parallel debugging in Acid depends on a crucial kernel modification: when the text segment of a program is written (usually to place a breakpoint), the segment is cloned to prevent other threads from encountering the breakpoint. Although this incurs a slight performance penalty, it is of little importance while debugging.

13. Communication Between Tools

The Plan 9 Alef and C compilers do not embed detailed type information in the symbol table of an executable file. However, they do accept a command line option causing them to emit descriptions of complex data types (e.g., aggregates and abstract data types) to an auxiliary file. The vehicle for expressing this information is Acid source code. When an Acid debugging session is subsequently started, that file is loaded with the other Acid libraries.

For each complex object in the program the compiler generates three pieces of Acid code. The first is a table describing the size and offset of each member of the complex data type. Following is an Acid function, named the same as the object, that formats and prints each member. Finally, Acid declarations associate the Alef or C program variables of a type with the functions to print them. The three forms of declaration are shown in the following example:

```
struct Bitmap {
        Rectangle      0 r;
        Rectangle     16 clipr;
        'D'    32 ldepth;
        'D'    36 id;
        'X'    40 cache;
};

defn
Bitmap(addr) {
        complex Bitmap addr;
        print("Rectangle    r {\n");
        Rectangle(addr.r);
        print("}\n");
        print("Rectangle    clipr {\n");
        Rectangle(addr.clipr);
        print("}\n");
        print("  ldepth    ", addr.ldepth,   "\n");
        print("  id        ", addr.id,   "\n");
        print("  cache     ", addr.cache,   "\n");
};

complex Bitmap darkgrey;
complex Bitmap Window_settag:b;
```

The struct declaration specifies decoding instructions for the complex type named Bitmap. Although the syntax is superficially similar to a C structure declaration, the semantics differ markedly: the C declaration specifies a layout, while the Acid declaration tells how to decode it. The declaration specifies a type, an offset, and name for each member of the complex object. The type is either the name of another complex declaration, for example, Rectangle, or a format code. The offset is the number of bytes from the start of the object to the member and the name is the member's name in the Alef or C declaration. This type description is a close match for C and Alef, but is simple enough to be language independent.

The Bitmap function expects the address of a Bitmap as its only argument. It uses the decoding information contained in the Bitmap structure declaration to extract, format, and print the value of each member of the complex object pointed to by the argument. The Alef compiler emits code to call other Acid functions where a member is another complex type; here, Bitmap calls Rectangle to print its contents.

The complex declarations associate Alef variables with complex types. In the example, darkgrey is the name of a global variable of type Bitmap in the program being debugged. Whenever the name darkgrey is evaluated by Acid, it automatically calls the Bitmap function with the address of darkgrey as the argument. The second complex declaration associates a local variable or parameter named b in function Window_settag with the Bitmap complex data type.

Acid borrows the C operators . and -> to access the decoding parameters of a member of a complex type. Although this representation is sufficiently general for describing the decoding of both C and Alef complex data types, it may prove too restrictive for target languages with more complicated type systems. Further, the assumption that the compiler can select the proper Acid format code for each basic type in the language is somewhat naive. For example, when a member of a complex type is a pointer, it is assigned a hexadecimal type code; integer members are always assigned a decimal type code. This heuristic proves inaccurate when an integer field is a bit mask or set of bit flags which are more appropriately displayed in hexadecimal or octal.

14. Code Verification

Acids ability to interact dynamically with an executing program allows passive test and verification of the target program. For example, a common concern is leak detection in programs using malloc. Of interest are two items: finding memory that was allocated but never freed and detecting bad pointers passed to free. An auxiliary Acid library contains Acid functions to monitor the execution of a program and detect these faults, either as they happen or in the automated post-mortem analysis of the memory arena. In the following example, the sort command is run under the control of the Acid memory leak library.

```
helix% acid -l malloc /bin/sort
/bin/sort: mips plan 9 executable
/lib/acid/port
/lib/acid/mips
/lib/acid/malloc
acid: go()
now
is
the
time
<ctrl-d>
is
now
the
time
27680 : breakpoint      _exits+0x4      MOVW    $0x8,R1
acid:
```

The go command creates a process and plants breakpoints at the entry to malloc and free. The program is then started and continues until it exits or stops. If the reason for stopping is anything other than the breakpoints in malloc and free, Acid prints the usual status information and returns to the interactive prompt.

When the process stops on entering malloc, the debugger must capture and save the address that malloc will return. After saving a stack trace so the calling routine can be identified, it places a breakpoint at the return address and restarts the program. When malloc returns, the breakpoint stops the program, allowing the debugger to grab the address of the new memory block from the return register. The address and stack trace are added to the list of outstanding memory blocks, the breakpoint is removed from the return point, and the process is restarted.

When the process stops at the beginning of free, the memory address supplied as the argument is compared to the list of outstanding memory blocks. If it is not found an error message and a stack trace of the call is reported; otherwise, the address is deleted from the list.

When the program exits, the list of outstanding memory blocks contains the addresses of all blocks that were allocated but never freed. The leak library function traverses the list producing a report describing the allocated blocks.

```
acid: leak()
Lost a total of 524288 bytes from:
     malloc() malloc.c:32 called from dofile+0xe8 sort.c:217
     dofile() sort.c:190 called from main+0xac sort.c:161
     main() sort.c:128 called from _main+0x20 main9.s:10
Lost a total of 64 bytes from:
     malloc() malloc.c:32 called from newline+0xfc sort.c:280
     newline() sort.c:248 called from dofile+0x110 sort.c:222
     dofile() sort.c:190 called from main+0xac sort.c:161
     main() sort.c:128 called from _main+0x20 main9.s:10
Lost a total of 64 bytes from:
     malloc() malloc.c:32 called from realloc+0x14 malloc.c:129
     realloc() malloc.c:123 called from bldkey+0x358 sort.c:1388
     buildkey() sort.c:1345 called from newline+0x150 sort.c:285
     newline() sort.c:248 called from dofile+0x110 sort.c:222
     dofile() sort.c:190 called from main+0xac sort.c:161
     main() sort.c:128 called from _main+0x20 main9.s:10
acid: refs()
data...bss...stack...
acid: leak()
acid:
```

The presence of a block in the allocation list does not imply it is there because of a leak; for instance, it may have been in use when the program terminated. The refs() library function scans the *data*, *bss*, and *stack* segments of the process looking for pointers into the allocated blocks. When one is found, the block is deleted from the outstanding block list. The leak function is used again to report the blocks remaining allocated and unreferenced. This strategy proves effective in detecting disconnected (but non-circular) data structures.

The leak detection process is entirely passive. The program is not specially compiled and the source code is not required. As with the Acid support functions for the Alef runtime environment, the author of the library routines has encapsulated the functionality of the library interface in Acid code. Any programmer may then check a program s use of the library routines without knowledge of either implementation. The performance impact of running leak detection is great (about 10 times slower), but it has not prevented interactive programs like sam and the 8½ window system from being tested.

15. Code Coverage

Another common component of software test uses *coverage* analysis. The purpose of the test is to determine which paths through the code have not been executed while running the test suite. This is usually performed by a combination of compiler support and a reporting tool run on the output generated by statements compiled into the program. The compiler emits code that logs the progress of the program as it executes basic blocks and writes the results to a file. The file is then processed by the reporting tool to determine which basic blocks have not been executed.

Acid can perform the same function in a language independent manner without modifying the source, object or binary of the program. The following example shows ls being run under the control of the Acid coverage library.

```
philw-helix% acid -l coverage /bin/ls
/bin/ls: mips plan 9 executable
/lib/acid/port
/lib/acid/mips
/lib/acid/coverage
acid: coverage()
acid
newstime
profile
tel
wintool
2: (error) msg: pid=11419 startstop: process exited
acid: analyse(ls)
ls.c:102,105
          102:     return 1;
          103: }
          104: if(db[0].qid.path&CHDIR && dflag==0){
          105:     output();
ls.c:122,126
          122:     memmove(dirbuf+ndir, db, sizeof(Dir));
          123:     dirbuf[ndir].prefix = 0;
          124:     p = utfrrune(s, '/');
          125:     if(p){
          126:         dirbuf[ndir].prefix = s;
```

The coverage function begins by looping through the text segment placing breakpoints at the entry to each basic block. The start of each basic block is found using the Acid builtin function follow. If the list generated by follow contains more than one element, then the addresses mark the start of basic blocks. A breakpoint is placed at each address to detect entry into the block. If the result of follow is a single address then no action is taken, and the next address is considered. Acid maintains a list of breakpoints already in place and avoids placing duplicates (an address may be the destination of several branches).

After placing the breakpoints the program is set running. Each time a breakpoint is encountered Acid deletes the address from the breakpoint list, removes the breakpoint from memory and then restarts the program. At any instant the breakpoint list contains the addresses of basic blocks which have not been executed. The analyse function reports the lines of source code bounded by basic blocks whose addresses are have not been deleted from the breakpoint list. These are the basic blocks which have not been executed. Program performance is almost unaffected since each breakpoint is executed only once and then removed.

The library contains a total of 128 lines of Acid code. An obvious extension of this algorithm could be used to provide basic block profiling.

16. Conclusion

Acid has two areas of weakness. As with other language-based tools like *awk*, a programmer must learn yet another language to step beyond the normal debugging functions and use the full power of the debugger. Second, the command line interface supplied by the *yacc* parser is inordinately clumsy. Part of the problem relates directly to the use of *yacc* and could be circumvented with a custom parser. However, structural problems would remain: Acid often requires too much typing to execute a simple command. A debugger should prostitute itself to its users, doing whatever is wanted with a minimum of encouragement; commands should be concise and obvious. The language interface is more consistent than an ad hoc command interface but is clumsy to use. Most of these problems are addressed by an Acme interface which is under construction. This should provide the best of both worlds: graphical debugging and access to the underlying acid language when required.

The name space clash between Acid variables, keywords, program variables, and functions is unavoidable. Although it rarely affects a debugging session, it is annoying when it happens and is sometimes difficult to circumvent. The current renaming scheme is too crude; the new names are too hard to remember.

Acid has proved to be a powerful tool whose applications have exceeded expectations. Of its strengths, portability, extensibility and parallel debugging support were by design and provide the expected utility. In retrospect, its use as a tool for code test and verification and as a medium for communicating type information and encapsulating interfaces has provided unanticipated benefits and altered our view of the debugging process.

17. Acknowledgments

Bob Flandrena was the first user and helped prepare the paper. Rob Pike endured three buggy Alef compilers and a new debugger in a single sitting.

18. References

[Pike90] R. Pike, D. Presotto, K. Thompson, H. Trickey, Plan 9 from Bell Labs , *UKUUG Proc. of the Summer 1990 Conf.*, London, England, 1990, reprinted, in a different form, in this volume.

[Gol93] M. Golan, D. Hanson, DUEL -- A Very High-Level Debugging Language , *USENIX Proc. of the Winter 1993 Conf.*, San Diego, CA, 1993.

[Lin90] M. A. Linton, The Evolution of DBX , *USENIX Proc. of the Summer 1990 Conf.*, Anaheim, CA, 1990.

[Stal91] R. M. Stallman, R. H. Pesch, Using GDB: A guide to the GNU source level debugger , Technical Report, Free Software Foundation, Cambridge, MA, 1991.

[Win93] P. Winterbottom, Alef reference Manual, this volume.

[Pike93] Rob Pike, Acme: A User Interface for Programmers , *USENIX Proc. of the Winter 1994 Conf.*, San Francisco, CA, reprinted in this volume.

[Ols90] Ronald A. Olsson, Richard H. Crawford, and W. Wilson Ho, Dalek: A GNU, improved programmable debugger , *USENIX Proc. of the Summer 1990 Conf.*, Anaheim, CA.

[May92] Paul Maybee, NeD: The Network Extensible Debugger *USENIX Proc. of the Summer 1992 Conf.*, San Antonio, TX.

[Aral] Ziya Aral, Ilya Gertner, and Greg Schaffer, Efficient debugging primitives for multiprocessors, *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, SIGPLAN notices Nr. 22, May 1989.

Acid Manual

Phil Winterbottom
philw@plan9.bell-labs.com

Introduction

Acid is a general purpose, source level symbolic debugger. The debugger is built around a simple command language. The command language, distinct from the language of the program being debugged, provides a flexible user interface that allows the debugger interface to be customized for a specific application or architecture. Moreover, it provides an opportunity to write test and verification code independently of a program's source code. Acid is able to debug multiple processes provided they share a common set of symbols, such as the processes in a threaded program.

Like other language-based solutions, Acid presents a poor user interface but provides a powerful debugging tool. Application of Acid to hard problems is best approached by writing functions off-line (perhaps loading them with the include function or using the support provided by *acme*(1)), rather than by trying to type intricate Acid operations at the interactive prompt.

Acid allows the execution of a program to be controlled by operating on its state while it is stopped and by monitoring and controlling its execution when it is running. Each program action that causes a change of execution state is reflected by the execution of an Acid function, which may be user defined. A library of default functions provides the functionality of a normal debugger.

A Plan 9 process is controlled by writing messages to a control file in the *proc*(3) file system. Each control message has a corresponding Acid function, which sends the message to the process. These functions take a process id (*pid*) as an argument. The memory and text file of the program may be manipulated using the indirection operators. The symbol table, including source cross reference, is available to an Acid program. The combination allows complex operations to be performed both in terms of control flow and data manipulation.

Input format and `whatis`

Comments start with // and continue to the end of the line. Input is a series of statements and expressions separated by semicolons. At the top level of the interpreter, the builtin function `print` is called automatically to display the result of all expressions except function calls. A unary + may be used as a shorthand to force the result of a function call to be printed.

Also at the top level, newlines are treated as semicolons by the parser, so semicolons are unnecessary when evaluating expressions.

When Acid starts, it loads the default program modules, enters interactive mode, and prints a prompt. In this state Acid accepts either function definitions or statements to be evaluated. In this interactive mode statements are evaluated immediately, while function definitions are stored for later invocation.

The whatis operator can be used to report the state of identifiers known to the interpreter. With no argument, whatis reports the name of all defined Acid functions; when supplied with an identifier as an argument it reports any variable, function, or type definition associated with the identifier. Because of the way the interpreter handles semicolons, the result of a whatis statement can be returned directly to Acid without adding semicolons. A syntax error or interrupt returns Acid to the normal evaluation mode; any partially evaluated definitions are lost.

Using the Library Functions

After loading the program binary, Acid loads the portable and architecture-specific library functions that form the standard debugging environment. These files are Acid source code and are human-readable. The following example uses the standard debugging library to show how language and program interact:

```
% acid /bin/ls
/bin/ls:mips plan 9 executable

/sys/lib/acid/port
/sys/lib/acid/mips
acid: new()
75721: system call _main ADD $-0x14,R29
75721: breakpoint main+0x4 MOVW R31,0x0(R29)
acid: bpset(ls)
acid: cont()
75721: breakpoint ls ADD $-0x16c8,R29
acid: stk()
At pc:0x0000141c:ls /sys/src/cmd/ls.c:87
ls(s=0x0000004d,multi=0x00000000) /sys/src/cmd/ls.c:87
    called from main+0xf4 /sys/src/cmd/ls.c:79
main(argc=0x00000000,argv=0x7ffffff0) /sys/src/cmd/ls.c:48
    called from _main+0x20 /sys/src/libc/mips/main9.s:10
acid: PC
0xc0000f60
acid: *PC
0x0000141c
acid: ls
0x0000141c
```

The function new() creates a new process and stops it at the first instruction. This change in state is reported by a call to the Acid function stopped, which is called by the interpreter whenever the debugged program stops. Stopped prints the status line giving the pid, the reason the program stopped and the address and instruction at the current PC. The function bpset makes an entry in the breakpoint table and plants a breakpoint in memory. The cont function continues the process, allowing it to run until some condition causes it to stop. In this case the program hits the breakpoint placed on the function ls in the C program. Once again the stopped routine is called to print the status of the program. The function stk prints a C stack trace of the current process. It is implemented using a builtin Acid function that returns the stack trace as a list; the code that formats the information is all written in Acid. The Acid variable PC holds the address of the cell where the current value of the processor register PC is stored. By indirecting through the value of PC the address where the program is stopped can be found. All of the processor registers are available by the same mechanism.

Types

An Acid variable has one of four types: *integer*, *float*, *list*, or *string*. The type of a variable is inferred from the type of the right-hand side of the assignment expression which last set its value. Referencing a variable that has not yet been assigned draws a "used but not set" error. Many of the operators may be applied to more than one type; for these operators the action of the operator is determined by the types of its operands. The action of each operator is defined in the *Expressions* section of this manual.

Variables

Acid has three kinds of variables: variables defined by the symbol table of the debugged program, variables that are defined and maintained by the interpreter as the debugged program changes state, and variables defined and used by Acid programs.

Some examples of variables maintained by the interpreter are the register pointers listed by name in the Acid list variable `registers`, and the symbol table listed by name and contents in the Acid variable `symbols`.

The variable `pid` is updated by the interpreter to select the most recently created process or the process selected by the `setproc` builtin function.

Formats

In addition to a type, variables have formats. The format is a code letter that determines the printing style and the effect of some of the operators on that variable. The format codes are derived from the format letters used by *db*(1). By default, symbol table variables and numeric constants are assigned the format code X, which specifies 32-bit hexadecimal. Printing a variable with this code yields the output `0x00123456`. The format code of a variable may be changed from the default by using the builtin function `fmt`. This function takes two arguments, an expression and a format code. After the expression is evaluated the new format code is attached to the result and forms the return value from `fmt`. The backslash operator is a short form of `fmt`. The format supplied by the backslash operator must be the format character rather than an expression. If the result is assigned to a variable the new format code is maintained in the variable. For example:

```
acid: x=10
acid: print(x)
0x0000000a
acid: x = fmt(x, 'D')
acid: print(x, fmt(x, 'X'))
10 0x0000000a
acid: x
10
acid: x\o
12
```

The supported format characters are:

- o Print two-byte integer in octal.
- O Print four-byte integer in octal.
- q Print two-byte integer in signed octal.
- Q Print four-byte integer in signed octal.
- B Print four-byte integer in binary.
- d Print two-byte integer in signed decimal.
- D Print four-byte integer in signed decimal.

V Print eight-byte integer in signed decimal.

Z Print eight-byte integer in unsigned decimal.

x Print two-byte integer in hexadecimal.

X Print four-byte integer in hexadecimal.

Y Print eight-byte integer in hexadecimal.

u Print two-byte integer in unsigned decimal.

U Print four-byte integer in unsigned decimal.

f Print single-precision floating point number.

F Print double-precision floating point number.

g Print a single precision floating point number in string format.

G Print a double precision floating point number in string format.

b Print byte in hexadecimal.

c Print byte as an ASCII character.

C Like c, with printable ASCII characters represented normally and others printed in the form \x*nn*.

s Interpret the addressed bytes as UTF characters and print successive characters until a zero byte is reached.

r Print a two-byte integer as a rune.

R Print successive two-byte integers as runes until a zero rune is reached.

i Print as machine instructions.

I As i above, but print the machine instructions in an alternate form if possible: sunsparc and mipsco reproduce the manufacturers syntax.

a Print the value in symbolic form.

Complex types

Acid permits the definition of the layout of memory. The usual method is to use the -a flag of the compilers to produce Acid-language descriptions of data structures (see *2c* (1)) although such definitions can be typed interactively. The keywords complex, adt, aggr, and union are all equivalent; the compiler uses the synonyms to document the declarations. A complex type is described as a set of members, each containing a format letter, an offset in the structure, and a name. For example, the C structure

```
struct List {
        int           type;
        struct List   *next;
};
```

is described by the Acid statement

```
complex List {
        'D'     0       type;
        'X'     4       next;
};
```

Scope

Variables are global unless they are either parameters to functions or are declared as local in a function body. Parameters and local variables are available only in the body of the function in which they are instantiated. Variables are dynamically bound: if a function declares a local variable with the same name as a global variable, the global variable will be hidden whenever the function is executing. For example, if a function f has a local called main, any function called below f will see the local version of main, not the external symbol.

Addressing

Since the symbol table specifies addresses, to access the value of program variables an extra level of indirection is required relative to the source code. For consistency, the registers are maintained as pointers as well; Acid variables with the names of processor registers point to cells holding the saved registers.

The location in a file or memory image associated with an address is calculated from a map associated with the file. Each map contains one or more quadruples (t, b, e, f), defining a segment named t (usually text, data, regs, or fpregs) mapping addresses in the range b through e to the part of the file beginning at offset f. The memory model of a Plan 9 process assumes that segments are disjoint. There can be more than one segment of a given type (e.g., a process may have more than one text segment) but segments may not overlap. An address a is translated to a file address by finding a segment for which $b \le a < e$; the location in the file is then $address + f - b$.

Usually, the text and initialized data of a program are mapped by segments called text and data. Since a program file does not contain bss, stack, or register data, these data are not mapped by the data segment. The text segment is mapped similarly in the memory image of a normal (i.e., non-kernel) process. However, the segment called *data maps memory from the beginning to the end of the program's data space. This region contains the program's static data, the bss, the heap and the stack. A segment called *regs maps the registers; *fpregs maps the floating point registers.

Sometimes it is useful to define a map with a single segment mapping the region from 0 to 0xFFFFFFFF; such a map allows the entire file to be examined without address translation. The builtin function map examines and modifies Acid's map for a process.

Name Conflicts

Name conflicts between keywords in the Acid language, symbols in the program, and previously defined functions are resolved when the interpreter starts up. Each name is made unique by prefixing enough $ characters to the front of the name to make it unique. Acid reports a list of each name change at startup. The report looks like this:

```
/bin/sam:  mips plan 9 executable
/lib/acid/port
/lib/acid/mips
Symbol renames:
        append=$append    T/0xa4e40
acid:
```

The symbol append is both a keyword and a text symbol in the program. The message reports that the text symbol is now named $append.

Expressions

Operators have the same binding and precedence as in C. For operators of equal precedence, expressions are evaluated from left to right.

Boolean expressions

If an expression is evaluated for a boolean condition the test performed depends on the type of the result. If the result is of *integer* or *floating* type the result is true if the value is non-zero. If the expression is a *list* the result is true if there are any members in the list. If the expression is a *string* the result is true if there are any characters in the string.

> *primary−expression:*
> *identifier*
> *identifier* : *identifier*
> *constant*
> (*expression*)
> { *elist* }

> *elist:*
> *expression*
> *elist* , *expression*

An identifier may be any legal Acid variable. The colon operator returns the address of parameters or local variables in the current stack frame. For example:

```
*main:argc
```

prints the number of arguments passed into main. Local variables and parameters can only be referenced after the frame has been established. It may be necessary to step a program over the first few instructions of a breakpointed function to properly set the frame.

Constants follow the same lexical rules as C. A list of expressions delimited by braces forms a list constructor. A new list is produced by evaluating each expression when the constructor is executed. The empty list is formed from {}.

```
acid:  x = 10
acid:  l = { 1, x, 2\D }
acid:  x = 20
acid:  l
{0x00000001    , 0x0000000a    , 2 }
```

Lists

Several operators manipulate lists.

> *list−expression:*
> *primary−expression*
> head *primary−expression*
> tail *primary−expression*
> append *expression* , *primary−expression*
> delete *expression* , *primary−expression*

The *primary−expression* for head and tail must yield a value of type *list*. If there are no elements in the list the value of head or tail will be the empty list. Otherwise head evaluates to the first element of the list and tail evaluates to the rest.

```
acid:  head {}
{}
acid:  head {1, 2, 3, 4}
0x00000001
acid:  tail {1, 2, 3, 4}
{0x00000002    , 0x00000003    , 0x00000004    }
```

The first operand of append and delete must be an expression that yields a *list*. Append places the result of evaluating *primary−expression* at the end of the list. The *primary−expression* supplied to delete must evaluate to an integer; delete removes the *n* th item from the list, where *n* is integral value of *primary−expression.* List indices are zero-based.

```
acid: append {1, 2}, 3
{0x00000001 , 0x00000002 , 0x00000003 }
acid: delete {1, 2, 3}, 1
{0x00000001 , 0x00000003 }
```

Assigning a list to a variable copies a reference to the list; if a list variable is copied it still points at the same list. To copy a list, the elements must be copied piecewise using `head` and `append`.

Operators

postfix−expression:
  list−expression
  postfix−expression [*expression*]
  postfix−expression (*argument−list*)
  postfix−expression . *tag*
  postfix−expression -> *tag*
  postfix−expression ++
  postfix−expression --

argument−list:
  expression
  argument−list , *expression*

The [*expression*] operator performs indexing. The indexing expression must result in an expression of *integer* type, say *n*. The operation depends on the type of *postfix−expression*. If the *postfix−expression* yields an *integer* it is assumed to be the base address of an array in the memory image. The index offsets into this array; the size of the array members is determined by the format associated with the *postfix−expression*. If the *postfix−expression* yields a *string* the index operator fetches the *n* th character of the string. If the index points beyond the end of the string, a zero is returned. If the *postfix−expression* yields a *list* then the indexing operation returns the *n* th item of the list. If the list contains less than *n* items the empty list {} is returned.

The ++ and -- operators increment and decrement integer variables. The amount of increment or decrement depends on the format code. These postfix operators return the value of the variable before the increment or decrement has taken place.

unary−expression:
  postfix−expression
  ++ *unary−expression*
  -- *unary−expression*

unary−operator: one of
  * @ + - ~ !

The operators * and @ are the indirection operators. @ references a value from the text file of the program being debugged. The size of the value depends on the format code. The * operator fetches a value from the memory image of a process. If either operator appears on the left-hand side of an assignment statement, either the file or memory will be written. The file can only be modified when Acid is invoked with the -w option. The prefix ++ and -- operators perform the same operation as their postfix counterparts but return the value after the increment or decrement has been performed. Since the ++ and * operators fetch and increment the correct amount for the specified format, the following function prints correct machine instructions on a machine with variable length instructions, such as the 68020 or 386:

```
defn asm(addr)
{
        addr = fmt(addr, 'i');
        loop 1, 10 do
                print(*addr++, "\n");
}
```

The operators ~ and ! perform bitwise and logical negation respectively. Their operands must be of *integer* type.

cast-expression:
    unary-expression
    unary-expression \ *format-char*
    (*complex-name*) *unary-expression*

A unary expression may be preceded by a cast. The cast has the effect of associating the value of *unary-expression* with a complex type structure. The result may then be dereferenced using the . and -> operators.

An Acid variable may be associated with a complex type to enable accessing the types members:

```
acid: complex List {
        'D'     0       type;
        'X'     4       next;
};
acid: complex List lhead
acid: lhead.type
10
acid: lhead = ((List)lhead).next
acid: lhead.type
-46
```

Note that the next field cannot be given a complex type automatically.

When entered at the top level of the interpreter, an expression of complex type is treated specially. If the type is called T and an Acid function also called T exists, then that function will be called with the expression as its argument. The compiler options -a and -aa will generate Acid source code defining such complex types and functions; see *2c*(1).

A *unary-expression* may be qualified with a format specifier using the \ operator. This has the same effect as passing the expression to the fmt builtin function.

multiplicative-expression:
    cast-expression
    multiplicative-expression * *multiplicative-expression*
    multiplicative-expression / *multiplicative-expression*
    multiplicative-expression % *multiplicative-expression*

These operate on *integer* and *float* types and perform the expected operations: * multiplication, / division, % modulus.

additive-expression:
    multiplicative-expression
    additive-expression + *multiplicative-expression*
    additive-expression - *multiplicative-expression*

These operators perform as expected for *integer* and *float* operands. Unlike in C, + and - do not scale the addition based on the format of the expression. This means that i=i+1 will always add 1 but i++ will add the size corresponding to the format stored with i. If both operands are of either *string* or *list* type then addition is defined as concatenation. Adding a string and an integer is treated as concatenation with the Unicode character corresponding to the integer. Subtraction is undefined for strings and lists.

shift-expression:
    additive-expression
    shift-expression << *additive-expression*
    shift-expression >> *additive-expression*

The >> and << operators perform bitwise right and left shifts respectively. Both require operands of *integer* type.

relational-expression:
    relational-expression < *shift-expression*
    relational-expression > *shift-expression*
    relational-expression <= *shift-expression*
    relational-expression >= *shift-expression*

equality-expression:
    relational-expression
    relational-expression == *equality-expression*
    relational-expression != *equality-expression*

The comparison operators are < (less than), > (greater than), <= (less than or equal to), >= (greater than or equal to), == (equal to) and != (not equal to). The result of a comparison is 0 if the condition is false, otherwise 1. The relational operators can only be applied to operands of *integer* and *float* type. The equality operators apply to all types. Comparing mixed types is legal. Mixed integer and float compare on the integral value. Other mixtures are always unequal. Two lists are equal if they have the same number of members and a pairwise comparison of the members results in equality.

AND-expression:
    equality-expression
    AND-expression & *equality-expression*

XOR-expression:
    AND-expression
    XOR-expression ^ *AND-expression*

OR-expression:
    XOR-expression
    OR-expression | *XOR-expression*

These operators perform bitwise logical operations and apply only to the *integer* type. The operators are & (logical and), ^ (exclusive or) and | (inclusive or).

logical-AND-expression:
    OR-expression
    logical-AND-expression && *OR-expression*

logical-OR-expression:
    logical-AND-expression
    logical-OR-expression || *logical-AND-expression*

The && operator returns 1 if both of its operands evaluate to boolean true, otherwise 0. The || operator returns 1 if either of its operands evaluates to boolean true, otherwise 0.

Statements

> `if` *expression* `then` *statement* `else` *statement*
> `if` *expression* `then` *statement*

The *expression* is evaluated as a boolean. If its value is true the statement after the `then` is executed, otherwise the statement after the `else` is executed. The `else` portion may be omitted.

> `while` *expression* `do` *statement*

In a while loop, *statement* is executed while the boolean *expression* evaluates true.

> `loop` *startexpr, endexpr* `do` *statement*

The two expressions *startexpr* and *endexpr* are evaluated prior to loop entry. *Statement* is evaluated while the value of *startexpr* is less than or equal to *endexpr*. Both expressions must yield *integer* values. The value of *startexpr* is incremented by one for each loop iteration. Note that there is no explicit loop variable; the *expressions* are just values.

> `return` *expression*

`return` terminates execution of the current function and returns to its caller. The value of the function is given by expression. Since `return` requires an argument, nil-valued functions should return the empty list `{}`.

> `local` *variable*

The `local` statement creates a local instance of *variable*, which exists for the duration of the instance of the function in which it is declared. Binding is dynamic: the local variable, rather than the previous value of *variable*, is visible to called functions. After a return from the current function the previous value of *variable* is restored.

If Acid is interrupted, the values of all local variables are lost, as if the function returned.

> `defn` *function-name* `(` *parameter-list* `)` *body*
>
> *parameter-list:*
>   *variable*
>   *parameter-list* , *variable*
>
> *body:*
>   `{` *statement* `}`

Functions are introduced by the `defn` statement. The definition of parameter names suppresses any variables of the same name until the function returns. The body of a function is a list of statements enclosed by braces.

Code variables

Acid permits the delayed evaluation of a parameter to a function. The parameter may then be evaluated at any time with the `eval` operator. Such parameters are called *code variables* and are defined by prefixing their name with an asterisk in their declaration.

For example, this function wraps up an expression for later evaluation:

```
acid: defn code(*e) { return e; }
acid: x = code(v+atoi("100")\D)
acid: print(x)
(v+atoi("100"))\D;
acid: eval x
<stdin>:5: (error) v used but not set
acid: v=5
acid: eval x
105
```

Source Code Management

Acid provides the means to examine source code. Source code is represented by lists of strings. Builtin functions provide mapping from address to lines and vice-versa. The default debugging environment has the means to load and display source files.

Builtin Functions

The Acid interpreter has a number of builtin functions, which cannot be redefined. These functions perform machine- or operating system-specific functions such as symbol table and process management. The following section presents a description of each builtin function. The notation {} is used to denote the empty list, which is the default value of a function that does not execute a return statement. The type and number of parameters for each function are specified in the description; where a parameter can be of any type it is specified as type *item*.

integer access(*string*) Check if a file can be read

Access returns the integer 1 if the file name in *string* can be read by the builtin functions file, readfile, or include, otherwise 0. A typical use of this function is to follow a search path looking for a source file; findsrc is used by

```
if access("main.c") then
    return file("main.c");
```

float atof(*string*) Convert a string to float

atof converts the string supplied as its argument into a floating point number. The function accepts strings in the same format as the C function of the same name. The value returned has the format code f. atof returns the value 0.0 if it is unable to perform the conversion.

```
acid: +atof("10.4e6")
1.04e+07
```

integer atoi(*string*) Convert a string to an integer

atoi converts the argument to an integer value. The function accepts strings in the same format as the C function of the same name. The value returned has the format code D. atoi returns the integer 0 if it is unable to perform a conversion.

```
acid: +atoi("-1255")
-1255
```

{} error(*string*) Generate an interpreter error

error generates an error message and returns the interpreter to interactive mode. If an Acid program is running, it is aborted. Processes being debugged are not affected. The values of all local variables are lost. error is commonly used to stop the debugger when some interesting condition arises in the debugged program.

```
while 1 do {
    step();
    if *main != @main then
        error("memory corrupted");
}
```

list file(*string*) Read the contents of a file into a list

file reads the contents of the file specified by *string* into a list. Each element in the list is a string corresponding to a line in the file. file breaks lines at the newline character, but the newline characters are not returned as part each string. file returns the empty list if it encounters an error opening or reading the data.

```
acid: print(file("main.c")[0])
#include  <u.h>
```

integer filepc(*string*) Convert source address to text address

filepc interprets its *string* argument as a source file address in the form of a file name and line offset. filepc uses the symbol table to map the source address into a text address in the debugged program. The *integer* return value has the format X. filepc returns an address of -1 if the source address is invalid. The source file address uses the same format as *acme*(1). This function is commonly used to set breakpoints from the source text.

```
acid: bpset(filepc("main:10"))
acid: bptab()
    0x00001020 usage      ADD   $-0xc,R29
```

item fmt(*item,fmt*) Set print, @ and * formats

fmt evaluates the expression *item* and sets the format of the result to *fmt*. The format of a value determines how it will be printed and what kind of object will be fetched by the * and @ operators. The \ operator is a short-hand form of the fmt builtin function. fmt leaves the format of the *item* unchanged.

```
acid: main=fmt(main, 'i') // as instructions
acid: print(main\X, "\t", *main)
0x00001020 ADD      $-64,R29
```

fmt fmtof(*item*) Get format

fmtof evaluates the expression *item* and returns the format of the result.

```
acid: +fmtof(33)
W
acid: +fmtof("string")
s
```

integer fmtsize(*item*) Get format size

fmtsize evaluates the expression *item* and returns the size in bytes of a single element of result's format.

```
acid: +fmtsize('c')
8
acid: +fmtsize('c'\c)
1
acid: +fmtsize(0\X)
4
acid: +fmtsize('c'\3)
10
```

list fnbound(*integer*) — Find start and end address of a function

fnbound interprets its *integer* argument as an address in the text of the debugged program. fnbound returns a list containing two integers corresponding to the start and end addresses of the function containing the supplied address. If the *integer* address is not in the text segment of the program then the empty list is returned. fnbound is used by next to detect stepping into new functions.

```
acid: print(fnbound(main))
{0x00001050, 0x000014b8}
```

{} follow(*integer*) — Compute follow set

The follow set is defined as the set of program counter values that could result from executing an instruction. follow interprets its *integer* argument as a text address, decodes the instruction at that address and, with the current register set, builds a list of possible next program counter values. If the instruction at the specified address cannot be decoded follow raises an error. follow is used to plant breakpoints on all potential paths of execution. The following code fragment plants breakpoints on top of all potential following instructions.

```
lst = follow(*PC);
while lst do
{
    *head lst = bpinst;
    lst = tail lst;
}
```

{} include(*string*) — Take input from a new file

include opens the file specified by *string* and uses its contents as command input to the interpreter. The interpreter restores input to its previous source when it encounters either an end of file or an error. include can be used to incrementally load symbol table information without leaving the interpreter.

```
acid: include("/sys/src/cmd/acme/syms")
```

{} interpret(*string*) — Take input from a string

interpret evaluates the *string* expression and uses its result as command input for the interpreter. The interpreter restores input to its previous source when it encounters either the end of string or an error. The interpret function allows Acid programs to write Acid code for later evaluation.

```
acid: interpret("main+10;")
0x0000102a
```

string itoa(*integer[,string]*) — Convert integer to string

itoa takes an integer argument and converts it into an ASCII string in the D format. an alternate format string may be provided in the % style of *print* (2). This function is commonly used to build rc command lines.

```
acid: rc("cat /proc/"+itoa(pid)+"/segment")
Stack      7fc00000 80000000        1
Data       00001000 00009000        1
Data       00009000 0000a000        1
Bss        0000a000 0000c000        1
```

{} kill(*integer*) Kill a process

kill writes a kill control message into the control file of the process specified by the *integer* pid. If the process was previously installed by setproc it will be removed from the list of active processes. If the *integer* has the same value as pid , then pid will be set to 0. To continue debugging, a new process must be selected using setproc . For example, to kill all the active processes:

```
while proclist do {
    kill(head proclist);
    proclist = tail proclist;
}
```

list map(*list*) Set or retrieve process memory map

map either retrieves all the mappings associated with a process or sets a single map entry to a new value. If the *list* argument is omitted then map returns a list of lists. Each sublist has four values and describes a single region of contiguous addresses in the memory or file image of the debugged program. The first entry is the name of the mapping. If the name begins with * it denotes a map into the memory of an active process. The second and third values specify the base and end address of the region and the fourth number specifies the offset in the file corresponding to the first location of the region. A map entry may be set by supplying a list in the same format as the sublist described above. The name of the mapping must match a region already defined by the current map. Maps are set automatically for Plan 9 processes and some kernels; they may need to be set by hand for other kernels and programs that run on bare hardware.

```
acid: map({"text", _start, end, 0x30})
```

integer match(*item,list*) Search list for matching value

match compares each item in *list* using the equality operator == with *item* . The *item* can be of any type. If the match succeeds the result is the integer index of the matching value, otherwise -1.

```
acid: list={8,9,10,11}
acid: print(list[match(10, list)]\D)
10
```

{} newproc(*string*) Create a new process

newproc starts a new process with an argument vector constructed from *string* . The argument vector excludes the name of the program to execute and each argument in *string* must be space separated. A new process can accept no more than 512 arguments. The internal variable pid is set to the pid of the newly created process. The new pid is also appended to the list of active processes stored in the variable proclist . The new process is created then halted at the first instruction, causing the debugger to call stopped . The library functions new and win should be used to start processes when using the standard debugging environment.

```
acid: newproc("-l .")
56720: system call      _main  ADD    $-0x14,R29
```

string pcfile(*integer*) Convert text address to source file name

pcfile interprets its *integer* argument as a text address in the debugged program. The address and symbol table are used to generate a string containing the name of the source file corresponding to the text address. If the address does not lie within the program the string ?file? is returned.

```
acid: print("Now at ", pcfile(*PC), ":", pcline(*PC))
Now at ls.c:46
```

integer pcline(*integer*) Convert text address to source line number

pcline interprets its *integer* argument as a text address in the debugged program. The address and symbol table are used to generate an integer containing the line number in the source file corresponding to the text address. If the address does not lie within the program the integer 0 is returned.

```
acid: +file("main.c")[pcline(main)]
main(int argc, char *argv[])
```

{} print(*item,item,...*) Print expressions

print evaluates each *item* supplied in its argument list and prints it to standard output. Each argument will be printed according to its associated format character. When the interpreter is executing, output is buffered and flushed every 5000 statements or when the interpreter returns to interactive mode. print accepts a maximum of 512 arguments.

```
acid: print(10, "decimal ", 10\D, "octal ", 10\o)
0x0000000a decimal 10 octal 000000000012
acid: print({1, 2, 3})
{0x00000001 , 0x00000002 , 0x00000003 }
acid: print(main, main\a, "\t", @main\i)
0x00001020 main     ADD    $-64,R29
```

{} printto(*string,item,item,...*) Print expressions to file

printto offers a limited form of output redirection. The first *string* argument is used as the path name of a new file to create. Each *item* is then evaluated and printed to the newly created file. When all items have been printed the file is closed. printto accepts a maximum of 512 arguments.

```
acid: printto("/env/foo", "hello")
acid: rc("echo -n $foo")
hello
```

string rc(*string*) Execute a shell command

rc evaluates *string* to form a shell command. A new command interpreter is started to execute the command. The Acid interpreter blocks until the command completes. The return value is the empty string if the command succeeds, otherwise the exit status of the failed command.

```
acid: rc("B "+itoa(-pcline(addr))+" "+pcfile(addr));
```

string readfile(*string*) Read file contents into a string

readfile takes the contents of the file specified by *string* and returns its contents as a new string. If readfile encounters a zero byte in the file, it terminates. If readfile encounters an error opening or reading the file then the empty list is returned. readfile can be used to read the contents of device files whose lines are not terminated with newline characters.

```
acid: ""+readfile("/dev/label")
helix
```

string reason(*integer*) Print cause of program stoppage

reason uses machine-dependent information to generate a string explaining why a process has stopped. The *integer* argument is the value of an architecture dependent status register, for example CAUSE on the MIPS.

```
acid: print(reason(*CAUSE))
system call
```

integer regexp(*pattern,string*) — Regular expression match

regexp matches the *pattern* string supplied as its first argument with the *string* supplied as its second. If the pattern matches the result is the value 1, otherwise 0.

```
acid: print(regexp(".*bar", "foobar"))
1
```

{} setproc(*integer*) — Set debugger focus

setproc selects the default process used for memory and control operations. It effectively shifts the focus of control between processes. The *integer* argument specifies the pid of the process to look at. The variable pid is set to the pid of the selected process. If the process is being selected for the first time its pid is added to the list of active processes proclist .

```
acid: setproc(68382)
acid: procs()
>68382: Stopped at main+0x4 setproc(68382)
```

{} start(*integer*) — Restart execution

start writes a start message to the control file of the process specified by the pid supplied as its *integer* argument. start draws an error if the process is not in the Stopped state.

```
acid: start(68382)
acid: procs()
>68382: Running at main+0x4 setproc(68382)
```

{} startstop(*integer*) — Restart execution, block until stopped

startstop performs the same actions as a call to start followed by a call to stop . The *integer* argument specifies the pid of the process to control. The process must be in the Stopped state. Execution is restarted, the debugger then waits for the process to return to the Stopped state. A process will stop if a startstop message has been written to its control file and any of the following conditions becomes true: the process executes or returns from a system call, the process generates a trap or the process receives a note. startstop is used to implement single stepping.

```
acid: startstop(pid)
75374: breakpoint      ls  ADD    $-0x16c8,R29
```

string status(*integer*) — Return process state

status uses the pid supplied by its *integer* argument to generate a string describing the state of the process. The string corresponds to the state returned by the sixth column of the *ps*(1) command. A process must be in the Stopped state to modify its memory or registers.

```
acid: ""+status(pid)
Stopped
```

{} stop(*integer*) Wait for a process to stop

stop writes a stop message to the control file of the process specified by the pid supplied as its *integer* argument. The interpreter blocks until the debugged process enters the Stopped state. A process will stop if a stop message has been written to its control file and any of the following conditions becomes true: the process executes or returns from a system call, the process generates a trap, the process is scheduled or the process receives a note. stop is used to wait for a process to halt before planting a breakpoint since Plan 9 only allows a process's memory to be written while it is in the Stopped state.

```
defn bpset(addr) {
    if (status(pid)!="Stopped") then {
        print("Waiting...\n");
        stop(pid);
    }
    ...
}
```

list strace(*pc,sp,linkreg*) Stack trace

strace generates a list of lists corresponding to procedures called by the debugged program. Each sublist describes a single stack frame in the active process. The first element is an *integer* of format X specifying the address of the called function. The second element is the value of the program counter when the function was called. The third and fourth elements contain lists of parameter and automatic variables respectively. Each element of these lists contains a string with the name of the variable and an *integer* value of format X containing the current value of the variable. The arguments to strace are the current value of the program counter, the current value of the stack pointer, and the address of the link register. All three parameters must be integers. The setting of *linkreg* is architecture dependent. On the MIPS linkreg is set to the address of saved R31, on the SPARC to the address of saved R15. For the other architectures *linkreg* is not used, but must point to valid memory.

```
acid: print(strace(*PC, *SP, linkreg))
{{0x0000141c, 0xc0000f74,
{{"s", 0x0000004d}, {"multi", 0x00000000}},
{{"db", 0x00000000}, {"fd", 0x000010a4},
{"n", 0x00000001}, {"i", 0x00009824}}}}
```

{} waitstop(*integer*) Wait for a process to stop

waitstop writes a waitstop message to the control file of the process specified by the pid supplied as its *integer* argument. The interpreter will remain blocked until the debugged process enters the Stopped state. A process will stop if a waitstop message has been written to its control file and any of the following conditions becomes true: the process generates a trap or receives a note. Unlike stop, the waitstop function is passive; it does not itself cause the program to stop.

```
acid: waitstop(pid)
75374: breakpoint       ls ADD    $-0x16c8,R29
```

Library Functions

A standard debugging environment is provided by modules automatically loaded when Acid is started. These modules are located in the directory /sys/lib/acid. 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.

{} Bsrc(*integer*) Load editor with source

Bsrc interprets the *integer* argument as a text address. The text address is used to produce a pathname and line number suitable for the B command to send to the text editor *sam* (1) or *acme* (1). Bsrc builds an *rc* (1) command to invoke B, which either selects an existing source file or loads a new source file into the editor. The line of source corresponding to the text address is then selected. In the following example stopped is redefined so that the editor follows and displays the source line currently being executed.

```
defn stopped(pid) {
   pstop(pid);
   Bsrc(*PC);
}
```

{} Fpr() Display double precision floating registers

For machines equipped with floating point, Fpr displays the contents of the floating point registers as double precision values.

```
acid: Fpr()
F0    0.  F2    0.
F4    0.  F6    0.
F8    0.  F10   0.
...
```

{} Ureg(*integer*) Display contents of Ureg structure

Ureg interprets the integer passed as its first argument as the address of a kernel Ureg structure. Each element of the structure is retrieved and printed. The size and contents of the Ureg structure are architecture dependent. This function can be used to decode the first argument passed to a *notify* (2) function after a process has received a note.

```
acid: Ureg(*notehandler:ur)
   status    0x3000f000
   pc 0x1020
   sp 0x7ffffe00
   cause  0x00004002
...
```

{} acidinit() Interpreter startup

acidinit is called by the interpreter after all modules have been loaded at initialization time. It is used to set up machine specific variables and the default source path. acidinit should not be called by user code.

{} addsrcdir(*string*) Add element to source search path

addsrcdir interprets its string argument as a new directory findsrc should search when looking for source code files. addsrcdir draws an error if the directory is already in the source search path. The search path may be examined by looking at the variable srcpath.

```
acid: rc("9fs fornax")
acid: addsrcpath("/n/fornax/sys/src/cmd")
```

{} asm(*integer*) Disassemble machine instructions

asm interprets its integer argument as a text address from which to disassemble machine instructions. asm prints the instruction address in symbolic and hexadecimal form, then prints the instructions with addressing modes. Up to twenty instructions will be disassembled. asm stops disassembling when it reaches the end of the current function. Instructions are read from the file image using the @ operator.

```
acid: asm(main)
main       0x00001020 ADD       $-0x64,R29
main+0x4 0x00001024 MOVW       R31,0x0(R29)
main+0x8 0x00001028 MOVW       R1,argc+4(FP)
main+0xc 0x0000102c MOVW       $bin(SB),R1
```

{} bpdel(*integer*) Delete breakpoint

bpdel removes a previously set breakpoint from memory. The *integer* supplied as its argument must be the address of a previously set breakpoint. The breakpoint address is deleted from the active breakpoint list bplist , then the original instruction is copied from the file image to the memory image so that the breakpoint is removed.

```
acid: bpdel(main+4)
```

{} bpset(*integer*) Set a breakpoint

bpset places a breakpoint instruction at the address specified by its *integer* argument, which must be in the text segment. bpset draws an error if a breakpoint has already been set at the specified address. A list of current breakpoints is maintained in the variable bplist . Unlike in *db* (1), breakpoints are left in memory even when a process is stopped, and the process must exist, perhaps by being created by either new or win , in order to place a breakpoint. (Db accepts breakpoint commands before the process is started.) On the MIPS and SPARC architectures, breakpoints at function entry points should be set 4 bytes into the function because the instruction scheduler may fill JAL branch delay slots with the first instruction of the function.

```
acid: bpset(main+4)
```

{} bptab() List active breakpoints

bptab prints a list of currently installed breakpoints. The list contains the breakpoint address in symbolic and hexadecimal form as well as the instruction the breakpoint replaced. Breakpoints are not maintained across process creation using new and win . They are maintained across a fork, but care must be taken to keep control of the child process.

```
acid: bpset(ls+4)
acid: bptab()
    0x00001420 ls+0x4      MOVW   R31,0x0(R29)
```

{} casm() Continue disassembly

casm continues to disassemble instructions from where the last asm or casm command stopped. Like asm , this command stops disassembling at function boundaries.

```
acid: casm()
main+0x10 0x00001030      MOVW   $0x1,R3
main+0x14 0x00001034      MOVW   R3,0x8(R29)
main+0x18 0x00001038      MOVW   $0x1,R5
main+0x1c 0x0000103c      JAL    Binit(SB)
```

{} cont() Continue program execution

cont restarts execution of the currently active process. If the process is stopped on a breakpoint, the breakpoint is first removed, the program is single stepped, the breakpoint is replaced and the program is then set executing. This may cause stopped() to be called twice. cont causes the interpreter to block until the process enters the Stopped state.

```
acid: cont()
95197: breakpoint       ls+0x4    MOVW   R31,0x0(R29)
```

{} dump(*integer,integer,string*) Formatted memory dump

dump interprets its first argument as an address, its second argument as a count and its third as a format string. dump fetches an object from memory at the current address and prints it according to the format. The address is incremented by the number of bytes specified by the format and the process is repeated count times. The format string is any combination of format characters, each preceded by an optional count. For each object, dump prints the address in hexadecimal, a colon, the object and then a newline. dump uses mem to fetch each object.

```
acid: dump(main+35, 4, "X2bi")
0x00001043: 0x0c8fa700 108 143 lwc2 r0,0x528f(R4)
0x0000104d: 0xa9006811       0    0 swc3 r0,0x0(R24)
0x00001057: 0x2724e800       4   37 ADD    $-0x51,R23,R31
0x00001061: 0xa200688d       6    0 NOOP
0x0000106b: 0x2710c000       7    0 BREAK
```

{} findsrc(*string*) Use source path to load source file

findsrc interprets its *string* argument as a source file. Each directory in the source path is searched in turn for the file. If the file is found, the source text is loaded using file and stored in the list of active source files called srctext. The name of the file is added to the source file name list srcfiles. Users are unlikely to call findsrc from the command line, but may use it from scripts to preload source files for a debugging session. This function is used by src and line to locate and load source code. The default search path for the MIPS is ./, /sys/src/libc/port, /sys/src/libc/9sys, /sys/src/libc/mips.

```
acid: findsrc(pcfile(main));
```

{} fpr() Display single precision floating registers

For machines equipped with floating point, fpr displays the contents of the floating point registers as single precision values. When the interpreter stores or manipulates floating point values it converts into double precision values.

```
acid: fpr()
F0    0.  F1    0.
F2    0.  F3    0.
F4    0.  F5    0.
...
```

{} func() Step while in function

func single steps the active process until it leaves the current function by either calling another function or returning to its caller. func will execute a single instruction after leaving the current function.

```
acid: func()
95197: breakpoint       ls+0x8    MOVW   R1,R8
95197: breakpoint       ls+0xc    MOVW   R8,R1
95197: breakpoint       ls+0x10   MOVW   R8,s+4(FP)
95197: breakpoint       ls+0x14   MOVW   $0x2f,R5
95197: breakpoint       ls+0x18   JAL    utfrrune(SB)
95197: breakpoint       utfrrune  ADD    $-0x18,R29
```

{} gpr() Display general purpose registers

gpr prints the values of the general purpose processor registers.

```
acid: gpr()
R1  0x00009562 R2    0x000010a4 R3    0x00005d08
R4  0x0000000a R5    0x0000002f R6    0x00000008
...
```

{} labstk(*integer*) Print stack trace from label

labstk performs a stack trace from a Plan 9 *label.* The kernel, C compilers store continuations in a common format. Since the compilers all use caller save conventions a continuation may be saved by storing a PC and SP pair. This data structure is called a label and is used by the the C function longjmp and the kernel to schedule threads and processes. labstk interprets its *integer* argument as the address of a label and produces a stack trace for the thread of execution. The value of the function ALEF_tid is a suitable argument for labstk.

```
acid: labstk(*mousetid)
At pc:0x00021a70:Rendez_Sleep+0x178 rendez.l:44
Rendez_Sleep(r=0xcd7d8,bool=0xcd7e0,t=0x0) rendez.l:5
   called from ALEF_rcvmem+0x198 recvmem.l:45
ALEF_rcvmem(c=0x000cd764,l=0x00000010) recvmem.l:6
...
```

{} lstk() Stack trace with local variables

lstk produces a long format stack trace. The stack trace includes each function in the stack, where it was called from, and the value of the parameters and automatic variables for each function. lstk displays the value rather than the address of each variable and all variables are assumed to be an integer in format X. To print a variable in its correct format use the : operator to find the address and apply the appropriate format before indirection with the * operator. It may be necessary to single step a couple of instructions into a function to get a correct stack trace because the frame pointer adjustment instruction may get scheduled down into the body of the function.

```
acid: lstk()
At pc:0x00001024:main+0x4 ls.c:48
main(argc=0x00000001,argv=0x7fffefec) ls.c:48
   called from _main+0x20 main9.s:10
   _argc=0x00000000
   _args=0x00000000
   fd=0x00000000
   buf=0x00000000
   i=0x00000000
```

{} mem(*integer,string*) Print memory object

mem interprets its first *integer* argument as the address of an object to be printed according to the format supplied in its second *string* argument. The format string can be any combination of format characters, each preceded by an optional count.

```
acid: mem(bdata+0x326, "2c2Xb")
P = 0xa94bc464 0x3e5ae44d       19
```

{} new() Create new process

new starts a new copy of the debugged program. The new program is started with the program arguments set by the variable progargs. The new program is stopped in the second instruction of main. The breakpoint list is reinitialized. new may be used several times to instantiate several copies of a program simultaneously. The user can rotate between the copies using setproc.

```
acid: progargs="-l"
acid: new()
60: external interrupt     _main  ADD    $-0x14,R29
60: breakpoint      main+0x4  MOVW   R31,0x0(R29)
```

{} next() Step through language statement

next steps through a single language level statement without tracing down through each statement in a called function. For each statement, next prints the machine instructions executed as part of the statement. After the statement has executed, source lines around the current program counter are displayed.

```
acid: next()
60: breakpoint      Binit+0x4 MOVW      R31,0x0(R29)
60: breakpoint      Binit+0x8 MOVW      f+8(FP),R4
binit.c:93
 88
 89   int
 90   Binit(Biobuf *bp, int f, int mode)
 91   {
>92       return Binits(bp, f, mode, bp->b, BSIZE);
 93   }
```

{} notestk(*integer*) Stack trace after receiving a note

notestk interprets its *integer* argument as the address of a Ureg structure passed by the kernel to a *notify* (2) function during note processing. notestk uses the PC, SP, and link register from the Ureg to print a stack trace corresponding to the point in the program where the note was received. To get a valid stack trace on the MIPS and SPARC architectures from a notify routine, the program must stop in a new function called from the notify routine so that the link register is valid and the notify parameters are addressable.

```
acid: notestk(*notify:ur)
Note pc:0x00001024:main+0x4 ls.c:48
main(argc=0x00000001,argv=0x7fffefec) ls.c:48
    called from _main+0x20 main9.s:10
    _argc=0x00000000
    _args=0x00000000
```

{} pfl(*integer*) Print source file and line

pfl interprets its argument as a text address and uses it to print the source file and line number corresponding to the address. The output has the same format as file addresses in *acme* (1).

```
acid: pfl(main)
ls.c:48
```

{} procs() Print active process list

procs prints a list of active process attached to the debugger. Each process produces a single line of output giving the pid, process state, the address the process is currently executing, and the setproc command required to make that process current. The current process is marked in the first column with a > character. The debugger maintains a list of processes in the proclist variable.

```
acid: procs()
>62: Stopped at main+0x4 setproc(62)
 60: Stopped at Binit+0x8 setproc(60)
```

{} pstop(*integer*) Print reason process stopped

pstop prints the status of the process specified by the *integer* pid supplied as its argument. pstop is usually called from stopped every time a process enters the Stopped state.

```
acid: pstop(62)
0x0000003e: breakpoint     main+0x4  MOVW   R31,0x0(R29)
```

{} regs() Print registers

regs prints the contents of both the general and special purpose registers. regs calls spr then gpr to display the contents of the registers.

{} source() Summarize source data base

source prints the directory search path followed by a list of currently loaded source files. The source management functions src and findsrc use the search path to locate and load source files. Source files are loaded incrementally into a source data base during debugging. A list of loaded files is stored in the variable srcfiles and the contents of each source file in the srctext variable.

```
acid: source()
/n/bootes/sys/src/libbio/
/sys/src/libc/port/
/sys/src/libc/9sys/
/sys/src/libc/mips/
    binit.c
```

{} spr() Print special purpose registers

spr prints the contents of the processor control and memory management registers. Where possible, the contents of the registers are decoded to provide extra information; for example the CAUSE register on the MIPS is printed both in hexadecimal and using the reason function.

```
acid: spr()
PC 0x00001024 main+0x4      ls.c:48
SP 0x7fffef68 LINK     0x00006264 _main+0x28 main9.s:12
STATUS    0x0000ff33 CAUSE    0x00000024 breakpoint
TLBVIR    0x000000d3 BADVADR     0x00001020
HI 0x00000004 LO       0x00001ff7
```

{} src(*integer*) Print lines of source

src interprets its *integer* argument as a text address and uses this address to print 5 lines of source before and after the address. The current line is marked with a > character. src uses the source search path maintained by source and addsrcdir to locate the required source files.

```
acid: src(*PC)
ls.c:47
 42   Biobuf    bin;
 43
 44   #define       HUNK   50
 45
 46   void
>47   main(int argc, char *argv[])
 48   {
 49       int i, fd;
 50       char buf[64];
 51
 52       Binit(&bin, 1, OWRITE);
```

{} step() Single step process

step causes the debugged process to execute a single machine level instruction. If the program is stopped on a breakpoint set by bpset it is first removed, the single step executed, and the breakpoint replaced. step uses follow to predict the address of the program counter after the current instruction has been executed. A breakpoint is placed at each of these predicted addresses and the process is started. When the process stops the breakpoints are removed.

```
acid: step()
62: breakpoint      main+0x8  MOVW   R1,argc+4(FP)
```

{} stk() Stack trace

stk produces a short format stack trace. The stack trace includes each function in the stack, where it was called from, and the value of the parameters. The short format omits the values of automatic variables. Parameters are assumed to be integer values in the format X; to print a parameter in the correct format use the : to obtain its address, apply the correct format, and use the * indirection operator to find its value. It may be necessary to single step a couple of instructions into a function to get a correct stack trace because the frame pointer adjustment instruction may get scheduled down into the body of the function.

```
acid: stk()
At pc:0x00001028:main+0x8 ls.c:48
main(argc=0x00000002,argv=0x7fffefe4) ls.c:48
    called from _main+0x20 main9.s:10
```

{} stmnt() Execute a single statement

stmnt executes a single language level statement. stmnt displays each machine level instruction as it is executed. When the executed statement is completed the source for the next statement is displayed. Unlike next, the stmnt function will trace down through function calls.

```
acid: stmnt()
62: breakpoint      main+0x18 MOVW      R5,0xc(R29)
62: breakpoint      main+0x1c JAL    Binit(SB)
62: breakpoint      Binit       ADD  $-0x18,R29
binit.c:91
 89   int
 90   Binit(Biobuf *bp, int f, int mode)
>91   {
```

{} stopped(*integer*) Report status of stopped process

stopped is called automatically by the interpreter every time a process enters the Stopped state, such as when it hits a breakpoint. The pid is passed as the *integer* argument. The default implementation just calls pstop , but the function may be changed to provide more information or perform fine control of execution. Note that stopped should return; for example, calling step in stopped will recur until the interpreter runs out of stack space.

```
acid: defn stopped(pid) {
    if *lflag != 0 then error("lflag modified");
    }
acid: progargs = "-l"
acid: new();
acid: while 1 do step();
<stdin>:7: (error) lflag modified
acid: stk()
At pc:0x00001220:main+0x200 ls.c:54
main(argc=0x00000001,argv=0x7fffffe8) ls.c:48
    called from _main+0x20 main9.s:10
```

{} symbols(*string*) Search symbol table

symbols uses the regular expression supplied by *string* to search the symbol table for symbols whose name matches the regular expression.

```
acid: symbols("main")
main   T   0x00001020
_main  T   0x0000623c
```

{} win() Start new process in a window

win performs exactly the same function as new but uses the window system to create a new window for the debugged process. The variable progargs supplies arguments to the new process. The environment variable $8½srv must be set to allow the interpreter to locate the mount channel for the window system. The window is created in the top left corner of the screen and is 400x600 pixels in size. The win function may be modified to alter the geometry. The window system will not be able to deliver notes in the new window since the pid of the created process is not passed when the server is mounted to create a new window.

```
acid: win()
```

Maintaining Files on Plan 9 with Mk

Andrew G. Hume
andrew@research.att.com
Bob Flandrena
bobf@plan9.bell-labs.com

ABSTRACT

Mk is a tool for describing and maintaining dependencies between files. It is similar to the UNIX program make, but provides several extensions. Mk's flexible rule specifications, implied dependency derivation, and parallel execution of maintenance actions are well-suited to the Plan 9 environment. Almost all Plan 9 maintenance procedures are automated using mk.

1. Introduction

This document describes how mk, a program functionally similar to make [Feld79], is used to maintain dependencies between files in Plan 9. Mk provides several extensions to the capabilities of its predecessor that work well in Plan 9's distributed, multi-architecture environment. It exploits the power of multiprocessors by executing maintenance actions in parallel and interacts with the Plan 9 command interpreter rc to provide a powerful set of maintenance tools. It accepts pattern-based dependency specifications that are not limited to describing rules for program construction. The result is a tool that is flexible enough to perform many maintenance tasks including database maintenance, hardware design, and document production.

This document begins by discussing the syntax of the control file, the pattern matching capabilities, and the special rules for maintaining archives. A brief description of mk's algorithm for deriving dependencies is followed by a discussion of the conventions used to resolve ambiguous specifications. The final sections describe parallel execution and special features.

An earlier paper [Hume87] provides a detailed discussion of mk's design and an appendix summarizes the differences between mk and make.

2. The `Mkfile`

Mk reads a file describing relationships among files and executes commands to bring the files up to date. The specification file, called a mkfile, contains three types of statements: assignments, includes, and rules. Assignment and include statements are similar to those in C. Rules specify dependencies between a *target* and its *prerequisites*. When the target and prerequisites are files, their modification times determine if they are out of date. Rules often contain a *recipe*, an *rc* (1) script that produces the target from the prerequisites.

This simple mkfile produces an executable from a C source file:

```
CC=pcc
f1:	f1.c
	$CC -o f1 f1.c
```

The first line assigns the name of the portable ANSI/POSIX compiler to the `mk` variable `CC`; subsequent references of the form `$CC` select this compiler. The only rule specifies a dependence between the target file `f1` and the prerequisite file `f1.c`. If the target does not exist or if the prerequisite has been modified more recently than the target, `mk` passes the recipe to `rc` for execution. Here, `f1.c` is compiled and loaded to produce `f1`.

The native Plan 9 environment requires executables for all architectures, not only the current one. The Plan 9 version of the same `mkfile` looks like:

```
</$objtype/mkfile

f1:	f1.$O
	$LD $LDFLAGS -o f1 f1.$O
f1.$O:	f1.c
	$CC $CFLAGS f1.c
```

The first line is an include statement that replaces itself with the contents of the file `/$objtype/mkfile`. The variable `$objtype` is inherited from the environment and contains the name of the target architecture. The prototype `mkfile` for that architecture defines architecture-specific variables: `CC` and `LD` are the names of the compiler and loader, `O` is the code character of the architecture. The rules compile the source file into an object file and invoke the loader to produce `f1`. Invoking `mk` from the command line as follows

```
% objtype=mips mk
vc -w f1.c
vl $LDFLAGS -o f1 f1.k
%
```

produces the `mips` executable of program `f1` regardless of the current architecture type.

We can extend the `mkfile` to build two programs:

```
</$objtype/mkfile
ALL=f1 f2

all:V:	$ALL

f1:	f1.$O
	$LD $LDFLAGS -o f1 f1.$O
f1.$O:	f1.c
	$CC $CFLAGS f1.c
f2:	f2.$O
	$LD $LDFLAGS -o f2 f2.$O
f2.$O:	f2.c
	$CC $CFLAGS f2.c
```

The target `all`, modified by the *attribute* `V`, builds both programs. The attribute identifies `all` as a dummy target that is not related to a file of the same name; its precise effect is explained later. This example describes cascading dependencies: the first target depends on another which depends on a third and so on. Here, individual rules build each program; later we'll see how to do this with a general rule.

3. Variables and the environment

Mk does not distinguish between its internal variables and rc variables in the environment. When mk starts, it imports each environment variable into a mk variable of the same name. Before executing a recipe, mk exports all variables, including those inherited from the environment, to the environment in which rc executes the recipe.

There are several ways for a variable to take a value. It can be set with an assignment statement, inherited from the environment, or specified on the command line. Mk also maintains several special internal variables that are described in *mk*(1). Assignments have the following decreasing order of precedence:

1) Command line assignment
2) Assignment statement
3) Imported from the environment
4) Implicitly set by mk

For example, a command line assignment overrides a value imported from the environment.

All variable values are strings. They can be used for pattern matching and comparison but not for arithmetic. A *list* is a string containing several values separated by white space. Each member is handled individually during pattern matching, target selection, and prerequisite evaluation.

A *namelist* is a list produced by transforming the members of an existing list. The transform applies a pattern to each member, replacing each matched string with a new string, much as in the substitute command in *sam*(1) or *ed*(1). The syntax is

${*var*:A%B=C%D}

where *var* is a variable. The pattern A%B matches a member beginning with the string *A* and ending with the string *B* with any string in between; it behaves like the regular expression A.*B. When a member of the *var* list matches this pattern, the string *C* replaces *A*, *D* replaces *B*, and the matched string replaces itself. Any of *A*, *B*, *C*, or *D* may be the empty string. In effect, a namelist is generated by applying the *ed*(1) substitute command

s/*A*(.*)*B*/*C*\1*D*/

to each member of a variable list.

Namelists are useful for generating a list based on a predictable transformation. For example,

```
SRC=a.c b.c c.c
OBJ=${SRC:%.c=%.v}
```

assigns the list (a.v b.v c.v) to OBJ. A namelist may be used anywhere a variable is allowed except in a recipe.

Command output is assigned to a variable using the normal rc syntax:

```
var=`{rc command}
```

The command executes in an environment populated with previously assigned variables, including those inherited from mk's execution environment. The command may be arbitrarily complex; for example,

```
TARG=`{ls -d *.[cy] | sed 's/..$//'}
```

assigns a list of the C and yacc source files in the current directory, stripped of their suffix, to the variable TARG.

4. The include statement

The include statement replaces itself with the contents of a file. It is functionally similar to the #include statement but uses a different syntax:

<filename

The contents of the file are evaluated as they are read. An include statement may be used anywhere except in a recipe.

Unlike make, mk has no built-in rules. Instead, the include statement allows generic rules to be imported from a prototype mkfile; most Plan 9 mkfiles use this approach [Flan95].

5. Rules

A rule has four elements: targets, prerequisites, attributes, and a recipe. It has the form:

```
targets:attributes:prerequisites
        recipe
```

The first line, containing the targets, attributes, and prerequisites is the *rule header*; it must begin at the left margin. The recipe contains zero or more lines, each of which begins with white space. One or more targets must be specified but the attributes, prerequisites, and recipe are optional. A rule specifies a dependency between the target(s) and its prerequisite(s), the recipe brings the target(s) up to date with the prerequisite(s) and attributes modify mk's evaluation of the dependency.

Normally the target is a file that depends on one or more prerequisite files. Mk compares the modification times of each target and each prerequisite; a target is considered out of date when it does not exist or when a prerequisite has been modified more recently. When a target is out of date, mk executes the recipe to bring it up to date. When the recipe completes, the modification time of the target is checked and used in later dependency evaluations. If the recipe does not update the target, evaluation continues with the out of date target.

A prerequisite of one rule may be the target of another. When this happens, the rules cascade to define a multi-step procedure. For example, an executable target depends on prerequisite object files, each of which is a target in a rule with a C source file as the prerequisite. Mk follows a chain of dependencies until it encounters a prerequisite that is not a target of another rule or it finds a target that is up to date. It then executes the recipes in reverse order to produce the desired target.

The rule header is evaluated when the rule is read. Variables are replaced by their values, namelists are generated, and commands are replaced by their output at this time.

Most attributes modify mk's evaluation of a rule. An attribute is usually a single letter but some are more complicated. This paper only discusses commonly used attributes; see *mk*(1) for a complete list.

The V attribute identifies a *virtual* target; that is, a target that is not a file. For example,

```
clean:V:
        rm *.$O $O.out
```

removes executables and compiler intermediate files. The target is virtual because it does not refer to a file named clean. Without the attribute, the recipe would not be executed if a file named clean existed. The Q attribute silences the printing of a recipe before execution. It is useful when the output of a recipe is similar to the recipe:

```
default:QV:
        echo 'No default target; use mk all or mk install'
```

The recipe is an rc script. It is optional but when it is missing, the rule is handled specially, as described later. Unlike make, mk executes recipes without interpretation. After stripping the first white space character from each line it passes the entire recipe to rc on standard input. Since mk does not interpret a recipe, escape conventions are exactly those of rc. Scripts for awk and sed commands can be embedded exactly as they would be entered from the command line. Mk invokes rc with the -e flag, which causes rc to stop if any command in the recipe exits with a non-zero status; the E attribute overrides this behavior and allows rc to continue executing in the face of errors. Before a recipe is executed, variables are exported to the environment where they are available to rc. Commands in the recipe may not read from standard input because mk uses it internally.

References to a variable can yield different values depending on the location of the reference in the mkfile. Mk resolves variable references in assignment statements and rule headers when the statement is read. Variable references in recipes are evaluated by rc when the recipe is executed; this happens after the entire mkfile has been read. The value of a variable in a recipe is the last value assigned in the file. For example,

```
STRING=all

all:VQ:
        echo $STRING
STRING=none
```

produces the message none. A variable assignment in a recipe does not affect the value of the variable in the mkfile for two reasons. First, mk does not import values from the environment when a recipe completes; one recipe cannot pass a value through the environment to another recipe. Second, no recipe is executed until mk has completed its evaluation, so even if a variable were changed, it would not affect the dependency evaluation.

6. Metarules

A *metarule* is a rule based on a pattern. The pattern selects a class of target(s) and identifies related prerequisites. Mk metarules may select targets and prerequisites based on any criterion that can be described by a pattern, not just the suffix transformations associated with program construction.

Metarule patterns are either *intrinsic* or regular expressions conforming to the syntax of *regexp* (6). The intrinsic patterns are shorthand for common regular expressions. The intrinsic pattern % matches one or more of anything; it is equivalent to the regular expression '.+'. The other intrinsic pattern, &, matches one or more of any characters except '/' and '.'. It matches a portion of a path and is equivalent to the regular expression '[^./]+'. An intrinsic pattern in a prerequisite references the string matched by the same intrinsic pattern in the target. For example, the rule

```
        %.v:    %.c
```

says that a file ending in .v depends on a file of the same name with a .c suffix: foo.v depends on foo.c, bar.v depends on bar.c, and so on. The string matched by an intrinsic pattern in the target is supplied to the recipe in the variable $stem. Thus the rule

```
%.$O:   %.c
        $CC $CFLAGS $stem.c
```

creates an object file for the target architecture from a similarly named C source file. If several object files are out of date, the rule is applied repeatedly and $stem refers to each file in turn. Since there is only one stem variable, there can only be one % or & pattern in a target; the pattern %-%.c is illegal.

Metarules simplify the mkfile for building programs f1 and f2:

```
</$objtype/mkfile

ALL=f1  f2

all:V:    $ALL

%:        %.$O
          $LD  -o  $target   $prereq
%.$O:     %.c
          $CC  $CFLAGS   $stem.c
clean:V:
          rm  -f  $ALL  *.[$OS]
```

(The variable $OS is a list of code characters for all architectures.) Here, metarules specify compile and load steps for all C source files. The loader rule relies on two internal variables set by mk during evaluation of the rule: $target is the name of the target(s) and $prereq the name of all prerequisite(s). Metarules allow this mkfile to be easily extended; a new program is supported by adding its name to the third line.

A regular expression metarule must have an R attribute. Prerequisites may reference matching substrings in the target using the form *n* where *n* is a digit from 1 to 9 specifying the *n*th parenthesized sub-expression. In a recipe, $stem*n* is the equivalent reference. For example, a compile rule could be specified using regular expressions:

```
(.+)\.$O:R:        \1.c
          $CC  $CFLAGS   $stem1.c
```

Here, \1 and $stem1 refer to the name of the target object file without the suffix. The variable $stem associated with an intrinsic pattern is undefined in a regular expression metarule.

7. Archives

Mk provides a special mechanism for maintaining an archive. An archive member is referenced using the form *lib*(*file*) where *lib* is the name of the archive and *file* is the name of the member. Two rules define the dependency between an object file and its membership in an archive:

```
$LIB(foo.8):N:     foo.8
$LIB:     $LIB(foo.8)
          ar  rv  $LIB  foo.8
```

The first rule establishes a dependency between the archive member and the object file. Normally, mk detects an error when a target does not exist and the rule contains no recipe; the N attribute overrides this behavior because the subsequent rule updates the member. The second rule establishes the dependency between the member and the archive; its recipe inserts the member into the archive. This two-step specification allows the modification time of the archive to represent the state of its members. Other rules can then specify the archive as a prerequisite instead of listing each member.

A metarule generalizes library maintenance:

```
LIB=lib.a
OBJS=etoa.$O atoe.$O ebcdic.$O

$LIB(%):N:      %
$LIB:     ${OBJS:%=$LIB(%)}
          ar rv $LIB $OBJS
```

The namelist prerequisite of the `$LIB` target generates archive member names for each object file name; for example, `etoa.$O` becomes `lib.a(etoa.$O)`. This formulation always updates all members. This is acceptable for a small archive, but may be slow for a big one. The rule

```
$LIB:     ${OBJS:%=$LIB(%)}
          ar rv $LIB `{membername $newprereq}
```

only updates out of date object files. The internal variable `$newprereq` contains the names of the out of date prerequisites. The rc script `membername` transforms an archive member specification into a file name: it translates `lib.a(etoa.$O)` into `etoa.$O`.

The mkfile

```
</$objtype/mkfile
LIB=lib.a
OBJS=etoa.$O atoe.$O ebcdic.$O

prog:     main.$O $LIB
          $LD -o $target $prereq

$LIB(%):N:      %
$LIB:     ${OBJS:%=$LIB(%)}
          ar rv $LIB $OBJS
```

builds a program by loading it with a library.

8. Evaluation algorithm

For each target of interest, mk uses the rules in a mkfile to build a data structure called a dependency graph. The nodes of the graph represent targets and prerequisites; a directed arc from one node to another indicates that the file associated with the first node depends on the file associated with the second. When the mkfile has been completely read, the graph is analyzed. In the first step, implied dependencies are resolved by computing the *transitive closure* of the graph. This calculation extends the graph to include all targets that are potentially derivable from the rules in the mkfile. Next the graph is checked for cycles; make accepts cyclic dependencies, but mk does not allow them. Subsequent steps prune subgraphs that are irrelevant for producing the desired target and verify that there is only one way to build it. The recipes associated with the nodes on the longest path between the target and an out of date prerequisite are then executed in reverse order.

The transitive closure calculation is sensitive to metarules; the patterns often select many potential targets and cause the graph to grow rapidly. Fortunately, dependencies associated with the desired target usually form a small part of the graph, so, after pruning, analysis is tractable. For example, the rules

```
%:        x.%
          recipe1
x.%:      %.k
          recipe2
%.k:      %.f
          recipe3
```

produce a graph with four nodes for each file in the current directory. If the desired target is foo, mk detects the dependency between it and the original file foo.f through intermediate dependencies on foo.k and x.foo. Nodes associated with other files are deleted during pruning because they are irrelevant to the production of foo.

Mk avoids infinite cycles by evaluating each metarule once. Thus, the rule

```
%:	%.z
	cp $prereq $prereq.z
```

copies the prerequisite file once.

9. Conventions for evaluating rules

There must be only one way to build each target. However, during evaluation metarule patterns often select potential targets that conflict with the targets of other rules. Mk uses several conventions to resolve ambiguities and to select the proper dependencies.

When a target selects more than one rule, mk chooses a regular rule over a metarule. For example, the mkfile

```
</$objtype/mkfile

FILES=f1.$O f2.$O f3.$O

prog:	$FILES
	$LD -o $target $prereq

%.$O:	%.c
	$CC $CFLAGS $stem.c

f2.$O:	f2.c
	$CC f2.c
```

contains two rules that could build f2.$O. Mk selects the last rule because its target, f2.$O, is explicitly specified, while the %.$O rule is a metarule. In effect, the explicit rule for f2.$O overrides the general rule for building object files from C source files.

When a rule has a target and prerequisites but no recipe, those prerequisites are added to all other rules with recipes that have the same target. All prerequisites, regardless of where they were specified, are exported to the recipe in variable $prereq. For example, in

```
</$objtype/mkfile

FILES=f1.$O f2.$O f3.$O

prog:	$FILES
	$LD -o $target $prereq

%.$O:	hdr.h

%.$O:	%.c
	$CC $CFLAGS $stem.c
```

the second rule adds hdr.h as a prerequisite of the compile metarule; an object file produced from a C source file depends on hdr.h as well as the source file. Notice that the recipe of the compile rule uses $stem.c instead of $prereq because the latter specification would attempt to compile hdr.h.

When a target is virtual and there is no other rule with the same target, mk evaluates each prerequisite. For example, adding the rule

```
all:V:    prog
```

to the preceding example builds the executable when either prog or all is the specified target. In effect, the all target is an alias for prog.

When two rules have identical rule headers and both have recipes, the later rule replaces the former one. For example, if a file named mkrules contains

```
$O.out:   $OFILES
          $LD  $LFLAGS   $OFILES
%.$O:     %.c
          $CC  $CFLAGS   $stem.c
```

the mkfile

```
OFILES=f1.$O    f2.$O   f3.$O

<mkrules

$O.out:   $OFILES
          $LD  $LFLAGS   -l  $OFILES   -lbio   -lc
```

overrides the general loader rule with a special rule using a non-standard library search sequence. A rule is neutralized by overriding it with a rule with a null recipe:

```
<mkrules

$O.out:Q:           $OFILES
          ;
```

The Q attribute suppresses the printing of the semicolon.

When a rule has no prerequisites, the recipe is executed only when the target does not exist. For example,

```
marker:
          touch  $target
```

defines a rule to manage a marker file. If the file exists, it is considered up to date regardless of its modification time. When a virtual target has no prerequisites the recipe is always executed. The clean rule is of this type:

```
clean:V:
          rm  -f  [$OS].out   *.[$OS]
```

When a rule without prerequisites has multiple targets, the extra targets are aliases for the rule. For example, in

```
clean  tidy  nuke:V:
          rm  -f  [$OS].out   *.[$OS]
```

the rule can be invoked by any of three names. The first rule in a mkfile is handled specially: when mk is invoked without a command line target all targets of the first non-metarule are built. If that rule has multiple targets, the recipe is executed once for each target; normally, the recipe of a rule with multiple targets is only executed once.

A rule applies to a target only when its prerequisites exist or can be derived. More than one rule may have the same target as long as only one rule with a recipe remains applicable after the dependency evaluation completes. For example, consider a program built from C and assembler source files. Two rules produce object files:

```
%.$O:    %.c
         $CC  $CFLAGS  $stem.c
%.$O:    %.s
         $AS  $AFLAGS  $stem.s
```

As long as there are not two source files with names like *foo*.c and *foo*.s, mk can unambiguously select the proper rule. If both files exist, the rules are ambiguous and mk exits with an error message.

In Plan 9, many programs consist of portable code stored in one directory and architecture-specific source stored in another. For example, the mkfile

```
</$objtype/mkfile

FILES=f1.$O  f2.$O  f3.$O  f3.$O

prog:    $FILES
         $LD  -o  $target  $prereq

%.$O:    %.$c
         $CC  $CFLAGS  $stem.c

%.$O:    ../port/%.c
         $CC  $CFLAGS  ../port/$stem.c
```

builds the program named prog using portable code in directory ../port and architecture-specific code in the current directory. As long as the names of the C source files in ../port do not conflict with the names of files in the current directory, mk selects the appropriate rule to build the object file. If like-named files exist in both directories, the specification is ambiguous and an explicit target must be specified to resolve the ambiguity. For example, adding the rule

```
f2.$O:   f2.c
         $CC  $CFLAGS  $f2.c
```

to the previous mkfile uses the architecture-specific version of f2.c instead of the portable one. Here, the explicit rule unambiguously documents which of the like-named source files is used to build the program.

Mk s heuristics can produce unintended results when rules are not carefully specified. For example, the rules that build object files from C or assembler source files

```
%.$O:    %.c
         $CC  $CFLAGS  $stem.c
%.$O:    %.s
         $AS  $AFLAGS  $stem.s
```

illustrate a subtle pitfall. Adding a header file dependency to the compile rule

```
%.$O:    %.c  hdr.h
         $CC  $CFLAGS  $stem.c
```

produces the error message

```
don't  know  how  to  make  'file.c'
```

when *file*.s is an assembler source file. This occurs because *file*.s satisfies the assembler rule and hdr.h satisfies the compile rule, so either rule can potentially produce the target. When a prerequisite exists or can be derived, all other prerequisites in that rule header must exist or be derivable; here, the existence of hdr.h forces the evaluation of a C source file. Specifying the dependencies in different rules avoids this interpretation:

```
%.$O:     hdr.h
%.$O:     %.c
          $CC $CFLAGS $stem.c
```

Although hdr.h is an additional prerequisite of the compile rule, the two rules are evaluated independently and the existence of the C source file is not linked to the existence of the header file. However, this specification describes a different dependency. Originally, only object files derived from C files depended on hdr.h; now all object files, including those built from assembler source, depend on the header file.

Metarule patterns should be as restrictive as possible to prevent conflicts with other rules. Consider the mkfile

```
</$objtype/mkfile
BIN=/$objtype/bin
PROG=foo

install:V:          $BIN/$PROG

%:        %.c
          $CC $stem.c
          $LD -o $target $stem.$O

$BIN/%:   %
          mv $stem $target
```

The first target builds an executable in the local directory; the second installs it in the directory of executables for the architecture. Invoking mk with the install target produces:

```
mk: ambiguous recipes for /mips/bin/foo:
/mips/bin/foo <-(mkfile:8)- /mips/bin/foo.c <-(mkfile:12)- foo.c
/mips/bin/foo <-(mkfile:12)- foo <-(mkfile:8)- foo.c
```

The prerequisite of the install rule, $BIN/$PROG, matches both metarules because the % pattern matches everything. The & pattern restricts the compile rule to files in the current directory and avoids the conflict:

```
&:        &.c
          $CC $stem.c
          $LD -o $target $stem.$O
```

10. Missing intermediates

Mk does not build a missing intermediate file if a target is up to date with the prerequisites of the intermediate. For example, when an executable is up to date with its source file, mk does not compile the source to create a missing object file. The evaluation only applies when a target is considered up to date by pretending that the intermediate exists. Thus, it does not apply when the intermediate is a command line target or when it has no prerequisites.

This capability is useful for maintaining archives. We can modify the archive update recipe to remove object files after they are archived:

```
$LIB(%):N:          %
$LIB:     ${OBJS:%=$LIB(%)}
          names=`{membername $newprereq}
          ar rv $LIB $names
          rm -f $names
```

A subsequent mk does not remake the object files as long as the members of the archive remain up to date with the source files. The -i command line option overrides this behavior and causes all intermediates to be built.

11. Alternative out-of-date determination

Sometimes the modification time is not useful for deciding when a target and prerequisite are out of date. The P attribute replaces the default mechanism with the result of a command. The command immediately follows the attribute and is repeatedly executed with each target and each prerequisite as its arguments; if its exit status is non-zero, they are considered out of date and the recipe is executed. Consider the mkfile

```
foo.ref:Pcmp -s:    foo
	cp $prereq $target
```

The command

```
cmp -s foo.ref foo
```

is executed and if foo.ref differs from foo, the latter file is copied to the former.

12. Parallel processing

When possible, mk executes recipes in parallel. The variable $NPROC specifies the maximum number of simultaneously executing recipes. Normally it is imported from the environment, where the system has set it to the number of available processors. It can be decreased by assigning a new value and can be set to 1 to force single-threaded recipe execution. This is necessary when several targets access a common resource such as a status file or data base. When there is no dependency between targets, mk assumes the recipes can be executed concurrently. Normally, this allows multiple prerequisites to be built simultaneously; for example, the object file prerequisites of a load rule can be produced by compiling the source files in parallel. Mk does not define the order of execution of independent recipes. When the prerequisites of a rule are not independent, the dependencies between them should be specified in a rule or the mkfile should be single-threaded. For example, the archive update rules

```
$LIB(%):N:	%
$LIB:	${OBJS:%=$LIB(%)}
	ar rv $LIB `{membername $newprereq}
```

compile source files in parallel but update all members of the archive at once. It is a mistake to merge the two rules

```
$LIB(%):	%
	ar rv $LIB $stem
```

because an ar command is executed for every member of the library. Not only is this inefficient, but the archive is updated in parallel, making interference likely.

The $nproc environment variable contains a number associated with the processor executing a recipe. It can be used to create unique names when the recipe may be executing simultaneously on several processors. Other maintenance tools provide mechanisms to control recipe scheduling explicitly [Cmel86], but mk's general rules are sufficient for all but the most unusual cases.

13. Deleting target files on errors

The D attribute causes mk to remove the target file when a recipe terminates prematurely. The error message describing the termination condition warns of the deletion. A partially built file is doubly dangerous: it is not only wrong, but is also considered to be up to date so a subsequent mk will not rebuild it. For example,

```
pic.out:D:	mk.ms
	pic $prereq | tbl | troff -ms > $target
```

produces the message

```
mk: pic mk.ms | ... : exit status=rc 685: deleting 'pic.out'
```

if any program in the recipe exits with an error status.

14. Unspecified dependencies

The -w command line flag forces the files following the flag to be treated as if they were just modified. We can use this flag with a command that selects files to force a build based on the selection criterion. For example, if the declaration of a global variable named *var* is changed in a header file, all source files that reference it can be rebuilt with the command

```
$ mk -w`{grep -l var *.[cyl]}
```

15. Conclusion

There are many programs related to make, each choosing a different balance between specialization and generality. Mk emphasizes generality but allows customization through its pattern specifications and include facilities.

Plan 9 presents a difficult maintenance environment with its heterogeneous architectures and languages. Mk's flexible specification language and simple interaction with rc work well in this environment. As a result, Plan 9 relies on mk to automate almost all maintenance. Tasks as diverse as updating the network database, producing the manual, or building a release are expressed as mk procedures.

16. References

[Cmel86] R. F. Cmelik, Concurrent Make: A Distributed Program in Concurrent C, AT&T Bell Laboratories Technical Report, 1986.

[Feld79] S. I. Feldman, Make — a program for maintaining computer programs, *Software Practice & Experience*, 1979 Vol 9 #4, pp. 255-266.

[Flan95] Bob Flandrena, Plan 9 Mkfiles, this volume.

[Hume87] A. G. Hume, Mk: A Successor to Make, *USENIX Summer Conf. Proc.*, Phoenix, Az.

17. Appendix: Differences between make and mk

The differences between mk and make are:

- Make builds targets when it needs them, allowing systematic use of side effects. Mk constructs the entire dependency graph before building any target.
- Make supports suffix rules and % metarules. Mk supports % and regular expression metarules. (Older versions of make support only suffix rules.)
- Mk performs transitive closure on metarules, make does not.
- Make supports cyclic dependencies, mk does not.
- Make evaluates recipes one line at a time, replacing variables by their values and executing some commands internally. Mk passes the entire recipe to the shell without interpretation or internal execution.
- Make supports parallel execution of single-line recipes when building the prerequisites for specified targets. Mk supports parallel execution of all recipes. (Older versions of make did not support parallel execution.)
- Make uses special targets (beginning with a period) to indicate special processing. Mk uses attributes to modify rule evaluation.
- Mk supports virtual targets that are independent of the file system.

Mk allows non-standard out-of-date determination; *make* does not.

It is usually easy to convert a *makefile* to or from an equivalent *mkfile*.

Plan 9 Mkfiles

Bob Flandrena
bobf@plan9.bell-labs.com

Introduction

Every Plan 9 source directory contains a file, called `mkfile`, specifying the rules for building the executable or library that is the product of the directory. *Mk*(1) interprets the rules in the file, calculates the dependencies, and executes an *rc*(1) script to construct the product. If necessary components are supplied by neighboring directories or sub-directories, the mkfiles in those directories are first executed to build the components before the local construction proceeds.

Most application source directories produce one of four types of product: a single executable, several executables, a local library, or a system library. Four generic mkfiles define the normal rules for building each type of product. The simplest mkfiles need only list the components and include the appropriate generic mkfile to do the work. More complex mkfiles may supply additional rules to augment, modify, or override the generic rules.

Using a Mkfile

To build a product, change to the directory containing its source and invoke *mk* with the appropriate target as an argument. All mkfiles provide the following standard targets:

`all` — Build a local version of the product or products for the current architecture. If the product is a single program, the result is stored in file `$O.out`. If the directory produces multiple executables, they are stored in the files named `$O.`*progname*, where *progname* is the name of each executable. A product may be built for a different architecture by prefacing the `mk` command with `objtype=`*architecture*, where *architecture* is the name of the target architecture. Directories producing system libraries always operate directly on the installed version of the library; in this case the target `all` is equivalent to the target `install`.

`install` — Build and install the product or products for the current architecture.

`installall` — Build and install the product or products for all architectures.

`clean` — Rid the directory and its subdirectories of the by-products of the build process. Intermediate files that are easily reproduced (e.g., object files, `yacc` intermediates, target executables) are always removed. Complicated intermediates, such as local libraries, are usually preserved.

`nuke` — Remove all intermediates from the directory and any subdirectories. This target guarantees that a subsequent build for the architecture is performed from scratch.

If no target is specified on the `mk` command line, the `all` target is built by default. In a directory producing multiple executables, there is no default target.

In addition to the five standard targets, additional targets may be supplied by each generic mkfile or by the directory's mkfile.

The environment variable NPROC is set by the system to the number of available processors. Setting this variable, either in the environment or in a mkfile, controls the amount of parallelism in the build. For example, the command

```
NPROC=1 mk
```

restricts a build to a single thread of execution.

Creating a Mkfile

The easiest way to build a new mkfile is to copy and modify an existing mkfile of the same type. Failing that, it is usually possible to create a new mkfile with minimal effort, since the appropriate generic mkfile predefines the rules that do all the work. In the simplest and most common cases, the new mkfile need only define a couple of variables and include the appropriate architecture-specific and generic mkfiles.

There are four generic mkfiles containing commonly used rules for building a product: mkone, mkmany, mklib, and mksyslib. These rules perform such actions as compiling C source files, loading object files, archiving libraries, and installing executables in the bin directory of the appropriate architecture. The generic mkfiles are stored in directory /sys/src/cmd. Mkfile mkone builds a single executable, mkmany builds several executables from the source in a single directory, and mklib and mksyslib, maintain local and system libraries, respectively. The rules in the generic mkfiles are driven by the values of variables, some of which must be set by the product mkfile and some of which are supplied by the generic mkfile. Variables in the latter class include:

Variable	*Default*	*Meaning*
CFLAGS	-FVw	C compiler flags
LDFLAGS		Loader flags
YFLAGS	-d	Yacc flags
AFLAGS		Assembler flags

The following variables are set by the product mkfile and used by the generic mkfile. Any may be empty depending on the specific product being made.

TARG	Name(s) of the executable(s) to be built
LIB	Library name(s)
OFILES	Object files
HFILES	Header files included by all source files
YFILES	Yacc input files
BIN	Directory where executables are installed

Mkfile Organization

All mkfiles share the following common structure:

</$objtype/mkfile	# architecture-dependent definitions
variable definitions	# TARG, OFILES, HFILES, etc.
</sys/src/cmd/*generic*	# mkone, mkmany, mklib, or mksyslib
variable overrides	# CFLAGS, objtype, etc.
extra rules	# overrides, augmented rules, additional targets

Note that the architecture-dependent mkfiles include /sys/src/mkfile.proto for system-wide variables that are common to all architectures.

The variables driving the expansion of the generic mkfile may be specified in any order as long as they are defined before the inclusion of the generic mkfile. The value of a variable may be changed by assigning a new value following the inclusion of the generic mkfile, but the effects are sometimes counter-intuitive. Such variable assignments do not apply to the target and prerequisite portions of any previously defined rules; the new values only apply to the recipes of rules preceding the assignment statement and to all parts of any rules following it.

The rules supplied by the generic mkfile may be overridden or augmented. The new rules must be specified after the inclusion of the generic mkfile. If the target and prerequisite portion of the rule exactly match the target and prerequisite portion of a previously defined rule and the new rule contains a recipe, the new rule replaces the old one. If the target of a new rule exactly matches the target of a previous rule and one or more new prerequisites are specified and the new rule contains no recipe, the new prerequisites are added to the prerequisites of the old rule.

Following sections discuss each generic mkfile in detail.

Mkone

The mkone generic mkfile contains rules for building a single executable from one or more files in a directory. The variable TARG specifies the name of the executable and variables OFILES and YFILES specify the object files and yacc source files used to build it. HFILES contains the names of the local header files included in all source files. BIN is the name of the directory where the executable is installed. LIB contains the names of local libraries used by the linker. This variable is rarely needed as libraries referenced by a #pragma directive in an associated header file, including all system libraries, are automatically searched by the loader.

If mk is executed without a target, the all target is built; it produces an executable in $O.out. Variable HFILES identifies the header files that are included in all or most or the C source files. Occasionally, a program has other header files that are only used in some source files. A header can be added to the prerequisites for those object files by adding a rule of the following form following the inclusion of mkone generic mkfile:

```
file.$O:        header.h
```

The mkfile for a directory producing a single executable using the normal set of rules is trivial: a list of some files followed by the inclusion of *mkone.* For example, /sys/src/cmd/diff/mkfile contains:

```
< /$objtype/mkfile

TARG=diff
OFILES=\
        diffdir.$O\
        diffio.$O\
        diffreg.$O\
        main.$O\

HFILES=diff.h

BIN=/$objtype/bin
</sys/src/cmd/mkone
```

The more complex mkfile in /sys/src/cmd/awk overrides compiler and loader variables to select the ANSI/POSIX Computing Environment with appropriately defined command line variables. It also overrides the default yacc rule to place the output soure in file awkgram.c and the clean and nuke rules, so it can remove the non-standard intermediate files. Finally, the last three rules build a version of maketab appropriate for the architecture where the mk is being run and then executes it to create source file proctab.c:

```
</$objtype/mkfile

TARG=awk
OFILES=re.$O\
        lex.$O\
        main.$O\
        parse.$O\
        proctab.$O\
        tran.$O\
        lib.$O\
        run.$O\
        awkgram.$O\

HFILES=awk.h\
        y.tab.h\
        proto.h\

YFILES=awkgram.y

BIN=/$objtype/bin
</sys/src/cmd/mkone
CFLAGS=-c -D_REGEXP_EXTENSION -D_RESEARCH_SOURCE \
        -D_BSD_EXTENSION -DUTF
YFLAGS=-S -d -v
CC=pcc
LD=pcc
cpuobjtype=`{sed -n 's/^O=//p' /$cputype/mkfile}

y.tab.h awkgram.c:      $YFILES
        $YACC -o awkgram.c $YFLAGS $prereq

clean:V:
        rm -f *.[$OS] [$OS].out [$OS].maketab y.tab.? y.debug\
                y.output $TARG

nuke:V:
        rm -f *.[$OS] [$OS].out [$OS].maketab y.tab.? y.debug\
                y.output awkgram.c $TARG

proctab.c:      $cpuobjtype.maketab
        ./$cpuobjtype.maketab >proctab.c

$cpuobjtype.maketab:    y.tab.h maketab.c
        objtype=$cputype
        mk maketab.$cputype

maketab.$cputype:V:     y.tab.h maketab.$O
        $LD -o $O.maketab maketab.$O
```

Mkmany

The mkmany generic mkfile builds several executables from the files in a directory. It differs from the operation of mkone in three respects: TARG specifies the names of all executables, there is no default command-line target, and additional rules allow a single executable to be built or installed.

The TARG variable specifies the names of all executables produced by the mkfile. The rules assume the name of each executable is also the name of the file containing its main function. OFILES specifies files containing common subroutines loaded with all executables. Consider the mkfile:

```
</$objtype/mkfile

TARG=alpha   beta
OFILES=common.$O
BIN=/$objtype/bin
</sys/src/cmd/mkmany
```

It assumes the main functions for executables alpha and beta are in files alpha.$O and beta.$O and that both programs use the subroutines in file common.$O. The all target builds all executables, leaving each in a file with a name of the form $O.*progname* where *progname* is the name of the executable. In this example the all target produces executables $O.alpha and $O.beta.

The mkmany rules provide additional targets for building a single executable:

$O.progname	Builds executable $O.*progname* in the current directory. When the target architecture is not the current architecture the mk command must be prefixed with the customary objtype=*architecture* assignment to select the proper compilers and loaders.
progname.install	Installs executable *progname* for the target architecture.
progname.installall	Installs executable *progname* for all architectures.

Mklib

The mklib generic mkfile builds a local library. Since this form of mkfile constructs no executable, the TARG and BIN variables are not needed. Instead, the LIB variable specifies the library to be built or updated. Variable OFILES contains the names of the object files to be archived in the library. The use of variables YFILES and HFILES does not change. When possible, only the out-of-date members of the library are updated.

The variable LIBDIR contains the name of the directory where the library is installed; by default it selects the current directory. It can be overridden by assigning the new directory name after the point where mklib is included.

The clean target removes object files and yacc intermediate files but does not touch the library. The nuke target removes the library as well as the files removed by the clean target. The command

```
mk -s clean all
```

causes the existing library to be updated, or created if it doesn't already exist. The command

```
mk -s nuke all
```

forces the library to be rebuilt from scratch.

The mkfile from /sys/src/cmd/upas/libString contains the following specifications to build the local library libString.a$O for the object architecture referenced by $O:

```
</$objtype/mkfile

LIB=libString.a$O
OFILES=   s_alloc.$O\
          s_append.$O\
          s_array.$O\
          s_copy.$O\
          s_getline.$O\
          s_grow.$O\
          s_nappend.$O\
          s_parse.$O\
          s_read.$O\
          s_read_line.$O\
          s_tolower.$O\

</sys/src/cmd/mklib

nuke:V:
          mk clean
          rm -f libString.a[$OS]
```

The override of the rule for target `nuke` removes the libraries for all architectures as opposed to the default recipe for this target which removes the library for the current architecture.

Mksyslib

The `mksyslib` generic mkfile is similar to the `mklib` mkfile except that it operates on a system library instead of a local library. The `install` and `all` targets are the same; since there is no local copy of the library, all updates are performed on the installed library. The rule for the `nuke` target is identical to that of the `clean` target; unlike the `nuke` target for local libraries, the library is never removed.

No attempt is made to determine if individual library members are up-to-date; all members of a library are always updated. Special targets support manipulation of a single object file; the target `objfile` updates file `objfile.$O` in the library of the current architecture and the target `objfile.all` updates `objfile.$O` in the libraries of all architectures.

Overrides

The rules provided by a generic mkfile or the variables used to control the evaluation of those rules may be overridden in most circumstances. Overrides must be specified in the product mkfile after the point where the generic mkfile is included; in general, variable and rule overrides occupy the end of a product mkfile.

The value of a variable is overridden by assigning a new value to the variable. Most variable overrides modify the values of flags or the names of commands executed in recipes. For example, the default value of `CFLAGS` is often overridden or augmented and the ANSI/POSIX Computing Environment is selected by setting the `CC` and `LD` variables to `pcc`.

Modifying rules is trickier than modifying variables. Additional constraints can be added to a rule by specifying the target and the new prerequisite. For example,

```
%.$O:     header.h
```

adds file `header.h` to the set of prerequisites for all object files. There is no mechanism for adding additional commands to an existing recipe; if a recipe is unsatisfactory, the rule and its recipe must be completely overridden. A rule is overridden only when the replacement rule matches the target and prerequisite portions of the original rule exactly. The recipe associated with the new rule then replaces the recipe of the original rule. For example, /sys/src/cmd/lex/mkfile overrides the default installall rule to perform the normal loop on all architectures and then copy a prototype file to the system library directory.

```
</$objtype/mkfile

TARG=lex
OFILES=lmain.$O\
        y.tab.$O\
        sub1.$O\
        sub2.$O\
        header.$O\

HFILES=ldefs.h\

YFILES=parser.y\

BIN=/$objtype/bin
</sys/src/cmd/mkone

installall:V:
        for(objtype in $CPUS)
                mk install
        cp ncform /sys/lib/lex
```

Another way to perform the same override is to add a dependency to the default installall rule that executes an additional rule to install the prototype file:

```
installall:V:   ncform.install

ncform.install:V:
        cp ncform /sys/lib/lex
```

Special Tricks

Two special cases require extra deviousness.

In the first, a file needed to build an executable is generated by a program that, in turn, is built from a source file that is not part of the product. In this case, the executable must be built for the target architecture, but the intermediate executable must be built for the architecture mk is executing on. The intermediate executable is built by recursively invoking mk with the appropriate target and the executing architecture as the target architecture. When that mk completes, the intermediate is executed to generate the source file to complete the build for the target architecture. The earlier example of /sys/src/cmd/awk/mkfile illustrates this technique.

Another awkward situation occurs when a directory contains source to build an executable as well as source for auxiliary executables that are not to be installed. In this case the mkmany generic rules are inappropriate, because all executables would be built and installed. Instead, use the mkone generic file to build the primary executable and provide extra targets to build the auxiliary files. This approach is also useful when the auxiliary files are not executables; /sys/src/cmd/spell/mkfile augments the default rules to build and install the spell executable with elaborate rules to generate and maintain the auxiliary spelling lists.

A Manual for the Plan 9 assembler

Rob Pike
rob@plan9.bell-labs.com

Machines

There is an assembler for each of the MIPS, SPARC, Intel 386, AMD64, Power PC, and ARM. The 68020 assembler, 2a, (no longer distributed) is the oldest and in many ways the prototype. The assemblers are really just variations of a single program: they share many properties such as left-to-right assignment order for instruction operands and the synthesis of macro instructions such as MOVE to hide the peculiarities of the load and store structure of the machines. To keep things concrete, the first part of this manual is specifically about the 68020. At the end is a description of the differences among the other assemblers.

The document, How to Use the Plan 9 C Compiler, by Rob Pike, is a prerequisite for this manual.

Registers

All pre-defined symbols in the assembler are upper-case. Data registers are R0 through R7; address registers are A0 through A7; floating-point registers are F0 through F7.

A pointer in A6 is used by the C compiler to point to data, enabling short addresses to be used more often. The value of A6 is constant and must be set during C program initialization to the address of the externally-defined symbol a6base.

The following hardware registers are defined in the assembler; their meaning should be obvious given a 68020 manual: CAAR, CACR, CCR, DFC, ISP, MSP, SFC, SR, USP, and VBR.

The assembler also defines several pseudo-registers that manipulate the stack: FP, SP, and TOS. FP is the frame pointer, so 0(FP) is the first argument, 4(FP) is the second, and so on. SP is the local stack pointer, where automatic variables are held (SP is a pseudo-register only on the 68020); 0(SP) is the first automatic, and so on as with FP. Finally, TOS is the top-of-stack register, used for pushing parameters to procedures, saving temporary values, and so on.

The assembler and loader track these pseudo-registers so the above statements are true regardless of what has been pushed on the hardware stack, pointed to by A7. The name A7 refers to the hardware stack pointer, but beware of mixed use of A7 and the above stack-related pseudo-registers, which will cause trouble. Note, too, that the PEA instruction is observed by the loader to alter SP and thus will insert a corresponding pop before all returns. The assembler accepts a label-like name to be attached to FP and SP uses, such as p+0(FP), to help document that p is the first argument to a routine. The name goes in the symbol table but has no significance to the result of the program.

Referring to data

All external references must be made relative to some pseudo-register, either PC (the virtual program counter) or SB (the static base register). PC counts instructions, not bytes of data. For example, to branch to the second following instruction, that is, to skip one instruction, one may write

```
        BRA     2(PC)
```

Labels are also allowed, as in

```
        BRA     return
        NOP
return:
        RTS
```

When using labels, there is no (PC) annotation.

The pseudo-register SB refers to the beginning of the address space of the program. Thus, references to global data and procedures are written as offsets to SB, as in

```
        MOVL    $array(SB),   TOS
```

to push the address of a global array on the stack, or

```
        MOVL    array+4(SB),   TOS
```

to push the second (4-byte) element of the array. Note the use of an offset; the complete list of addressing modes is given below. Similarly, subroutine calls must use SB:

```
        BSR     exit(SB)
```

File-static variables have syntax

```
        local<>+4(SB)
```

The <> will be filled in at load time by a unique integer.

When a program starts, it must execute

```
        MOVL    $a6base(SB),   A6
```

before accessing any global data. (On machines such as the MIPS and SPARC that cannot load a register in a single instruction, constants are loaded through the static base register. The loader recognizes code that initializes the static base register and treats it specially. You must be careful, however, not to load large constants on such machines when the static base register is not set up, such as early in interrupt routines.)

Expressions

Expressions are mostly what one might expect. Where an offset or a constant is expected, a primary expression with unary operators is allowed. A general C constant expression is allowed in parentheses.

Source files are preprocessed exactly as in the C compiler, so #define and #include work.

Addressing modes

The simple addressing modes are shared by all the assemblers. Here, for completeness, follows a table of all the 68020 addressing modes, since that machine has the richest set. In the table, o is an offset, which if zero may be elided, and d is a displacement, which is a constant between -128 and 127 inclusive. Many of the modes listed have the same name; scrutiny of the format will show what default is being applied. For instance, indexed mode with no address register supplied operates as though a zero-valued register were used. For "offset" read "displacement." For ".s" read one of .L, or .W followed by *1, *2, *4, or *8 to indicate the size and scaling of the data.

```
data register                  R0
address register               A0
floating-point register        F0
special names                  CAAR, CACR, etc.
constant                       $con
floating point constant        $fcon
external symbol                name+o(SB)
local symbol                   name<>+o(SB)
automatic symbol               name+o(SP)
argument                       name+o(FP)
address of external            $name+o(SB)
address of local               $name<>+o(SB)
indirect post-increment        (A0)+
indirect pre-decrement         -(A0)
indirect with offset           o(A0)
indexed with offset            o()(R0.s)
indexed with offset            o(A0)(R0.s)
external indexed               name+o(SB)(R0.s)
local indexed                  name<>+o(SB)(R0.s)
automatic indexed              name+o(SP)(R0.s)
parameter indexed              name+o(FP)(R0.s)
offset indirect post-indexed   d(o())(R0.s)
offset indirect post-indexed   d(o(A0))(R0.s)
external indirect post-indexed d(name+o(SB))(R0.s)
local indirect post-indexed    d(name<>+o(SB))(R0.s)
automatic indirect post-indexed d(name+o(SP))(R0.s)
parameter indirect post-indexed d(name+o(FP))(R0.s)
offset indirect pre-indexed    d(o()(R0.s))
offset indirect pre-indexed    d(o(A0))
offset indirect pre-indexed    d(o(A0)(R0.s))
external indirect pre-indexed  d(name+o(SB))
external indirect pre-indexed  d(name+o(SB)(R0.s))
local indirect pre-indexed     d(name<>+o(SB))
local indirect pre-indexed     d(name<>+o(SB)(R0.s))
automatic indirect pre-indexed d(name+o(SP))
automatic indirect pre-indexed d(name+o(SP)(R0.s))
parameter indirect pre-indexed d(name+o(FP))
parameter indirect pre-indexed d(name+o(FP)(R0.s))
```

Laying down data

Placing data in the instruction stream, say for interrupt vectors, is easy: the pseudo-instructions LONG and WORD (but not BYTE) lay down the value of their single argument, of the appropriate size, as if it were an instruction:

```
        LONG    $12345
```

places the long 12345 (base 10) in the instruction stream. (On most machines, the only such operator is WORD and it lays down 32-bit quantities. The 386 has all three: LONG, WORD, and BYTE. The AMD64 adds QUAD to that for 64-bit values. The 960 has only one, LONG.)

Placing information in the data section is more painful. The pseudo-instruction DATA does the work, given two arguments: an address at which to place the item, including its size, and the value to place there. For example, to define a character array

array containing the characters abc and a terminating null:

```
DATA    array+0(SB)/1,   $'a'
DATA    array+1(SB)/1,   $'b'
DATA    array+2(SB)/1,   $'c'
GLOBL   array(SB),  $4
```

or

```
DATA    array+0(SB)/4,   $"abc\z"
GLOBL   array(SB),  $4
```

The /1 defines the number of bytes to define, GLOBL makes the symbol global, and the $4 says how many bytes the symbol occupies. Uninitialized data is zeroed automatically. The character \z is equivalent to the C \0. The string in a DATA statement may contain a maximum of eight bytes; build larger strings piecewise. Two pseudo-instructions, DYNT and INIT, allow the (obsolete) Alef compilers to build dynamic type information during the load phase. The DYNT pseudo-instruction has two forms:

```
DYNT    , ALEF_SI_5+0(SB)
DYNT    ALEF_AS+0(SB),   ALEF_SI_5+0(SB)
```

In the first form, DYNT defines the symbol to be a small unique integer constant, chosen by the loader, which is some multiple of the word size. In the second form, DYNT defines the second symbol in the same way, places the address of the most recently defined text symbol in the array specified by the first symbol at the index defined by the value of the second symbol, and then adjusts the size of the array accordingly.

The INIT pseudo-instruction takes the same parameters as a DATA statement. Its symbol is used as the base of an array and the data item is installed in the array at the offset specified by the most recent DYNT pseudo-instruction. The size of the array is adjusted accordingly. The DYNT and INIT pseudo-instructions are not implemented on the 68020.

Defining a procedure

Entry points are defined by the pseudo-operation TEXT, which takes as arguments the name of the procedure (including the ubiquitous (SB)) and the number of bytes of automatic storage to pre-allocate on the stack, which will usually be zero when writing assembly language programs. On machines with a link register, such as the MIPS and SPARC, the special value -4 instructs the loader to generate no PC save and restore instructions, even if the function is not a leaf. Here is a complete procedure that returns the sum of its two arguments:

```
TEXT    sum(SB),  $0
        MOVL    arg1+0(FP),   R0
        ADDL    arg2+4(FP),   R0
        RTS
```

An optional middle argument to the TEXT pseudo-op is a bit field of options to the loader. Setting the 1 bit suspends profiling the function when profiling is enabled for the rest of the program. For example,

```
TEXT    sum(SB),  1,  $0
        MOVL    arg1+0(FP),   R0
        ADDL    arg2+4(FP),   R0
        RTS
```

will not be profiled; the first version above would be. Subroutines with peculiar state, such as system call routines, should not be profiled.

Setting the 2 bit allows multiple definitions of the same TEXT symbol in a program; the loader will place only one such function in the image. It was emitted only by the Alef compilers.

Subroutines to be called from C should place their result in R0, even if it is an address. Floating point values are returned in F0. Functions that return a structure to a C program receive as their first argument the address of the location to store the result; R0 is unused in the calling protocol for such procedures. A subroutine is responsible for saving its own registers, and therefore is free to use any registers without saving them (caller saves). A6 and A7 are the exceptions as described above.

When in doubt

If you get confused, try using the -S option to 2c and compiling a sample program. The standard output is valid input to the assembler.

Instructions

The instruction set of the assembler is not identical to that of the machine. It is chosen to match what the compiler generates, augmented slightly by specific needs of the operating system. For example, 2a does not distinguish between the various forms of MOVE instruction: move quick, move address, etc. Instead the context does the job. For example,

```
MOVL    $1, R1
MOVL    A0, R2
MOVW    SR, R3
```

generates official MOVEQ, MOVEA, and MOVESR instructions. A number of instructions do not have the syntax necessary to specify their entire capabilities. Notable examples are the bitfield instructions, the multiply and divide instructions, etc. For a complete set of generated instruction names (in 2a notation, not Motorola's) see the file /sys/src/cmd/2c/2.out.h. Despite its name, this file contains an enumeration of the instructions that appear in the intermediate files generated by the compiler, which correspond exactly to lines of assembly language.

Laying down instructions

The loader modifies the code produced by the assembler and compiler. It folds branches, copies short sequences of code to eliminate branches, and discards unreachable code. The first instruction of every function is assumed to be reachable. The pseudo-instruction NOP, which you may see in compiler output, means no instruction at all, rather than an instruction that does nothing. The loader discards all NOPs.

To generate a true NOP instruction, or any other instruction not known to the assembler, use a WORD pseudo-instruction. Such instructions on RISCs are not scheduled by the loader and must have their delay slots filled manually.

MIPS

The registers are only addressed by number: R0 through R31. R29 is the stack pointer; R30 is used as the static base pointer, the analogue of A6 on the 68020. Its value is the address of the global symbol setR30(SB). The register holding returned values from subroutines is R1. When a function is called, space for the first argument is reserved at 0(FP) but in C (not Alef) the value is passed in R1 instead.

The loader uses R28 as a temporary. The system uses R26 and R27 as interrupt-time temporaries. Therefore none of these registers should be used in user code.

The control registers are not known to the assembler. Instead they are numbered registers M0, M1, etc. Use this trick to access, say, STATUS:

```
#define  STATUS   12
         MOVW     M(STATUS),   R1
```

Floating point registers are called F0 through F31. By convention, F24 must be initialized to the value 0.0, F26 to 0.5, F28 to 1.0, and F30 to 2.0; this is done by the operating system.

The instructions and their syntax are different from those of the manufacturer's manual. There are no lui and kin; instead there are MOVW (move word), MOVH (move halfword), and MOVB (move byte) pseudo-instructions. If the operand is unsigned, the instructions are MOVHU and MOVBU. The order of operands is from left to right in dataflow order, just as on the 68020 but not as in MIPS documentation. This means that the Bcond instructions are reversed with respect to the book; for example, a va BGTZ generates a MIPS bltz instruction.

The assembler is for the R2000, R3000, and most of the R4000 and R6000 architectures. It understands the 64-bit instructions MOVV, MOVVL, ADDV, ADDVU, SUBV, SUBVU, MULV, MULVU, DIVV, DIVVU, SLLV, SRLV, and SRAV. The assembler does not have any cache, load-linked, or store-conditional instructions.

Some assembler instructions are expanded into multiple instructions by the loader. For example the loader may convert the load of a 32 bit constant into an lui followed by an ori.

Assembler instructions should be laid out as if there were no load, branch, or floating point compare delay slots; the loader will rearrange *schedule* the instructions to guarantee correctness and improve performance. The only exception is that the correct scheduling of instructions that use control registers varies from model to model of machine (and is often undocumented) so you should schedule such instructions by hand to guarantee correct behavior. The loader generates

```
         NOR      R0,  R0,  R0
```

when it needs a true no-op instruction. Use exactly this instruction when scheduling code manually; the loader recognizes it and schedules the code before it and after it independently. Also, WORD pseudo-ops are scheduled like no-ops.

The NOSCHED pseudo-op disables instruction scheduling (scheduling is enabled by default); SCHED re-enables it. Branch folding, code copying, and dead code elimination are disabled for instructions that are not scheduled.

SPARC

Once you understand the Plan 9 model for the MIPS, the SPARC is familiar. Registers have numerical names only: R0 through R31. Forget about register windows: Plan 9 doesn't use them at all. The machine has 32 global registers, period. R1 [sic] is the stack pointer. R2 is the static base register, with value the address of setSB(SB). R7 is the return register and also the register holding the first argument to a C (not Alef) function, again with space reserved at 0(FP). R14 is the loader temporary.

Floating-point registers are exactly as on the MIPS.

The control registers are known by names such as FSR. The instructions to access these registers are MOVW instructions, for example

```
         MOVW     Y,  R8
```

for the SPARC instruction

```
         rdy      %r8
```

Move instructions are similar to those on the MIPS: pseudo-operations that turn into appropriate sequences of sethi instructions, adds, etc. Instructions read from left to right. Because the arguments are flipped to SUBCC, the condition codes are not inverted as on the MIPS.

The syntax for the ASI stuff is, for example to move a word from ASI 2:

```
MOVW	(R7, 2), R8
```

The syntax for double indexing is

```
MOVW	(R7+R8), R9
```

The SPARC's instruction scheduling is similar to the MIPS's. The official no-op instruction is:

```
ORN	R0, R0, R0
```

i960

Registers are numbered R0 through R31. Stack pointer is R29; return register is R4; static base is R28; it is initialized to the address of setSB(SB). R3 must be zero; this should be done manually early in execution by

```
SUBO	R3, R3
```

R27 is the loader temporary.

There is no support for floating point.

The Intel calling convention is not supported and cannot be used; use BAL instead. Instructions are mostly as in the book. The major change is that LOAD and STORE are both called MOV. The extension character for MOV is as in the manual: O for ordinal, W for signed, etc.

i386

The assembler assumes 32-bit protected mode. The register names are SP, AX, BX, CX, DX, BP, DI, and SI. The stack pointer (not a pseudo-register) is SP and the return register is AX. There is no physical frame pointer but, as for the MIPS, FP is a pseudo-register that acts as a frame pointer.

Opcode names are mostly the same as those listed in the Intel manual with an L, W, or B appended to identify 32-bit, 16-bit, and 8-bit operations. The exceptions are loads, stores, and conditionals. All load and store opcodes to and from general registers, special registers (such as CR0, CR3, GDTR, IDTR, SS, CS, DS, ES, FS, and GS) or memory are written as

```
MOVx	src,dst
```

where *x* is L, W, or B. Thus to get AL use a MOVB instruction. If you need to access AH, you must mention it explicitly in a MOVB:

```
MOVB	AH, BX
```

There are many examples of illegal moves, for example,

```
MOVB	BP, DI
```

that the loader actually implements as pseudo-operations.

The names of conditions in all conditional instructions (J, SET) follow the conventions of the 68020 instead of those of the Intel assembler: JOS, JOC, JCS, JCC, JEQ, JNE, JLS, JHI, JMI, JPL, JPS, JPC, JLT, JGE, JLE, and JGT instead of JO, JNO, JB, JNB, JZ, JNZ, JBE, JNBE, JS, JNS, JP, JNP, JL, JNL, JLE, and JNLE.

The addressing modes have syntax like AX, (AX), (AX)(BX*4), 10(AX), and 10(AX)(BX*4). The offsets from AX can be replaced by offsets from FP or SB to access names, for example extern+5(SB)(AX*2).

Other notes: Non-relative JMP and CALL have a * added to the syntax. Only LOOP, LOOPEQ, and LOOPNE are legal loop instructions. Only REP and REPN are recognized repeaters. These are not prefixes, but rather stand-alone opcodes that precede the strings, for example

```
CLD; REP; MOVSL
```

Segment override prefixes in MOD/RM fields are not supported.

AMD64

The assembler assumes 64-bit mode unless a MODE pseudo-operation is given:

```
MODE $32
```

to change to 32-bit mode. The effect is mainly to diagnose instructions that are illegal in the given mode, but the loader will also assume 32-bit operands and addresses, and 32-bit PC values for call and return. The assembler's conventions are similar to those for the 386, above. The architecture provides extra fixed-point registers R8 to R15. All registers are 64 bit, but instructions access low-order 8, 16 and 32 bits as described in the processor handbook. For example, MOVL to AX puts a value in the low-order 32 bits and clears the top 32 bits to zero. Literal operands are limited to signed 32 bit values, which are sign-extended to 64 bits in 64 bit operations; the exception is MOVQ, which allows 64-bit literals. The external registers in Plan 9's C are allocated from R15 down.

There are many new instructions, including the MMX and XMM media instructions, and conditional move instructions. MMX registers are M0 to M7, and XMM registers are X0 to X15. As with the 386 instruction names, all new 64-bit integer instructions, and the MMX and XMM instructions uniformly use L for long word (32 bits) and Q for quad word (64 bits). Some instructions use O (octword) for 128-bit values, where the processor handbook variously uses O or DQ. The assembler also consistently uses PL for 'packed long' in XMM instructions, instead of Q, DQ or PI. Either MOVL or MOVQ can be used to move values to and from control registers, even when the registers might be 64 bits. The assembler often accepts the handbook's name to ease conversion of existing code (but remember that the operand order is uniformly source then destination).

C's long long type is 64 bits, but passed and returned by value, not by reference. More notably, C pointer values are 64 bits, and thus long long and unsigned long long are the only integer types wide enough to hold a pointer value. The C compiler and library use the XMM floating-point instructions, not the old 387 ones, although the latter are implemented by assembler and loader. Unlike the 386, the first integer or pointer argument is passed in a register, which is BP for an integer or pointer (it can be referred to in assembly code by the pseudonym RARG). AX holds the return value from subroutines as before. Floating-point results are returned in X0, although currently the first floating-point parameter is not passed in a register. All parameters less than 8 bytes in length have 8 byte slots reserved on the stack to preserve alignment and simplify variable-length argument list access, including the first parameter when passed in a register, even though bytes 4 to 7 are not initialized.

Power PC

The Power PC follows the Plan 9 model set by the MIPS and SPARC, not the elaborate ABIs. The 32-bit instructions of the 60x and 8xx PowerPC architectures are supported; there is no support for the older POWER instructions. Registers are R0 through R31. R0 is initialized to zero; this is done by C start up code and assumed by the compiler and loader. R1 is the stack pointer. R2 is the static base register, with value the address of setSB(SB). R3 is the return register and also the register holding the first argument to a C function, with space reserved at 0(FP) as on the MIPS. R31 is the loader temporary. The external registers in Plan 9's C are allocated from R30 down.

Floating point registers are called F0 through F31. By convention, several registers are initialized to specific values; this is done by the operating system. F27 must be initialized to the value 0x4330000080000000 (used by float-to-int conversion), F28 to the value 0.0, F29 to 0.5, F30 to 1.0, and F31 to 2.0.

As on the MIPS and SPARC, the assembler accepts arbitrary literals as operands to MOVW, and also to ADD and others where immediate variants exist, and the loader generates sequences of addi, addis, oris, etc. as required. The register indirect addressing modes use the same syntax as the SPARC, including double indexing when allowed.

The instruction names are generally derived from the Motorola ones, subject to slight transformation: the . marking the setting of condition codes is replaced by CC, and when the letter o represents OE=1 it is replaced by V. Thus add, addo. and subfzeo. become ADD, ADDVCC and SUBFZEVCC. As well as the three-operand conditional branch instruction BC, the assembler provides pseudo-instructions for the common cases: BEQ, BNE, BGT, BGE, BLT, BLE, BVC, and BVS. The unconditional branch instruction is BR. Indirect branches use (CTR) or (LR) as target.

Load or store operations are replaced by MOV variants in the usual way: MOVW (move word), MOVH (move halfword with sign extension), and MOVB (move byte with sign extension, a pseudo-instruction), with unsigned variants MOVHZ and MOVBZ, and byte-reversing MOVWBR and MOVHBR. Load or store with update versions are MOVWU, MOVHU, and MOVBZU. Load or store multiple is MOVMW. The exceptions are the string instructions, which are LSW and STSW, and the reservation instructions lwarx and stwcx., which are LWAR and STWCCC, all with operands in the usual data-flow order. Floating-point load or store instructions are FMOVD, FMOVDU, FMOVS, and FMOVSU. The register to register move instructions fmr and fmr. are written FMOVD and FMOVDCC.

The assembler knows the commonly used special purpose registers: CR, CTR, DEC, LR, MSR, and XER. The rest, which are often architecture-dependent, are referenced as SPR(n). The segment registers of the 60x series are similarly SEG(n), but *n* can also be a register name, as in SEG(R3). Moves between special purpose registers and general purpose ones, when allowed by the architecture, are written as MOVW, replacing mfcr, mtcr, mfmsr, mtmsr, mtspr, mfspr, mftb, and many others.

The fields of the condition register CR are referenced as CR(0) through CR(7). They are used by the MOVFL (move field) pseudo-instruction, which produces mcrf or mtcrf. For example:

```
MOVFL   CR(3), CR(0)
MOVFL   R3, CR(1)
MOVFL   R3, $7, CR
```

They are also accepted in the conditional branch instruction, for example

```
BEQ     CR(7), label
```

Fields of the FPSCR are accessed using MOVFL in a similar way:

```
MOVFL   FPSCR, F0
MOVFL   F0, FPSCR
MOVFL   F0, $7, FPSCR
MOVFL   $0, FPSCR(3)
```

producing mffs, mtfsf or mtfsfi, as appropriate.

ARM

The assembler provides access to R0 through R14 and the PC. The stack pointer is R13, the link register is R14, and the static base register is R12. R0 is the return register and also the register holding the first argument to a subroutine. The external registers in Plan 9's C are allocated from R10 down. R11 is used by the loader as a temporary register. The assembler supports the CPSR and SPSR registers. It also knows about coprocessor registers C0 through C15. Floating registers are F0 through F7, FPSR and FPCR.

As with the other architectures, loads and stores are called MOV, e.g. MOVW for load word or store word, and MOVM for load or store multiple, depending on the operands.

Addressing modes are supported by suffixes to the instructions: .IA (increment after), .IB (increment before), .DA (decrement after), and .DB (decrement before). These can only be used with the MOV instructions. The move multiple instruction, MOVM, defines a range of registers using brackets, e.g. [R0-R12]. The special MOVM addressing mode bits W, U, and P are written in the same manner, for example, MOVM.DB.W. A .S suffix allows a MOVM instruction to access user R13 and R14 when in another processor mode. Shifts and rotates in addressing modes are supported by binary operators << (logical left shift), >> (logical right shift), -> (arithmetic right shift), and @> (rotate right); for example R7>>R2 or R2@>2. The assembler does not support indexing by a shifted expression; only names can be doubly indexed.

Any instruction can be followed by a suffix that makes the instruction conditional: .EQ, .NE, and so on, as in the ARM manual, with synonyms .HS (for .CS) and .LO (for .CC), for example ADD.NE. Arithmetic and logical instructions can have a .S suffix, as ARM allows, to set condition codes.

The syntax of the MCR and MRC coprocessor instructions is largely as in the manual, with the usual adjustments. The assembler directly supports only the ARM floating-point coprocessor operations used by the compiler: CMP, ADD, SUB, MUL, and DIV, all with F or D suffix selecting single or double precision. Floating-point load or store become MOVF and MOVD. Conversion instructions are also specified by moves: MOVWD, MOVWF, MOVDW, MOVWD, MOVFD, and MOVDF.

Rc The Plan 9 Shell

Tom Duff
td@plan9.bell-labs.com

ABSTRACT

Rc is a command interpreter for Plan 9 that provides similar facilities to UNIX s Bourne shell, with some small additions and less idiosyncratic syntax. This paper uses numerous examples to describe *rc* s features, and contrasts *rc* with the Bourne shell, a model that many readers will be familiar with.

1. Introduction

Rc is similar in spirit but different in detail from UNIX s Bourne shell. This paper describes *rc* s principal features with many small examples and a few larger ones. It assumes familiarity with the Bourne shell.

2. Simple commands

For the simplest uses *rc* has syntax familiar to Bourne-shell users. All of the following behave as expected:

```
date
cat /lib/news/build
who >user.names
who >>user.names
wc <file
echo [a-f]*.c
who | wc
who; date
vc *.c &
mk && v.out /*/bin/fb/*
rm -r junk || echo rm failed!
```

3. Quotation

An argument that contains a space or one of *rc* s other syntax characters must be enclosed in apostrophes (’):

```
rm 'odd file name'
```

An apostrophe in a quoted argument must be doubled:

```
echo 'How''s your father?'
```

4. Patterns

An unquoted argument that contains any of the characters * ? [is a pattern to be matched against file names. A * character matches any sequence of characters, ? matches any single character, and [*class*] matches any character in the class, unless the first character of *class* is ~, in which case the class is complemented. The *class* may also contain pairs of characters separated by -, standing for all characters lexically between the two. The character / must appear explicitly in a pattern, as must the path name components . and ... A pattern is replaced by a list of arguments, one for each path name matched, except that a pattern matching no names is not replaced by the empty list; rather it stands for itself.

5. Variables

UNIX s Bourne shell offers string-valued variables. *Rc* provides variables whose values are lists of arguments   that is, arrays of strings. This is the principal difference between *rc* and traditional UNIX command interpreters. Variables may be given values by typing, for example:

```
path=(.   /bin)
user=td
font=/lib/font/bit/pelm/ascii.9.font
```

The parentheses indicate that the value assigned to path is a list of two strings. The variables user and font are assigned lists containing a single string.

The value of a variable can be substituted into a command by preceding its name with a $, like this:

```
echo  $path
```

If path had been set as above, this would be equivalent to

```
echo  . /bin
```

Variables may be subscripted by numbers or lists of numbers, like this:

```
echo  $path(2)
echo  $path(2   1  2)
```

These are equivalent to

```
echo  /bin
echo  /bin  . /bin
```

There can be no space separating the variable s name from the left parenthesis; otherwise, the subscript would be considered a separate parenthesized list.

The number of strings in a variable can be determined by the $# operator. For example,

```
echo  $#path
```

would print 2 for this example.

The following two assignments are subtly different:

```
empty=()
null=''
```

The first sets empty to a list containing no strings. The second sets null to a list containing a single string, but the string contains no characters.

Although these may seem like more or less the same thing (in Bourne s shell, they are indistinguishable), they behave differently in almost all circumstances. Among other things

```
echo  $#empty
```

prints 0, whereas

```
echo  $#null
```

prints 1.

All variables that have never been set have the value ().

Occasionally, it is convenient to treat a variable's value as a single string. The elements of a string are concatenated into a single string, with spaces between the elements, by the $" operator. Thus, if we set

```
list=(How now brown cow)
string=$"list
```

then both

```
echo $list
```

and

```
echo $string
```

cause the same output, viz:

```
How now brown cow
```

but

```
echo $#list $#string
```

will output

```
4 1
```

because $list has four members, but $string has a single member, with three spaces separating its words.

6. Arguments

When *rc* is reading its input from a file, the file has access to the arguments supplied on *rc*'s command line. The variable $* initially has the list of arguments assigned to it. The names $1, $2, etc. are synonyms for $*(1), $*(2), etc. In addition, $0 is the name of the file from which *rc*'s input is being read.

7. Concatenation

Rc has a string concatenation operator, the caret ^, to build arguments out of pieces.

```
echo hully^gully
```

is exactly equivalent to

```
echo hullygully
```

Suppose variable i contains the name of a command. Then

```
vc $i^.c
vl -o $1 $i^.v
```

might compile the command's source code, leaving the result in the appropriate file.

Concatenation distributes over lists. The following

```
echo (a b c)^(1 2 3)
src=(main subr io)
cc $src^.c
```

are equivalent to

```
echo a1 b2 c3
cc main.c subr.c io.c
```

In detail, the rule is: if both operands of ^ are lists of the same non-zero number of strings, they are concatenated pairwise. Otherwise, if one of the operands is a single string, it is concatenated with each member of the other operand in turn. Any other combination of operands is an error.

8. Free carets

User demand has dictated that *rc* insert carets in certain places, to make the syntax look more like the Bourne shell. For example, this:

```
cc -$flags $stems.c
```

is equivalent to

```
cc -^$flags $stems^.c
```

In general, *rc* will insert ^ between two arguments that are not separated by white space. Specifically, whenever one of $'` follows a quoted or unquoted word, or an unquoted word follows a quoted word with no intervening blanks or tabs, an implicit ^ is inserted between the two. If an unquoted word immediately following a $ contains a character other than an alphanumeric, underscore or *, a ^ is inserted before the first such character.

9. Command substitution

It is often useful to build an argument list from the output of a command. *Rc* allows a command, enclosed in braces and preceded by a left quote, `{...}, anywhere that an argument is required. The command is executed and its standard output captured. The characters stored in the variable ifs are used to split the output into arguments. For example,

```
cat `{ls -tr|sed 10q}
```

will concatenate the ten oldest files in the current directory in temporal order, given the default ifs setting of space, tab, and newline.

10. Pipeline branching

The normal pipeline notation is general enough for almost all cases. Very occasionally it is useful to have pipelines that are not linear. Pipeline topologies more general than trees can require arbitrarily large pipe buffers, or worse, can cause deadlock. *Rc* has syntax for some kinds of non-linear but treelike pipelines. For example,

```
cmp <{old} <{new}
```

will regression-test a new version of a command. < or > followed by a command in braces causes the command to be run with its standard output or input attached to a pipe. The parent command (cmp in the example) is started with the other end of the pipe attached to some file descriptor or other, and with an argument that will connect to the pipe when opened (e.g., /dev/fd/6). Some commands are unprepared to deal with input files that turn out not to be seekable. For example diff needs to read its input twice.

11. Exit status

When a command exits it returns status to the program that executed it. On Plan 9 status is a character string describing an error condition. On normal termination it is empty.

Rc captures command exit status in the variable $status. For a simple command the value of $status is just as described above. For a pipeline $status is set to the concatenation of the statuses of the pipeline components with | characters for separators.

Rc has a several kinds of control flow, many of them conditioned by the status returned from previously executed commands. Any `$status` containing only `0` s and `|` s has boolean value *true*. Any other status is *false*.

12. Command grouping

A sequence of commands enclosed in `{}` may be used anywhere a command is required. For example:

```
{sleep 3600;echo 'Time''s up!'}&
```

will wait an hour in the background, then print a message. Without the braces,

```
sleep 3600;echo 'Time''s up!'&
```

would lock up the terminal for an hour, then print the message in the background.

13. Control flow `for`

A command may be executed once for each member of a list by typing, for example:

```
for(i in printf scanf putchar) look $i /usr/td/lib/dw.dat
```

This looks for each of the words `printf`, `scanf` and `putchar` in the given file. The general form is

for(*name* in *list*) *command*

or

for(*name*) *command*

In the first case *command* is executed once for each member of *list* with that member assigned to variable *name*. If the clause `in` *list* is missing, `in $*` is assumed.

14. Conditional execution `if`

Rc also provides a general if-statement. For example:

```
for(i in *.c) if(cpp $i >/tmp/$i) vc /tmp/$i
```

runs the C compiler on each C source program that cpp processes without error. An `if not` statement provides a two-tailed conditional. For example:

```
for(i){
    if(test -f /tmp/$i) echo $i already in /tmp
    if not cp $i /tmp
}
```

This loops over each file in `$*`, copying to `/tmp` those that do not already appear there, and printing a message for those that do.

15. Control flow `while`

Rc s while statement looks like this:

```
while(newer subr.v subr.c) sleep 5
```

This waits until `subr.v` is newer than `subr.c`, presumably because the C compiler finished with it.

If the controlling command is empty, the loop will not terminate. Thus,

```
while() echo y
```

emulates the *yes* command.

16. Control flow — switch

Rc provides a switch statement to do pattern-matching on arbitrary strings. Its general form is

```
switch(word){
case pattern ...
	commands
case pattern ...
	commands
...
}
```

Rc attempts to match the word against the patterns in each case statement in turn. Patterns are the same as for filename matching, except that / and . and .. need not be matched explicitly.

If any pattern matches, the commands following that case up to the next case (or the end of the switch) are executed, and execution of the switch is complete. For example,

```
switch($#*){
case 1
	cat >>$1
case 2
	cat >>$2 <$1
case *
	echo 'Usage: append [from] to'
}
```

is an append command. Called with one file argument, it appends its standard input to the named file. With two, the first is appended to the second. Any other number elicits an error message.

The built-in ~ command also matches patterns, and is often more concise than a switch. Its arguments are a string and a list of patterns. It sets $status to true if and only if any of the patterns matches the string. The following example processes option arguments for the *man*(1) command:

```
opt=()
while(~ $1 -* [1-9] 10){
	switch($1){
	case [1-9] 10
		sec=$1 secn=$1
	case -f
		c=f s=f
	case -[qwnt]
		cmd=$1
	case -T*
		T=$1
	case -*
		opt=($opt $1)
	}
	shift
}
```

17. Functions

Functions may be defined by typing

```
fn name { commands }
```

Subsequently, whenever a command named *name* is encountered, the remainder of the command's argument list will assigned to $* and *rc* will execute the *commands*. The value of $* will be restored on completion. For example:

```
fn g {
    grep $1 *.[hcyl]
}
```

defines g *pattern* to look for occurrences of *pattern* in all program source files in the current directory.

Function definitions are deleted by writing

```
fn name
```

with no function body.

18. Command execution

Rc does one of several things to execute a simple command. If the command name is the name of a function defined using fn, the function is executed. Otherwise, if it is the name of a built-in command, the built-in is executed directly by *rc*. Otherwise, directories mentioned in the variable $path are searched until an executable file is found. Extensive use of the $path variable is discouraged in Plan 9. Instead, use the default (. /bin) and bind what you need into /bin.

19. Built-in commands

Several commands are executed internally by *rc* because they are difficult to implement otherwise.

. [-i] *file ...*
: Execute commands from *file*. $* is set for the duration to the reminder of the argument list following *file*. $path is used to search for *file*. Option -i indicates interactive input a prompt (found in $prompt) is printed before each command is read.

builtin *command ...*
: Execute *command* as usual except that any function named *command* is ignored. For example,

```
fn cd{
    builtin cd $* && pwd
}
```

defines a replacement for the cd built-in (see below) that announces the full name of the new directory.

cd [*dir*]
: Change the current directory to *dir*. The default argument is $home. $cdpath is a list of places in which to search for *dir*.

eval [*arg ...*]
: The arguments are concatenated (separated by spaces) into a string, read as input to *rc*, and executed. For example,

```
x='$y'
y=Doody
eval echo Howdy, $x
```

would echo

```
Howdy, Doody
```

since the arguments of eval would be

```
echo Howdy, $y
```

after substituting $x for

exec [*command ...*]
: *Rc* replaces itself with the given *command*. This is like a *goto* — *rc* does not wait for the command to exit, and does not return to read any more commands.

exit [*status*]
: *Rc* exits immediately with the given status. If none is given, the current value of $status is used.

flag *f* [+−]
: This command manipulates and tests the command line flags (described below).

```
flag f +
```

sets flag *f*.

```
flag f -
```

clears flag *f*.

```
flag f
```

tests flag *f*, setting $status appropriately. Thus

```
if(flag x) flag v +
```

sets the -v flag if the -x flag is already set.

rfork [nNeEsfF]
: This uses the Plan 9 *rfork* system entry to put *rc* into a new process group with the following attributes:

Flag	Name	Function
n	RFNAMEG	Make a copy of the parent's name space
N	RFCNAMEG	Start with a new, empty name space
e	RFENVG	Make a copy of the parent's environment
E	RFCENVG	Start with a new, empty environment
s	RFNOTEG	Make a new note group
f	RFFDG	Make a copy of the parent's file descriptor space
F	RFCFDG	Make a new, empty file descriptor space

Section *fork* (2) of the Programmer's Manual describes these attributes in more detail.

shift [*n*]
: Delete the first *n* (default 1) elements of $*.

wait [*pid*]
: Wait for the process with the given *pid* to exit. If no *pid* is given, all outstanding processes are waited for.

whatis *name ...*
: Print the value of each *name* in a form suitable for input to *rc*. The output is an assignment to a variable, the definition of a function, a call to builtin for a built-in command, or the path name of a binary program. For example,

```
whatis path g cd who
```

might print

```
path=(. /bin)
fn g {gre -e $1 *.[hycl]}
builtin cd
/bin/who
```

~ *subject pattern ...*

The *subject* is matched against each *pattern* in turn. On a match, $status is set to true. Otherwise, it is set to 'no match'. Patterns are the same as for filename matching. The *patterns* are not subjected to filename replacement before the ~ command is executed, so they need not be enclosed in quotation marks, unless of course, a literal match for * [or ? is required. For example

```
~ $1 ?
```

matches any single character, whereas

```
~ $1 '?'
```

only matches a literal question mark.

20. Advanced I/O Redirection

Rc allows redirection of file descriptors other than 0 and 1 (standard input and output) by specifying the file descriptor in square brackets [ ] after the < or >. For example,

```
vc junk.c >[2]junk.diag
```

saves the compiler's diagnostics from standard error in junk.diag.

File descriptors may be replaced by a copy, in the sense of *dup*(2), of an already-open file by typing, for example

```
vc junk.c >[2=1]
```

This replaces file descriptor 2 with a copy of file descriptor 1. It is more useful in conjunction with other redirections, like this

```
vc junk.c >junk.out >[2=1]
```

Redirections are evaluated from left to right, so this redirects file descriptor 1 to junk.out, then points file descriptor 2 at the same file. By contrast,

```
vc junk.c >[2=1] >junk.out
```

redirects file descriptor 2 to a copy of file descriptor 1 (presumably the terminal), and then directs file descriptor 1 to a file. In the first case, standard and diagnostic output will be intermixed in junk.out. In the second, diagnostic output will appear on the terminal, and standard output will be sent to the file.

File descriptors may be closed by using the duplication notation with an empty right-hand side. For example,

```
vc junk.c >[2=]
```

will discard diagnostics from the compilation.

Arbitrary file descriptors may be sent through a pipe by typing, for example,

```
vc junk.c |[2] grep -v '^$'
```

This deletes blank lines from the C compiler's error output. Note that the output of grep still appears on file descriptor 1.

Occasionally you may wish to connect the input side of a pipe to some file descriptor other than zero. The notation

```
cmd1 |[5=19] cmd2
```

creates a pipeline with cmd1 s file descriptor 5 connected through a pipe to cmd2 s file descriptor 19.

21. Here documents

Rc procedures may include data, called here documents , to be provided as input to commands, as in this version of the *tel* command

```
for(i) grep $i <<!
...
tor 2T-402 2912
kevin 2C-514 2842
bill 2C-562 7214
...
!
```

A here document is introduced by the redirection symbol << , followed by an arbitrary EOF marker (! in the example). Lines following the command, up to a line containing only the EOF marker are saved in a temporary file that is connected to the command s standard input when it is run.

Rc does variable substitution in here documents. The following command:

```
ed $3 <<EOF
g/$1/s//$2/g
w
EOF
```

changes all occurrences of $1 to $2 in file $3 . To include a literal $ in a here document, type $$. If the name of a variable is followed immediately by ^, the caret is deleted.

Variable substitution can be entirely suppressed by enclosing the EOF marker following << in quotation marks, as in <<'EOF' .

Here documents may be provided on file descriptors other than 0 by typing, for example,

```
cmd <<[4]End
...
End
```

If a here document appears within a compound block, the contents of the document must be after the whole block:

```
for(i in $*){
        mail $i <<EOF
}
words to live by
EOF
```

22. Catching Notes

Rc scripts normally terminate when an interrupt is received from the terminal. A function with the name of a UNIX signal, in lower case, is defined in the usual way, but called when *rc* receives the corresponding note. The *notify* (2) section of the Programmer s Manual discusses notes in some detail. Notes of interest are:

sighup
: The note was hangup . Plan 9 sends this when the terminal has disconnected from *rc*.

`sigint`
: The note was *interrupt*, usually sent when the interrupt character (ASCII DEL) is typed on the terminal.

`sigterm`
: The note was *kill*, normally sent by *kill*(1).

`sigexit`
: An artificial note sent when *rc* is about to exit.

As an example,

```
fn sigint{
	rm /tmp/junk
	exit
}
```

sets a trap for the keyboard interrupt that removes a temporary file before exiting.

Notes will be ignored if the note routine is set to `{}`. Signals revert to their default behavior when their handlers' definitions are deleted.

23. Environment

The environment is a list of name-value pairs made available to executing binaries. On Plan 9, the environment is stored in a file system named `#e`, normally mounted on `/env`. The value of each variable is stored in a separate file, with components terminated by zero bytes. (The file system is maintained entirely in core, so no disk or network access is involved.) The contents of `/env` are shared on a per-process group basis — when a new process group is created it effectively attaches `/env` to a new file system initialized with a copy of the old one. A consequence of this organization is that commands can change environment entries and see the changes reflected in *rc*.

Functions also appear in the environment, named by prefixing `fn#` to their names, like `/env/fn#roff`.

24. Local Variables

It is often useful to set a variable for the duration of a single command. An assignment followed by a command has this effect. For example

```
a=global
a=local echo $a
echo $a
```

will print

```
local
global
```

This works even for compound commands, like

```
f=/fairly/long/file/name {
	{ wc $f; spell $f; diff $f.old $f } |
	  pr -h 'Facts about '$f | lp -dfn
}
```

25. Examples — *cd, pwd*

Here is a pair of functions that provide enhanced versions of the standard `cd` and `pwd` commands. (Thanks to Rob Pike for these.)

```
ps1='% '            # default prompt
tab='	'            # a tab character
fn cd{
  builtin cd $1 &&
  switch($#*){
  case 0
    dir=$home
    prompt=($ps1 $tab)
  case *
    switch($1)
    case /*
      dir=$1
      prompt=(`{basename `{pwd}}^$ps1 $tab)
    case */* ..*
      dir=()
      prompt=(`{basename `{pwd}}^$ps1 $tab)
    case *
      dir=()
      prompt=($1^$ps1 $tab)
    }
  }
}
fn pwd{
  if(~ $#dir 0)
    dir=`{/bin/pwd}
  echo $dir
}
```

Function `pwd` is a version of the standard `pwd` that caches its value in variable `$dir`, because the genuine `pwd` can be quite slow to execute. (Recent versions of Plan 9 have very fast implementations of `pwd`, reducing the advantage of the `pwd` function.)

Function `cd` calls the `cd` built-in, and checks that it was successful. If so, it sets `$dir` and `$prompt`. The prompt will include the last component of the current directory (except in the home directory, where it will be null), and `$dir` will be reset either to the correct value or to `()`, so that the `pwd` function will work correctly.

26. Examples *man*

The *man* command prints pages of the Programmer's Manual. It is called, for example, as

```
man 2 sinh
man rc
man -t cat
```

In the first case, the page for *sinh* in section 2 is printed. In the second case, the manual page for *rc* is printed. Since no manual section is specified, all sections are searched for the page, and it is found in section 1. In the third case, the page for *cat* is typeset (the `-t` option).

```
cd /sys/man || {
  echo $0: No manual! >[1=2]
  exit 1
}
NT=n  # default nroff
s='*' # section, default try all
for(i) switch($i){
case -t
  NT=t
case -n
  NT=n
case -*
  echo Usage: $0 '[-nt] [section] page ...' >[1=2]
  exit 1
case [1-9] 10
  s=$i
case *
  eval 'pages='$s/$i
  for(page in $pages){
    if(test -f $page)
      $NT^roff -man $page
    if not
      echo $0: $i not found >[1=2]
  }
}
```

Note the use of eval to make a list of candidate manual pages. Without eval, the * stored in $s would not trigger filename matching — it s enclosed in quotation marks, and even if it weren t, it would be expanded when assigned to $s. Eval causes its arguments to be re-processed by *rc* s parser and interpreter, effectively delaying evaluation of the * until the assignment to $pages.

27. Examples — *holmdel*

The following *rc* script plays the deceptively simple game *holmdel*, in which the players alternately name Bell Labs locations, the winner being the first to mention Holmdel.

This script is worth describing in detail (rather, it would be if it weren t so silly.)

Variable $t is an abbreviation for the name of a temporary file. Including $pid, initialized by *rc* to its process-id, in the names of temporary files insures that their names won t collide, in case more than one instance of the script is running at a time.

Function read s argument is the name of a variable into which a line gathered from standard input is read. $ifs is set to just a newline. Thus read s input is not split apart at spaces, but the terminating newline is deleted.

A handler is set to catch sigint, sigquit, and sighup, and the artificial sigexit signal. It just removes the temporary file and exits.

The temporary file is initialized from a here document containing a list of Bell Labs locations, and the main loop starts.

First, the program guesses a location (in $lab) using the fortune program to pick a random line from the location list. It prints the location, and if it guessed Holmdel, prints a message and exits.

Then it uses the read function to get lines from standard input and validity-check them until it gets a legal name. Note that the condition part of a while can be a compound command. Only the exit status of the last command in the sequence is checked.

```
t=/tmp/holmdel$pid
fn read{
        $1=`{awk '{print;exit}'}
}
ifs='
'       # just a newline
fn sigexit sigint sigquit sighup{
        rm -f $t
        exit
}
cat <<'!' >$t
Allentown
Atlanta
Cedar Crest
Chester
Columbus
Elmhurst
Fullerton
Holmdel
Indian Hill
Merrimack Valley
Morristown
Neptune
Piscataway
Reading
Short Hills
South Plainfield
Summit
Whippany
West Long Branch
!
while(){
   lab=`{fortune $t}
   echo $lab
   if(~ $lab Holmdel){
      echo You lose.
      exit
   }
   while(read lab; ! grep -i -s $lab $t) echo No such location.
   if(~ $lab [hH]olmdel){
      echo You win.
      exit
   }
}
```

Again, if the result is Holmdel, it prints a message and exits. Otherwise it goes back to the top of the loop.

28. Design Principles

Rc draws heavily from Steve Bourne's /bin/sh. Any successor of the Bourne shell is bound to suffer in comparison. I have tried to fix its best-acknowledged shortcomings and to simplify things wherever possible, usually by omitting inessential features. Only when irresistibly tempted have I introduced novel ideas. Obviously I have tinkered extensively with Bourne's syntax.

The most important principle in *rc*'s design is that it's not a macro processor. Input is never scanned more than once by the lexical and syntactic analysis code (except, of course, by the eval command, whose *raison d'être* is to break the rule).

Bourne shell scripts can often be made to run wild by passing them arguments containing spaces. These will be split into multiple arguments using IFS, often at inopportune times. In *rc*, values of variables, including command line arguments, are not re-read when substituted into a command. Arguments have presumably been scanned in the parent process, and ought not to be re-read.

Why does Bourne re-scan commands after variable substitution? He needs to be able to store lists of arguments in variables whose values are character strings. If we eliminate re-scanning, we must change the type of variables, so that they can explicitly carry lists of strings.

This introduces some conceptual complications. We need a notation for lists of words. There are two different kinds of concatenation, for strings $a^$b, and lists ($a $b). The difference between () and '' is confusing to novices, although the distinction is arguably sensible: a null argument is not the same as no argument.

Bourne also rescans input when doing command substitution. This is because the text enclosed in back-quotes is not a string, but a command. Properly, it ought to be parsed when the enclosing command is, but this makes it difficult to handle nested command substitutions, like this:

```
size=`wc -l \`ls -t|sed 1q\``
```

The inner back-quotes must be escaped to avoid terminating the outer command. This can get much worse than the above example; the number of \'s required is exponential in the nesting depth. *Rc* fixes this by making the backquote a unary operator whose argument is a command, like this:

```
size=`{wc -l `{ls -t|sed 1q}}
```

No escapes are ever required, and the whole thing is parsed in one pass.

For similar reasons *rc* defines signal handlers as though they were functions, instead of associating a string with each signal, as Bourne does, with the attendant possibility of getting a syntax error message in response to typing the interrupt character. Since *rc* parses input when typed, it reports errors when you make them.

For all this trouble, we gain substantial semantic simplifications. There is no need for the distinction between $* and $@. There is no need for four types of quotation, nor the extremely complicated rules that govern them. In *rc* you use quotation marks when you want a syntax character to appear in an argument, or an argument that is the empty string, and at no other time. IFS is no longer used, except in the one case where it was indispensable: converting command output into argument lists during command substitution.

This also avoids an important UNIX security hole. In UNIX, the *system* and *popen* functions call /bin/sh to execute a command. It is impossible to use either of these routines with any assurance that the specified command will be executed, even if the caller of *system* or *popen* specifies a full path name for the command. This can be devastating if it occurs in a set-userid program. The problem is that IFS is used to split the command into words, so an attacker can just set IFS=/ in his environment and leave a Trojan horse named usr or bin in the current working directory before running the privileged program. *Rc* fixes this by never rescanning input for any reason.

Most of the other differences between *rc* and the Bourne shell are not so serious. I eliminated Bourne's peculiar forms of variable substitution, like

```
echo ${a=b} ${c-d} ${e?error}
```

because they are little used, redundant and easily expressed in less abstruse terms. I deleted the builtins export, readonly, break, continue, read, return, set, times and unset because they seem redundant or only marginally useful.

Where Bourne s syntax draws from Algol 68, *rc* s is based on C or Awk. This is harder to defend. I believe that, for example

```
if(test -f junk) rm junk
```

is better syntax than

```
if test -f junk; then rm junk; fi
```

because it is less cluttered with keywords, it avoids the semicolons that Bourne requires in odd places, and the syntax characters better set off the active parts of the command.

The one bit of large-scale syntax that Bourne unquestionably does better than *rc* is the `if` statement with `else` clause. *Rc* s `if` has no terminating `fi`-like bracket. As a result, the parser cannot tell whether or not to expect an `else` clause without looking ahead in its input. The problem is that after reading, for example

```
if(test -f junk) echo junk found
```

in interactive mode, *rc* cannot decide whether to execute it immediately and print `$prompt(1)`, or to print `$prompt(2)` and wait for the `else` to be typed. In the Bourne shell, this is not a problem, because the `if` command must end with `fi`, regardless of whether it contains an `else` or not.

Rc s admittedly feeble solution is to declare that the `else` clause is a separate statement, with the semantic proviso that it must immediately follow an `if`, and to call it `if not` rather than `else`, as a reminder that something odd is going on. The only noticeable consequence of this is that the braces are required in the construction

```
for(i){
    if(test -f $i) echo $i found
    if not echo $i not found
}
```

and that *rc* resolves the dangling `else` ambiguity in opposition to most peoples expectations.

It is remarkable that in the four most recent editions of the UNIX system programmer s manual the Bourne shell grammar described in the manual page does not admit the command `who|wc`. This is surely an oversight, but it suggests something darker: nobody really knows what the Bourne shell s grammar is. Even examination of the source code is little help. The parser is implemented by recursive descent, but the routines corresponding to the syntactic categories all have a flag argument that subtly changes their operation depending on the context. *Rc* s parser is implemented using *yacc*, so I can say precisely what the grammar is.

29. Acknowledgements

Rob Pike, Howard Trickey and other Plan 9 users have been insistent, incessant sources of good ideas and criticism. Some examples in this document are plagiarized from [Bourne], as are most of *rc* s good features.

30. Reference

S. R. Bourne, UNIX Time-Sharing System: The UNIX Shell, Bell System Technical Journal, Volume 57 number 6, July-August 1978

Plumbing and Other Utilities

Rob Pike

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Plumbing is a new mechanism for inter-process communication in Plan 9, specifically the passing of messages between interactive programs as part of the user interface. Although plumbing shares some properties with familiar notions such as cut and paste, it offers a more general data exchange mechanism without imposing a particular user interface.

The core of the plumbing system is a program called the *plumber*, which handles all messages and dispatches and reformats them according to configuration rules written in a special-purpose language. This approach allows the contents and context of a piece of data to define how it is handled. Unlike with drag and drop or cut and paste, the user doesn't need to deliver the data; the contents of a plumbing message, as interpreted by the plumbing rules, determine its destination.

The plumber has an unusual architecture: it is a language-driven file server. This design has distinct advantages. It makes plumbing easy to add to an existing, Unix-like command environment; it guarantees uniform handling of inter-application messages; it off-loads from those applications most of the work of extracting and dispatching messages; and it works transparently across a network.

Introduction

Data moves from program to program in myriad ways. Command-line arguments, shell pipelines, cut and paste, drag and drop, and other user interface techniques all provide some form of interprocess communication. Then there are tricks associated with special domains, such as HTML hyperlinks or the heuristics mail readers use to highlight URLs embedded in mail messages. Some systems provide implicit ways to automate the attachment of program to data—the best known examples are probably the resource forks in MacOS and the file name extension associations in Microsoft Windows—but in practice humans must too often carry their data from program to program.

Why should a human do the work? Usually there is one obvious thing to do with a piece of data, and the data itself suggests what this is. Resource forks and associations speak to this issue directly, but statically and narrowly and with little opportunity to control the behavior. Mechanisms with more generality, such as cut and paste or drag and drop, demand too much manipulation by the user and are (therefore) too error-prone.

We want a system that, given a piece of data, hands it to the appropriate application by default with little or no human intervention, while still permitting the user to override the defaults if desired.

The plumbing system is an attempt to address some of these issues in a single, coherent, central way. It provides a mechanism for formatting and sending arbitrary messages between applications, typically interactive programs such as text editors, web browsers, and the window system, under the control of a central message-handling server called the *plumber*. Interactive programs provide application-specific connections to the plumber, triggering with minimal user action the transfer of data or control to other programs. The result is similar to a hypertext system in which all the links are implicit, extracted automatically by examining the data and the user's actions. It obviates cut and paste and other such hand-driven interprocess communication mechanisms. Plumbing delivers the goods to the right place automatically.

Overview

The plumber is implemented as a Plan 9 file server [Pike93]; programs send messages by writing them to the plumber's file /mnt/plumb/send, and receive messages by reading them from *ports*, which are other plumber files in /mnt/plumb. For example, /mnt/plumb/edit is by convention the file from which a text editor reads messages requesting it to open and display a file for editing. (See Figure 1.)

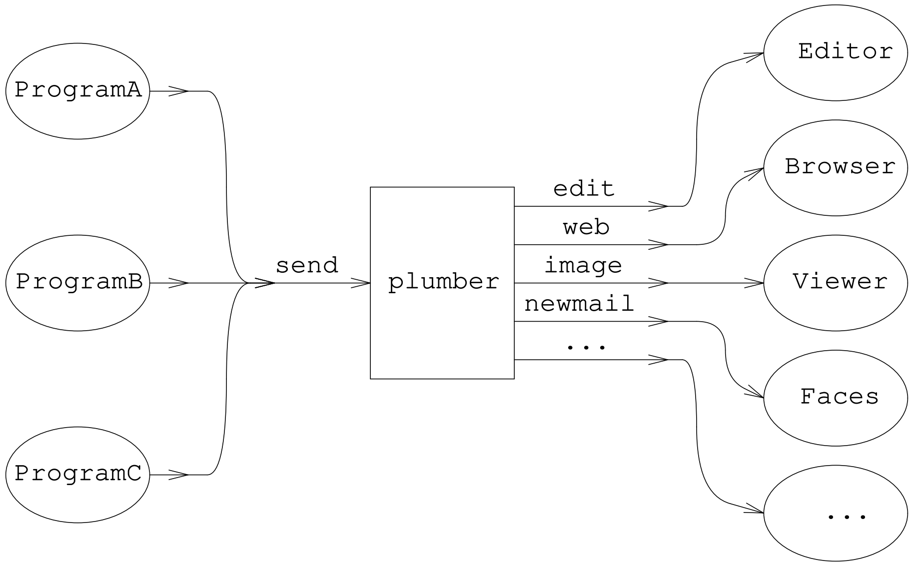

Figure 1. The plumber controls the flow of messages between applications. Programs write to the file send and receive on ports of various names representing services such as edit or web. Although the figure doesn't illustrate it, some programs may both send and receive messages, and some ports are read by multiple applications.

The plumber takes messages from the send file and interprets their contents using rules defined by a special-purpose pattern-action language. The language specifies any rewriting of the message that is to be done by the plumber and defines how to dispose of a message, such as by sending it to a port or starting a new process to handle it.

The behavior is best described by example. Imagine that the user has, in a terminal emulator window, just run a compilation that has failed:

```
% make
cc -c rmstar.c
rmstar.c:32: syntax error
...
```

The user points the typing cursor somewhere in the string rmstar.c:32: and executes the plumb menu entry. This causes the terminal emulator to format a plumbing message containing the entire string surrounding the cursor, rmstar:32:, and to write it to /mnt/plumb/send. The plumber receives this message and compares it sequentially to the various patterns in its configuration. Eventually, it will find one that breaks the string into pieces, rmstar.c, a colon, 32, and the final colon. Other associated patterns verify that rmstar.c is a file in the current directory of the program generating the message, and that 32 looks like a line number within it. The plumber rewrites the message, setting the data to the string rmstar.c and attaching an indication that 32 is a line number to display. Finally, it sends the resulting message to the edit port. The text editor picks up the message, opens rmstar.c (if it's not already open) and highlights line 32, the location of the syntax error.

From the user's point of view, this process is simple: the error message appears, it is plumbed, and the editor jumps to the problem.

Of course, there are many different ways to cause compiler messages to pop up the source of an error, but the design of the plumber addresses more general issues than the specific goal of shortening the compile/debug/edit cycle. It facilitates the general exchange of data among programs, interactive or otherwise, throughout the environment, and its architecture a central, language-driven file server although unusual, has distinct advantages. It makes plumbing easy to add to an existing, Unix-like command environment; it guarantees uniform handling of inter-application messages; it off-loads from those applications most of the work of extracting and dispatching messages; and it works transparently and effortlessly across a network.

This paper is organized bottom-up, beginning with the format of the messages and proceeding through the plumbing language, the handling of messages, and the interactive user interface. The last sections discuss the implications of the design and compare the plumbing system to other environments that provide similar services.

Format of messages

Since the language that controls the plumber is defined in terms of the contents of plumbing messages, we begin by describing their layout.

Plumbing messages have a fixed-format textual header followed by a free-format data section. The header consists of six lines of text, in set order, each specifying a property of the message. Any line may be blank except the last, which is the length of the data portion of the message, as a decimal string. The lines are, in order:

The source application, the name of the program generating the message.

The destination port, the name of the port to which the messages should be sent.

The working directory in which the message was generated.

The type of the data, analogous to a MIME type, such as text or image/gif.

Attributes of the message, given as blank-separated *name=value* pairs. The values may be quoted to protect blanks or quotes; values may not contain newlines.

The length of the data section, in bytes.

Here is a sample message, one that (conventionally) tells the editor to open the file /usr/rob/src/mem.c and display line 27 within it:

```
plumbtest
edit
/usr/rob/src
text
addr=27
5
mem.c
```

Because in general it need not be text, the data section of the message has no terminating newline.

A library interface simplifies the processing of messages by translating them to and from a data structure, Plumbmsg, defined like this:

```
typedef struct Plumbattr Plumbattr;
typedef struct Plumbmsg  Plumbmsg;

struct Plumbmsg
{
    char       *src;    /* source application */
    char       *dst;    /* destination port */
    char       *wdir;   /* working directory */
    char       *type;   /* type of data */
    Plumbattr  *attr;   /* attribute list */
    int        ndata;   /* #bytes of data */
    char       *data;
};

struct Plumbattr
{
    char       *name;
    char       *value;
    Plumbattr  *next;
};
```

The library also includes routines to send a message, receive a message, manipulate the attribute list, and so on.

The Language

An instance of the plumber runs for each user on each terminal or workstation. It begins by reading its rules from the file lib/plumbing in the users home directory, which in turn may use include statements to interpolate macro definitions and rules from standard plumbing rule libraries stored in /sys/lib/plumb.

The rules control the processing of messages. They are written in a pattern-action language comprising a sequence of blank-line-separated *rule sets*, each of which contains one or more *patterns* followed by one or more *actions*. Each incoming message is compared against the rule sets in order. If all the patterns within a rule set succeed, one of the associated actions is taken and processing completes.

The syntax of the language is straightforward. Each rule (pattern or action) has three components, separated by white space: an *object*, a *verb*, and optional *arguments*. The object identifies a part of the message, such as the source application (src), or the data portion of the message (data), or the rules own arguments (arg); or it is the keyword plumb, which introduces an action. The verb specifies an operation to perform on the object, such as the word is to require precise equality between the object and the argument, or isdir to require that the object be the name of a directory.

For instance, this rule set sends messages containing the names of files ending in .gif, .jpg, etc. to a program, page, to display them; it is analogous to a Windows association rule:

```
# image files go to page
type is text
data matches '[a-zA-Z0-9_\-./]+'
data matches '([a-zA-Z0-9_\-./]+)\.(jpe?g|gif|bit|tiff|ppm)'
arg isfile $0
plumb to image
plumb client page -wi
```

(Lines beginning with # are commentary.) Consider how this rule handles the following message, annotated down the left column for clarity:

src	plumbtest
dst	
wdir	/usr/rob/pics
type	text
attr	
ndata	9
data	horse.gif

The `is` verb specifies a precise match, and the `type` field of the message is the string `text`, so the first pattern succeeds. The `matches` verb invokes a regular expression pattern match of the object (here `data`) against the argument pattern. Both `matches` patterns in this rule set will succeed, and in the process set the variables `$0` to the matched string, `$1` to the first parenthesized submatch, and so on (analogous to `&`, `\1`, etc. in `ed`'s regular expressions). The pattern `arg isfile $0` verifies that the named file, `horse.gif`, is an actual file in the directory `/usr/rob/pics`. If all the patterns succeed, one of the actions will be executed.

There are two actions in this rule. The `plumb to` rule specifies `image` as the destination port of the message. By convention, the plumber mounts its services in the directory `/mnt/plumb`, so in this case if the file `/mnt/plumb/image` has been opened, the message will be made available to the program reading from it. Note that the message does not name a port, but the rule set that matches the message does, and that is sufficient to dispatch the message. If on the other hand a message matches no rule but has an explicit port mentioned, that too is sufficient.

If no client has opened the `image` port, that is, if the program `page` is not already running, the `plumb client` action gives the execution script to start the application and send the message on its way; the `-wi` arguments tell `page` to create a window and to receive its initial arguments from the plumbing port. The process by which the plumber starts a program is described in more detail in the next section.

It may seem odd that there are two `matches` rules in this example. The reason is related to the way the plumber can use the rules themselves to refine the *data* in the message, somewhat in the manner of Structural Regular Expressions [Pike87a]. For example, consider what happens if the cursor is at the last character of

```
% make nightmare>horse.gif
```

and the user asks to plumb what the cursor is pointing at. The program creating the plumbing message—in this case the terminal emulator running the window—can send the entire white-space-delimited string `nightmare>horse.gif` or even the entire line, and the combination of `matches` rules can determine that the user was referring to the string `horse.gif`. The user could of course select the entire string `horse.gif`, but it's more convenient just to point in the general location and let the machine figure out what should be done. The process is as follows.

The application generating the message adds a special attribute to the message, named `click`, whose numerical value is the offset of the cursor—the selection point—within the data string. This attribute tells the plumber two things: first, that the regular expressions in `matches` rules should be used to identify the relevant data; and second, approximately where the relevant data lies. The plumber will then use the first `matches` pattern to identify the longest leftmost match that touches the cursor, which will extract the string `horse.gif`, and the second pattern will then verify that that names a picture file. The rule set succeeds and the data is winnowed to the matching substring before being sent to its destination.

Each `matches` pattern within a given rule set must match the same portion of the string, which guarantees that the rule set fails to match a string for which the second pattern matches only a portion. For instance, our example rule set should not execute if the data is the string `horse.gift`, and although the first pattern will match `horse.gift`, the second will match only `horse.gif` and the rule set will fail.

The same approach of multiple `matches` rules can be used to exclude, for instance, a terminal period from a file name or URL, so a file name or URL at the end of a sentence is recognized properly.

If a `click` attribute is not specified, all patterns must match the entire string, so the user has an option: he or she may select exactly what data to send, or may instead indicate where the data is by clicking the selection button on the mouse and letting the machine locate the URL or image file name within the text. In other words, the user can control the contents of the message precisely when required, but the default, simplest action in the user interface does the right thing most of the time.

How Messages are Handled in the Plumber

An application creates a message header, fills in whatever fields it wishes to define, attaches the data, and writes the result to the file `send` in the plumber's service directory, `/mnt/plumb`. The plumber receives the message and applies the plumbing rules successively to it. When a rule set matches, the message is dispatched as indicated by that rule set and processing continues with the next message. If no rule set matches the message, the plumber indicates this by returning a write error to the application, that is, the write to `/mnt/plumb/send` fails, with the resulting error string describing the failure. (Plan 9 uses strings rather than pre-defined numbers to describe error conditions.) Thus a program can discover whether a plumbing message has been sent successfully.

After a matching rule set has been identified, the plumber applies a series of rewriting steps to the message. Some rewritings are defined by the rule set; others are implicit. For example, if the message does not specify a destination port, the outgoing message will be rewritten to identify it. If the message does specify the port, the rule set will only match if any `plumb to` action in the rule set names the same port. (If it matches no rule sets, but mentions a port, it will be sent there unmodified.)

The rule set may contain actions that explicitly rewrite components of the message. These may modify the attribute list or replace the data section of the message. Here is a sample rule set that does both. It matches strings of the form `plumb.h` or `plumb.h:27`. If that string identifies a file in the standard C include directory, `/sys/include`, perhaps with an optional line number, the outgoing message is rewritten to contain the full path name and an attribute, `addr`, to hold the line number:

```
# .h files are looked up in /sys/include and passed to edit
type is text
data matches '([a-zA-Z0-9]+\.h)(:([0-9]+))?'
arg isfile /sys/include/$1
data set /sys/include/$1
attr add addr=$3
plumb to edit
```

The `data set` rule replaces the contents of the data, and the `attr add` rule adds a new attribute to the message. The intent of this rule is to permit one to plumb an include file name in a C program to trigger the opening of that file, perhaps at a specified line, in the text editor. A variant of this rule, discussed below, tells the editor how to interpret syntax errors from the compiler, or the output of `grep -n`, both of which use a fixed syntax *file*`:`*line* to identify a line of source.

The Plan 9 text editors interpret the `addr` attribute as the definition of which portion of the file to display. In fact, the real rule includes a richer definition of the address syntax, so one may plumb strings such as `plumb.h:/plumbsend` (using a regular expression after the `:`) to pop up the declaration of a function in a C header file.

Another form of rewriting is that the plumber may modify the attribute list of the message to clarify how to handle the message. The primary example of this involves the treatment of the click attribute, described in the previous section. If the message contains a click attribute and the matching rule set uses it to extract the matching substring from the data, the plumber deletes the click attribute and replaces the data with the matching substring.

Once the message is rewritten, the actions of the matching rule set are examined. If the rule set contains a plumb to action and the corresponding port is open that is, if a program is already reading from that port the message is delivered to the port. The application will receive the message and handle it as it sees fit. If the port is not open, a plumb start or plumb client action will start a new program to handle the message.

The plumb start action is the simpler: its argument specifies a command to run instead of passing on the message; the message is discarded. Here for instance is a rule that, given the process id (pid) of an existing process, starts the acid debugger [Wint94] in a new window to examine that process:

```
# processes go to acid (assuming strlen(pid) >= 2)
type is text
data matches '[a-zA-Z0-9.:_\-/]+'
data matches '[0-9][0-9]+'
arg isdir /proc/$0
plumb start window acid $0
```

(Note the use of multiple matches rules to avoid misfires from strings like party.1999.) The arg isdir rule checks that the pid represents a running process (or broken one; Plan 9 does not create core files but leaves broken processes around for debugging) by checking that the process file system has a directory for that pid [Kill84]. Using this rule, one may plumb the pid string printed by the ps command or by the operating system when the program breaks; the debugger will then start automatically.

The other startup action, plumb client, is used when a program will read messages from the plumbing port. For example, text editors can read files specified as command arguments, so one could use a plumb start rule to begin editing a file. If, however, the editor will read messages from the edit plumbing port, letting it read the message from the port insures that it uses other information in the message, such as the line number to display. The plumb client action is therefore like plumb start, but keeps the message around for delivery when the application opens the port. Here is the full rule set to pass a regular file to the text editor:

```
# existing files, possibly tagged by address, go to editor
type is text
data matches '([.a-zA-Z0-9_/\-]*[a-zA-Z0-9_/\-])('$addr')?'
arg isfile $1
data set $1
attr add addr=$3
plumb to edit
plumb client window $editor
```

If the editor is already running, the plumb to rule causes it to receive the message on the port. If not, the command window $editor will create a new window (using the Plan 9 program window) to run the editor, and once that starts it will open the edit plumbing port as usual and discover this first message already waiting.

The variables $editor and $addr in this rule set are macros defined in the plumbing rules file; they specify the name of the users favorite text editor and a regular expression that matches that editors address syntax, such as line numbers and patterns. This rule set lives in a library of shared plumbing rules that users private rules can build on, so the rule set needs to be adaptable to different editors and their address syntax. The macro definitions for Acme and Sam [Pike94, Pike87b] look like this:

```
editor=acme
# or editor=sam
addrelem='((#?[0-9]+)|(/[A-Za-z0-9_\^]+/?)|[.$])'
addr=:($addrelem([,;+\-]$addrelem)*)
```

Finally, the application reads the message from the appropriate port, such as /mnt/plumb/edit, unpacks it, and goes to work.

Message Delivery

In summary, a message is delivered by writing it to the send file and having the plumber, perhaps after some rewriting, send it to the destination port or start a new application to handle it. If no destination can be found by the plumber, the original write to the send file will fail, and the application will know the message could not be delivered.

If multiple applications are reading from the destination port, each will receive an identical copy of the message; that is, the plumber implements fan-out. The number of messages delivered is equal to the number of clients that have opened the destination port. The plumber queues the messages and makes sure that each application that opened the port before the message was written gets exactly one copy.

This design minimizes blocking in the sending applications, since the write to the send file can complete as soon as the message has been queued for the appropriate port. If the plumber waited for the message to be read by the recipient, the sender could block unnecessarily. Unfortunately, this design also means that there is no way for a sender to know when the message has been handled; in fact, there are cases when the message will not be delivered at all, such as if the recipient exits while there are still messages in the queue. Since the plumber is part of a user interface, and not an autonomous message delivery system, the decision was made to give the non-blocking property priority over reliability of message delivery. In practice, this tradeoff has worked out well: applications almost always know when a message has failed to be delivered (the write fails because no destination could be found), and those occasions when the sender believes incorrectly that the message has been delivered are both extremely rare and easily recognized by the user—usually because the recipient application has exited.

The Rules File

The plumber begins execution by reading the user's startup plumbing rules file, lib/plumbing. Since the plumber is implemented as a file server, it can also present its current rules as a dynamic file, a design that provides an easily understood way to maintain the rules.

The file /mnt/plumb/rules is the text of the rule set the plumber is currently using, and it may be edited like a regular file to update those rules. To clear the rules, truncate that file; to add a new rule set, append to it:

```
% echo 'type is text
data is self-destruct
plumb start rm -rf $HOME' >> /mnt/plumb/rules
```

This rule set will take effect immediately. If it has a syntax error, the write will fail with an error message from the plumber, such as 'malformed rule' or 'undefined verb'.

To restore the plumber to its startup configuration,

```
% cp /usr/$user/lib/plumbing /mnt/plumb/rules
```

For more sophisticated changes, one can of course use a regular text editor to modify

/mnt/plumb/rules.

This simple way of maintaining an active service could profitably be adopted by other systems. It avoids the need to reboot, to update registries with special tools, or to send asynchronous signals to critical programs.

The User Interface

One unusual property of the plumbing system is that the user interface that programs provide to access it can vary considerably, yet the result is nonetheless a unifying force in the environment. Shells talk to editors, image viewers, and web browsers; debuggers talk to editors; editors talk to themselves; and the window system talks to everybody.

The plumber grew out of some of the ideas of the Acme editor/window-system/user interface [Pike94], in particular its acquisition feature. With a three-button mouse, clicking the right button in Acme on a piece of text tells Acme to get the thing being pointed to. If it is a file name, open the file; if it is a directory, open a viewer for its contents; if a line number, go to that line; if a regular expression, search for it. This one-click access to anything describable textually was very powerful but had several limitations, of which the most important were that Acme's rules for interpreting the text (that is, the implicit hyperlinks) were hard-wired and inflexible, and that they only applied to and within Acme itself. One could not, for example, use Acme's power to open an image file, since Acme is a text-only system.

The plumber addresses these limitations, even with Acme itself: Acme now uses the plumber to interpret the right button clicks for it. When the right button is clicked on some text, Acme constructs a plumbing message much as described above, using the click attribute and the white-space-delimited text surrounding the click. It then writes the message to the plumber; if the write succeeds, all is well. If not, it falls back to its original, internal rules, which will result in a context search for the word within the current document.

If the message is sent successfully, the recipient is likely to be Acme itself, of course: the request may be to open a file, for example. Thus Acme has turned the plumber into an external component of its own operation, while expanding the possibilities; the operation might be to start an image viewer to open a picture file, something Acme cannot do itself. The plumber expands the power of Acme's original user interface.

Traditional menu-driven programs such as the text editor Sam [Pike87b] and the default shell window of the window system 8½ [Pike91] cannot dedicate a mouse button solely to plumbing, but they can certainly dedicate a menu entry. The editing menu for such programs now contains an entry, plumb, that creates a plumbing message using the current selection. (Acme manages to send a message by clicking on the text with one button; other programs require a click with the select button and then a menu operation.) For example, after this happens in a shell window:

```
% make
cc -c shaney.c
shaney.c:232: i undefined
...
```

one can click anywhere on the string shaney.c:232, execute the plumb menu entry, and have line 232 appear in the text editor, be it Sam or Acme—whichever has the edit port open. (If this were an Acme shell window, it would be sufficient to right-click on the string.)

[An interesting side line is how the window system knows what directory the shell is running in; in other words, what value to place in the wdir field of the plumb message. Recall that 8½ is, like many Plan 9 programs, a file server. It now serves a new file, /dev/wdir, that is private to each window. Programs, in particular the Plan 9 shell, rc, can write that file to inform the window system of its current directory. When a cd command is executed in an interactive shell, rc updates the contents of /dev/wdir and plumbing can proceed with local file names.]

Of course, users can plumb image file names, process ids, URLs, and other items any string whose syntax and disposition are defined in the plumbing rules file. An example of how the pieces fit together is the way Plan 9 now handles mail, particularly MIME-encoded messages.

When a new mail message arrives, the mail receiver process sends a plumbing message to the newmail port, which notifies any interested process that new mail is here. The plumbing message contains information about the mail, including its sender, date, and current location in the file system. The interested processes include a program, faces, that gives a graphical display of the mail box using faces to represent the senders of messages [PiPr85], as well as interactive mail programs such as the Acme mail viewer [Pike94]. The user can then click on the face that appears, and the faces program will send another plumbing message, this time to the showmail port. Here is the rule for that port:

```
# faces -> new mail window for message
type is text
data matches '[a-zA-Z0-9_\-./]+'
data matches '/mail/fs/[a-zA-Z0-9/]+/[0-9]+'
plumb to showmail
plumb start window edmail -s $0
```

If a program, such as the Acme mail reader, is reading that port, it will open a new window in which to display the message. If not, the plumb start rule will create a new window and run edmail, a conventional mail reading process, to examine it. Notice how the plumbing connects the components of the interface together the same way regardless of which components are actually being used to view mail.

There is more to the mail story. Naturally, mailboxes in Plan 9 are treated as little file systems, which are synthesized on demand by a special-purpose file server that takes a flat mailbox file and converts it into a set of directories, one per message, with component files containing the header, body, MIME information, and so on. Multi-part MIME messages are unpacked into multi-level directories, like this:

```
% ls -l /mail/fs/mbox/25
d-r-xr-xr-x  M 20 rob rob      0 Nov 21 13:06 /mail/fs/mbox/25/1
d-r-xr-xr-x  M 20 rob rob      0 Nov 21 13:06 /mail/fs/mbox/25/2
--r--r--r--  M 20 rob rob  28678 Nov 21 13:06 /mail/fs/mbox/25/body
--r--r--r--  M 20 rob rob      0 Nov 21 13:06 /mail/fs/mbox/25/cc
...
% mail
25 messages
: 25
From: presotto
Date: Sun Nov 21 13:05:51 EST 1999
To: rob

Check this out.

===> 2/ (image/jpeg) [inline]
        /mail/fs/mbox/25/2/fabio.jpg
:
```

Since the components are all (synthetic) files, the user can plumb the pieces to view embedded pictures, URLs, and so on. Note that the mail program can plumb the contents of inline attachments automatically, without user interaction; in other words, plumbing lets the mailer handle multimedia data without itself interpreting it.

At a more mundane level, a shell command, plumb, can be used to send messages:

```
% cd /usr/rob/src
% plumb mem.c
```

will send the appropriate message to the edit port. A surprising use of the plumb command is in actions within the plumbing rules file. In our lab, we commonly receive Microsoft Word documents by mail, but we do not run Microsoft operating systems on our machines so we cannot view them without at least rebooting. Therefore, when a Word document arrives in mail, we could plumb the .doc file but the text editor could not decode it. However, we have a program, doc2txt, that decodes the Word file format to extract and format the embedded text. The solution is to use plumb in a plumb start action to invoke doc2txt on .doc files and synthesize a plain text file:

```
# rule set for microsoft word documents
type is text
data matches '[a-zA-Z0-9_\-./]+'
data matches '([a-zA-Z0-9_\-./]+)\.doc'
arg isfile $0
plumb start doc2txt $data | \
     plumb -i -d edit -a action=showdata -a filename=$0
```

The arguments to plumb tell it to take standard input as its data rather than the text of the arguments (-i), define the destination port (-d edit), and set a conventional attribute so the editor knows to show the message data itself rather than interpret it as a filename (-a action=showdata) and provide the original filename (-a filename=$0). Now when a user plumbs a .doc file the plumbing rules run a process to extract the text and send it as a temporary file to the editor for viewing. It's imperfect, but it's easy and it beats rebooting.

Another simple example is a rule that turns man pages into hypertext. Manual page entries of the form plumber(1) can be clicked on to pop up a window containing the formatted man page. That man page will in turn contain more such citations, which will also be clickable. The rule is a little like that for Word documents:

```
# man index entries are synthesized
type is text
data matches '([a-zA-Z0-9_\-./]+)\(([0-9])\)'
plumb start man $2 $1 | \
     plumb -i -d edit -a action=showdata -a filename=/man/$1($2)
```

There are many other inventive uses of plumbing. One more should give some of the flavor. We have a shell script, src, that takes as argument the name of an executable binary file. It examines the symbol table of the binary to find the source file from which it was compiled. Since the Plan 9 compilers place full source path names in the symbol table, src can discover the complete file name. That is then passed to plumb, complete with the line number to find the main symbol. For example,

```
% src plumb
```

is all it takes to pop up an editor window on the main routine of the plumb command, beginning at line 39 of /sys/src/cmd/plumb/plumb.c. Like most uses of plumbing, this is not a breakthrough in functionality, but it is a great convenience.

Why This Architecture?

The design of the plumbing system is peculiar: a centralized language-based file server does most of the work, while compared to other systems the applications themselves contribute relatively little. This architecture is deliberate, of course.

That the plumber's behavior is derived from a linguistic description gives the system great flexibility and dynamism rules can be added and changed at will, without rebooting but the existence of a central library of rules ensures that, for most users, the environment behaves in well-established ways.

That the plumber is a file server is perhaps the most unusual aspect of its design, but is also one of the most important. Messages are passed by regular I/O operations on files, so no extra technology such as remote procedure call or request brokers needs to be provided; messages are transmitted by familiar means. Almost every service in Plan 9 is a file server, so services can be exported trivially using the system's remote file system operations [Pike93]. The plumber is no exception; plumbing messages pass routinely across the network to remote applications without any special provision, in contrast to some commercial IPC mechanisms that become significantly more complex when they involve multiple machines. As I write this, my window system is talking to applications running on three different machines, but they all share a single instance of the plumber and so can interoperate to integrate my environment. Plan 9 uses a shared file name space to combine multiple networked machines compute servers, file servers, and interactive workstations into a single computing environment; plumbing's design as a file server is a natural by-product of, and contributor to, the overall system architecture [Pike92].

The centrality of the plumber is also unusual. Other systems tend to let the applications determine where messages will go; consider mail readers that recognize and highlight URLs in the messages. Why should just the mail readers do this, and why should they just do it for URLs? (Acme was guilty of similar crimes.) The plumber, by removing such decisions to a central authority, guarantees that all applications behave the same and simultaneously frees them all from figuring out what's important. The ability for the plumber to excerpt useful data from within a message is critical to the success of this model.

The entire system is remarkably small. The plumber itself is only about two thousand lines of C code. Most applications work fine in a plumbing environment without knowing about it at all; some need trivial changes such as to standardize their error output; a few need to generate and receive plumbing messages. But even to add the ability to send and receive messages in a program such as text editor is short work, involving typically a few dozen lines of code. Plumbing fits well into the existing environment.

But plumbing is new and it hasn't been pushed far enough yet. Most of the work so far has been with textual messages, although the underlying system is capable of handling general data. We plan to reimplement some of the existing data movement operations, such as cut and paste or drag and drop, to use plumbing as their exchange mechanism. Since the plumber is a central message handler, it is an obvious place to store the clipboard. The clipboard could be built as a special port that holds onto messages rather than deleting them after delivery. Since the clipboard would then be held in a plumbing message rather than plain text, as in the current Plan 9 environment, it would become possible to cut and paste arbitrary data without providing new mechanism. In effect, we would be providing a new user interface to the existing plumbing facilities.

Another possible extension is the ability to override plumbing operations interactively. Originally, the plan was to provide a mechanism, perhaps a pop-up menu, that one could use to direct messages, for example to send a PostScript file to the editor rather than the PostScript viewer by naming an explicit destination in the message. Although this deficiency should one day be addressed, it should be done without complicating the interface for invoking the default behavior. Meanwhile, in practice the default behavior seems to work very well in practice as it must if plumbing is to be successful so the lack of overrides is not keenly felt.

Comparison with Other Systems

The ideas of the plumbing system grew from an attempt to generalize the way Acme acquires files and data. Systems further from that lineage also share some properties with plumbing. Most, however, require explicit linking or message passing rather than plumbing s implicit, context-based pattern matching, and none has the plumber s design of a language-based file server.

Reisss FIELD system [Reis95] probably comes the closest to providing the facilities of the plumber. It has a central message-passing mechanism that connects applications together through a combination of a library and a pattern-matching central message dispatcher that handles message send and reply. The main differences between FIELDs message dispatcher and the plumber are first that the plumber is based on a special-purpose language while the FIELD system uses an object-oriented library, second that the plumber has no concept of a reply to a message, and finally that the FIELD system has no concept of port. But the key distinction is probably in the level of use. In FIELD, the message dispatcher is a critical integrating force of the underlying programming environment, handling everything from debugging events to changing the working directory of a program. Plumbing, by contrast, is intended primarily for integrating the user interface of existing tools; it is more modest and very much simpler. The central advantage of the plumber is its convenience and dynamism; the FIELD system does not share the ease with which message dispatch rules can be added or modified.

The inspiration for Acme was the user interface to the object-oriented Oberon system [WiGu92]. Oberon s user interface interprets mouse clicks on strings such as Obj.meth to invoke calls to the method meth of the object Obj . This was the starting point for Acme s middle-button execution [Pike94], but nothing in Oberon is much like Acme s right-button acquisition, which was the starting point for the plumber. Oberon s implicit method-based linking is not nearly as general as the pattern-matched linking of the plumber, nor does its style of user-triggered method call correspond well to the more general idea of inter-application communication of plumbing messages.

Microsofts OLE interface is another relative. It allows one application to *embed* its own data within another s, for example to place an Excel spreadsheet within a Frame document; when Frame needs to format the page, it will start Excel itself, or at least some of its DLLs, to format the spreadsheet. OLE data can only be understood by the application that created it; plumbing messages, by contrast, contain arbitrary data with a rigidly formatted header that will be interpreted by the pattern matcher and the destination application. The plumber s simplified message format may limit its flexibility but makes messages easy and efficient to dispatch and to interpret. At least for the cut-and-paste style of exchange OLE encourages, plumbing gives up some power in return for simplicity, while avoiding the need to invoke a vestigial program (if Excel can be called a vestige) every time the pasted data is examined. Plumbing is also better suited to other styles of data exchange, such as connecting compiler errors to the text editor.

The Hyperbole [Wein] package for Emacs adds hypertext facilities to existing documents. It includes explicit links and, like plumbing, a rule-driven way to form implicit links. Since Emacs is purely textual, like Acme, Hyperbole does not easily extend to driving graphical applications, nor does it provide a general interprocess communication method. For instance, although Hyperbole provides some integration for mail applications, it cannot provide the glue that allows a click on a face icon in an external program to open a mail message within the viewer. Moreover, since it is not implemented as a file server, Hyperbole does not share the advantages of that architecture.

Henry s error program in 4BSD echoes a small but common use of plumbing. It takes the error messages produced by a compiler and drives a text editor through the steps of looking at each one in turn; the notion is to quicken the compile/edit/debug cycle. Similar results are achieved in EMACS by writing special M-LISP macros to parse the error messages from various compilers. Although for this particular purpose they may be more convenient than plumbing, these are specific solutions to a specific problem and lack plumbing's generality.

Of course, the resource forks in MacOS and the association rules for file name extensions in Windows also provide some of the functionality of the plumber, although again without the generality or dynamic nature.

Closer to home, Ousterhouts Tcl (Tool Command Language) [Oust90] was originally designed to embed a little command interpreter in each application to control interprocess communication and provide a level of integration. Plumbing, on the other hand, provides minimal support within the application, offloading most of the message handling and all the command execution to the central plumber.

The most obvious relative to plumbing is perhaps the hypertext links of a web browser. Plumbing differs by synthesizing the links on demand. Rather than constructing links within a document as in HTML, plumbing uses the context of a button click to derive what it should link to. That the rules for this decision can be modified dynamically gives it a more fluid feel than a standard web browsing world. One possibility for future work is to adapt a web browser to use plumbing as its link-following engine, much as Acme used plumbing to offload its acquisition rules. This would connect the web browser to the existing tools, rather than the current trend in most systems of replacing the tools by a browser.

Each of these prior systems and there are others, e.g. [Pasa93, Free93] addresses a particular need or subset of the issues of system integration. Plumbing differs because its particular choices were different. It focuses on two key issues: centralizing and automating the handling of interprocess communication among interactive programs, and maximizing the convenience (or minimizing the trouble) for the human user of its services. Moreover, the plumber s implementation as a file server, with messages passed over files it controls, permits the architecture to work transparently across a network. None of the other systems discussed here integrated distributed systems as smoothly as local ones without the addition of significant extra technology.

Discussion

There were a few surprises during the development of plumbing. The first version of plumbing was done for the Inferno system [Dorw97a, Dorw97b], using its file-to-channel mechanism to mediate the IPC. Although it was very simple to build, it encountered difficulties because the plumber was too disconnected from its clients; in particular, there was no way to discover whether a port was in use. When plumbing was implemented afresh for Plan 9, it was provided through a true file server. Although this was much more work, it paid off handsomely. The plumber now knows whether a port is open, which makes it easy to decide whether a new program must be started to handle a message, and the ability to edit the rules file dynamically is a major advantage. Other advantages arise from the file-server design, such as the ease of exporting plumbing ports across the network to remote machines and the implicit security model a file-based interface provides: no one has permission to open my private plumbing files.

On the other hand, Inferno was an all-new environment and the user interface for plumbing was able to be made uniform for all applications. This was impractical for Plan 9, so more *ad hoc* interfaces had to be provided for that environment. Yet even in Plan 9 the advantages of efficient, convenient, dynamic interprocess communication outweigh the variability of the user interface. In fact, it is perhaps a telling point that the system works well for a variety of interfaces; the provision of a central, convenient message-passing service is a good idea regardless of how the programs use it.

Plumbings rule language uses only regular expressions and a few special rules such as isfile for matching text. There is much more that could be done. For example, in the current system a JPEG file can be recognized by a .jpg suffix but not by its contents, since the plumbing language has no facility for examining the *contents* of files named in its messages. To address this issue without adding more special rules requires rethinking the language itself. Although the current system seems a good balance of complexity and functionality, perhaps a richer, more general-purpose language would permit more exotic applications of the plumbing model.

In conclusion, plumbing adds an effective, easy-to-use inter-application communication mechanism to the Plan 9 user interface. Its unusual design as a language-driven file server makes it easy to add context-dependent, dynamically interpreted, general-purpose hyperlinks to the desktop, for both existing tools and new ones.

Acknowledgements

Dave Presotto wrote the mail file system and edmail. He, Russ Cox, Sape Mullender, and Cliff Young influenced the design, offered useful suggestions, and suffered early versions of the software. They also made helpful comments on this paper, as did Dennis Ritchie and Brian Kernighan.

References

[Dorw97a] Sean Dorward, Rob Pike, David Leo Presotto, Dennis M. Ritchie, Howard W. Trickey, and Philip Winterbottom, Inferno, *Proceedings of the IEEE Compcon 97 Conference*, San Jose, 1997, pp. 241-244.

[Dorw97b] Sean Dorward, Rob Pike, David Leo Presotto, Dennis M. Ritchie, Howard W. Trickey, and Philip Winterbottom, The Inferno Operating System, *Bell Labs Technical Journal*, **2**, 1, Winter, 1997.

[Free93] FreeBSD, Syslog configuration file format, *syslog.conf* (0).

[Kill84] T. J. Killian, Processes as Files, *Proceedings of the Summer 1984 USENIX Conference*, Salt Lake City, 1984, pp. 203-207.

[Oust90] John K. Ousterhout, Tcl: An Embeddable Command Languages, *Proceedings of the Winter 1990 USENIX Conference*, Washington, 1990, pp. 133-146.

[Pasa93] Vern Paxson and Chris Saltmarsh, "Glish: A User-Level Software Bus for Loosely-Coupled Distributed Systems", *Proceedings of the Winter 1993 USENIX Conference*, San Diego, 1993, pp. 141-155.

[Pike87a] Rob Pike, Structural Regular Expressions, *EUUG Spring 1987 Conference Proceedings*, Helsinki, May 1987, pp. 21-28.

[Pike87b] Rob Pike, The Text Editor sam, *Software – Practice and Experience*, **17**, 5, Nov. 1987, pp. 813-845.

[Pike91] Rob Pike, 8½, the Plan 9 Window System, *Proceedings of the Summer 1991 USENIX Conference*, Nashville, 1991, pp. 257-265.

[Pike93] Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, and Phil Winterbottom, The Use of Name Spaces in Plan 9, *Operating Systems Review*, **27**, 2, April 1993, pp. 72-76.

[Pike94] Rob Pike, Acme: A User Interface for Programmers, *Proceedings of the Winter 1994 USENIX Conference*, San Francisco, 1994, pp. 223-234.

[PiPr85] Rob Pike and Dave Presotto, Face the Nation, *Proceedings of the USENIX Summer 1985 Conference*, Portland, 1985, pg. 81.

[Reis95] Steven P. Reiss, *The FIELD Programming Environment: A Friendly Integrated Environment for Learning and Development*, Kluwer, Boston, 1995.

[Wein] Bob Weiner, *Hyperbole User Manual*, http://www.cs.indiana.edu/elisp/hyperbole/hyperbole_1.html

[Wint94] Philip Winterbottom, ACID: A Debugger based on a Language, *Proceedings of* *the USENIX Winter Conference* , San Francisco, CA, 1994.

[WiGu92] Niklaus Wirth and Jurg Gutknecht, *Project Oberon: The Design of an Operating System and Compilers* , Addison-Wesley, Reading, 1992.

Hello World
or
°±»·¼–Á± °ÌÃ¼µ
or
こんにちは世界

Rob Pike
Ken Thompson

rob,ken@plan9.bell-labs.com

ABSTRACT

Plan 9 from Bell Labs has recently been converted from ASCII to an ASCII-compatible variant of the Unicode Standard, a 16-bit (now 21-bit) character set. In this paper we explain the reasons for the change, describe the character set and representation we chose, and present the programming models and software changes that support the new text format. Although we stopped short of full internationalization—for example, system error messages are in Unixese, not Japanese—we believe Plan 9 is the first system to treat the representation of all major languages on a uniform, equal footing throughout all its software.

Introduction

The world is multilingual but most computer systems are based on English and ASCII. The first release of Plan 9 [Pike90], a new distributed operating system from Bell Laboratories, seemed a good occasion to correct this chauvinism. It is easier to make such deep changes when building new systems than by refitting old ones.

The ANSI C standard [ANSIC] contains some guidance on the matter of wide and multi-byte characters but falls far short of solving the myriad associated problems. We could find no literature on how to convert a *system* to larger character sets, although some individual *programs* had been converted. This paper reports what we discovered as we explored the problem of representing multilingual text at all levels of an operating system, from the file system and kernel through the applications and up to the window system and display.

Plan 9 has not been internationalized: its manuals are in English, its error messages are in English, and it can display text that goes from left to right only. But before we can address these other problems, we need to handle, uniformly and comfortably, the textual representation of all the major written languages. That subproblem is richer than we had anticipated.

Originally appeared, in a slightly different form, in *Proc. of the Winter 1993 USENIX Conf.*, pp. 43-50, San Diego. It has been revised to reflect the move to 21-bit Unicode.

Standards

Our first step was to select a standard. At the time (January 1992), there were only two viable options: ISO 10646 [ISO10646] and Unicode [Unicode]. The documents describing both proposals were still in the draft stage.

The draft of ISO 10646 was not very attractive to us. It defined a sparse set of 32-bit characters, which would be hard to implement and have punitive storage requirements. Also, the draft attempted to mollify national interests by allocating 16-bit subspaces to national committees to partition individually. The suggested mode of use was to flip between separate national standards to implement the international standard. This did not strike us as a sound basis for a character set. As well, transmitting 32-bit values in a byte stream, such as in pipes, would be expensive and hard to implement. Since the standard does not define a byte order for such transmission, the byte stream would also have to carry state to enable the values to be recovered.

The Unicode Standard is a proposal by a consortium of mostly American computer companies formed to protest the technical failings of ISO 10646. It defines a uniform 16-bit code based on the principle of unification: two characters are the same if they look the same even though they are from different languages. This principle, called Han unification, allows the large Japanese, Chinese, and Korean character sets to be packed comfortably into a 16-bit representation.

We chose the Unicode Standard for its technical merits and because its code space was better defined. Moreover, the Unicode Consortium was derailing the ISO 10646 standard. (Now, in 1995, ISO 10646 is a standard with one 16-bit group defined, which is almost exactly the Unicode Standard. As most people expected, the two standards bodies reached a détente and ISO 10646 and Unicode represent the same character set.)

The Unicode Standard defines an adequate character set but an unreasonable representation. It states that all characters are 16 bits wide and are communicated and stored in 16-bit units. It also reserves a pair of characters (hexadecimal FFFE and FEFF) to detect byte order in transmitted text, requiring state in the byte stream. (The Unicode Consortium was thinking of files, not pipes.) To adopt this encoding, we would have had to convert all text going into and out of Plan 9 between ASCII and Unicode, which cannot be done. Within a single program, in command of all its input and output, it is possible to define characters as 16-bit quantities; in the context of a networked system with hundreds of applications on diverse machines by different manufacturers, it is impossible.

We needed a way to adapt the Unicode Standard to the tools-and-pipes model of text processing embodied by the Unix system. To do that, we needed an ASCII-compatible textual representation of Unicode characters for transmission and storage. In the draft ISO standard there was an informative (non-required) Annex called UTF that provided a byte stream encoding of the 32-bit ISO code. The encoding uses multibyte sequences composed from the 190 printable characters of Latin-1 to represent character values larger than 159.

The UTF encoding has several good properties. By far the most important is that a byte in the ASCII range 0-127 represents itself in UTF. Thus UTF is backward compatible with ASCII.

UTF has other advantages. It is a byte encoding and is therefore byte-order independent. ASCII control characters appear in the byte stream only as themselves, never as an element of a sequence encoding another character, so newline bytes separate lines of UTF text. Finally, ANSI C's strcmp function applied to UTF strings preserves the ordering of Unicode characters.

To encode and decode UTF is expensive (involving multiplication, division, and modulo operations) but workable. UTF's major disadvantage is that the encoding is not self-synchronizing. It is in general impossible to find the character boundaries in a UTF string without reading from the beginning of the string, although in practice control characters such as newlines, tabs, and blanks provide synchronization points.

In August 1992, X-Open circulated a proposal for another UTF-like byte encoding of Unicode characters. Their major concern was that an embedded character in a file name (in particular a slash) could be part of an escape sequence in UTF and therefore confuse a traditional file system. Their proposal would allow all 7-bit ASCII characters to represent themselves, *and only themselves*, in text. Multibyte sequences would contain only characters with the high bit set. We proposed a modification to the new UTF that would address our synchronization problem. Our proposal, which was originally known informally as UTF-2 and FSS-UTF, is now referred to as UTF-8 and has been approved by ISO to become Annex P to ISO 10646.

The model for text in Plan 9 is chosen from these three standards*: the Unicode character set encoded as a byte stream by UTF-8, from (soon to be) Annex P of ISO 10646. Although this mixture may seem like a precarious position for us to adopt, it is not as bad as it sounds. ISO 10646 and the Unicode Standard have converged, other systems such as Linux have adopted the same character set and encoding, and the general feeling seems to be that Unicode and UTF-8 will be accepted as the way to exchange text between systems. The prognosis for wide acceptance is good.

There are a couple of aspects of the Unicode Standard we have not faced. One is the issue of right-to-left text such as Hebrew or Arabic. Since that is an issue of display, not representation, we believe we can defer that problem for the moment without affecting our ability to solve it later. Another issue is diacriticals and combining characters, which cause overstriking of multiple Unicode characters. Although necessary for some scripts, such as Thai, Arabic, and Hebrew, such characters confuse the issues for Latin languages because they generate multiple representations for accented characters. ISO 10646 describes three levels of implementation; in Plan 9 we decided not to address the issue. Again, this can be labeled as a display issue and its finer points are still being debated, so we felt comfortable deferring. Mañana.

Although we converted Plan 9 in the altruistic interests of serving foreign languages, we have found the large character set attractive for other reasons. The Unicode Standard includes many characters mathematical symbols, scientific notation, more general punctuation, and more that we now use daily in our work. We no longer test our imaginations to find ways to include non-ASCII symbols in our text; why type :-) when you can use the character ☺? Most compelling is the ability to absorb documents and data that contain non-ASCII characters; our browser for the Oxford English Dictionary lets us see the dictionary as it really is, with pronunciation in the IPA font, foreign phrases properly rendered, and so on, *in plain text.*

As of Unicode 4.0, characters are now 21 bits wide and the longest UTF-8 encoding of a character requires 4 bytes. We are adapting the system to match.

In the rest of this paper, except when stated otherwise, the term UTF refers to the UTF-8 encoding of Unicode characters as adopted by Plan 9.

C Compiler

The first program to be converted to UTF was the C Compiler. There are two levels of conversion. On the syntactic level, input to the C compiler is UTF; on the semantic level, the C language needs to define how compiled programs manipulate the UTF set.

The syntactic part is simple. The ANSI C language standard defines the source character set to be ASCII. Since UTF is backward compatible with ASCII, the compiler needs little change. The only places where a larger character set is allowed are in

---

* Thats the nice thing about standards theres so many to choose from. Andy Tannenbaum (no, the other one)

character constants, strings, and comments. Since 7-bit ASCII characters can represent only themselves in UTF, the compiler does not have to be careful while looking for the termination of a string or comment.

The Plan 9 compiler extends ANSI C to treat any Unicode character with a value outside of the ASCII range as an alphabetic. To a Greek programmer or an English mathematician, ± is a sensible and now valid variable name.

On the semantic level, ANSI C allows, but does not tie down, the notion of a *wide character* and admits string and character constants of this type. We chose the wide character type to be `unsigned short` (now `unsigned long`). In the libraries, the word `Rune` is now defined by a `typedef` to be equivalent to `unsigned long` and is used to signify a Unicode character.

There are surprises; for example:

`L'x'`	is 120
`'x'`	is 120
`L'ÿ'`	is 255
`'ÿ'`	is -1, stdio `EOF` (if `char` is signed)
`L'±'`	is 945
`'±'`	is illegal

In the string constants,

```
"こんにちは世界"
L"こんにちは世界",
```

the former is an array of `chars` with 22 elements and a null byte, while the latter is an array of `unsigned longs` (`Runes`) with 8 elements and a null `Rune`.

The Plan 9 library provides an output conversion function, `print` (analogous to `printf`), with formats `%c`, `%C`, `%s`, and `%S`. Since `print` produces text, its output is always UTF. The character conversion `%c` (lower case) masks its argument to 8 bits before converting to UTF. Thus `L'ÿ'` and `'ÿ'` printed under `%c` will be identical, but `L'±'` will print as the Unicode character with decimal value 177. The character conversion `%C` (upper case) masks its argument to 16 bits before converting to UTF. Thus `L'ÿ'` and `L'±'` will print correctly under `%C`, but `'ÿ'` will not. The conversion `%s` (lower case) expects a pointer to `char` and copies UTF sequences up to a null byte. The conversion `%S` (upper case) expects a pointer to `Rune` and performs sequential `%C` conversions until a null `Rune` is encountered.

Another problem in format conversion is the definition of `%10s`: does the number refer to bytes or characters? We decided that such formats were most often used to align output columns and so made the number count characters. Some programs, however, use the count to place blank-padded strings in fixed-sized arrays. These programs must be found and corrected.

Here is a complete example:

```
#include <u.h>

char c[] = "こんにちは世界";
Rune s[] = L"こんにちは世界";

main(void)
{
        print("%d, %d\n", sizeof(c), sizeof(s));
        print("%s\n", c);
        print("%S\n", s);
}
```

This program prints 23, 18 and then two identical lines of UTF text. In practice, %S and L"..." are rare in programs; one reason is that most formatted I/O is done in unconverted UTF.

Ramifications

All programs in Plan 9 now read and write text as UTF, not ASCII. This change breaks two deep-rooted symmetries implicit in most C programs:

1. A character is no longer a char.
2. The internal representation (Rune) of a character now differs from its external representation (UTF).

In the sections that follow, we show how these issues were faced in the layers of system software from the operating system up to the applications. The effects are wide-reaching and often surprising.

Operating system

Since UTF is the only format for text in Plan 9, the interface to the operating system had to be converted to UTF. Text strings cross the interface in several places: command arguments, file names, user names (people can log in using their native name), error messages, and miscellaneous minor places such as commands to the I/O system. Little change was required: null-terminated UTF strings are equivalent to null-terminated ASCII strings for most purposes of the operating system. The library routines described in the next section made that change straightforward.

The window system, once called 8.5, is now rightfully called 8½.

Libraries

A header file included by all programs (see [Pike92]) declares the Rune type to hold 21-bit character values:

```
typedef unsigned long Rune;
```

Also defined are several constants relevant to UTF:

```
enum
{
    UTFmax      = 4,        /* maximum bytes per rune */
    Runesync    = 0x80,     /* cannot be in a UTF sequence (<) */
    Runeself    = 0x80,     /* rune==UTF sequence (<) */
    Runeerror   = 0xFFFD,       /* decoding error in UTF */
    Runemax     = 0x10FFFF,     /* largest 21-bit rune */
    Runemask    = 0x1FFFFF,     /* bits used by runes (see grep) */
};
```

(With the original UTF, Runesync was hexadecimal 21 and Runeself was A0.) UTFmax bytes are sufficient to hold the UTF encoding of any Unicode character. Characters of values less than Runesync only appear in a UTF string as themselves, never as part of a sequence encoding another character. Characters of value less than Runeself encode into single bytes of the same value. Finally, when the library detects errors in UTF input byte sequences that are not valid UTF sequences it converts the first byte of the error sequence to the character Runeerror. There is little a rune-oriented program can do when given bad data except exit, which is unreasonable, or carry on. Originally the conversion routines, described below, returned errors when given invalid UTF, but we found ourselves repeatedly checking for errors and ignoring them. We therefore decided to convert a bad sequence to a valid rune and continue processing. (The ANSI C routines, on the other hand, return errors.)

This technique does have the unfortunate property that converting invalid UTF byte strings in and out of runes does not preserve the input, but this circumstance only occurs when non-textual input is given to a textual program. The Unicode Standard defines an error character value FFFD, to stand for characters from other sets that it does not represent. The Runeerror character is a different concept, related to the encoding rather than the character set.

The Plan 9 C library contains a number of routines for manipulating runes. The first set converts between runes and UTF strings:

```
extern   int     runetochar(char*,   Rune*);
extern   int     chartorune(Rune*,   char*);
extern   int     runelen(long);
extern   int     fullrune(char*,   int);
```

Runetochar translates a single Rune to a UTF sequence and returns the number of bytes produced. Chartorune goes the other way, reporting how many bytes were consumed. Runelen returns the number of bytes in the UTF encoding of a rune. Fullrune examines a UTF string up to a specified number of bytes and reports whether the string begins with a complete UTF encoding. All these routines use the Runeerror character to work around encoding problems.

There is also a set of routines for examining null-terminated UTF strings, based on the model of the ANSI standard str routines, but with utf substituted for str and rune for chr:

```
extern   int     utflen(char*);
extern   char*   utfrune(char*,   long);
extern   char*   utfrrune(char*,   long);
extern   char*   utfutf(char*,   char*);
```

Utflen returns the number of runes in a UTF string; utfrune returns a pointer to the first occurrence of a rune in a UTF string, and utfrrune a pointer to the last. Utfutf searches for the first occurrence of a UTF string in another UTF string. Given the synchronizing property of UTF-8, utfutf is the same as strstr if the arguments point to valid UTF strings.

It is a mistake to use strchr or strrchr unless searching for a 7-bit ASCII character, that is, a character less than Runeself.

We have no routines for manipulating null-terminated arrays of Runes. Although they should probably exist for completeness, we have found no need for them, for the same reason that %S and L"..." are rarely used.

Most Plan 9 programs use a new buffered I/O library, BIO, in place of Standard I/O. BIO contains routines to read and write UTF streams, converting to and from runes. Bgetrune returns, as a Rune within a long, the next character in the UTF input stream; Bputrune takes a rune and writes its UTF representation. Bungetrune puts a rune back into the input stream for rereading.

Plan 9 programs use a simple set of macros to process command line arguments. Converting these macros to UTF automatically updated the argument processing of most programs. In general, argument flag names can no longer be held in a byte, and arrays of 256 bytes cannot be used to hold a set of flags.

We have done nothing analogous to ANSI C's locales, partly because we do not feel qualified to define locales and partly because we remain unconvinced of that model for dealing with the problems. That is really more an issue of internationalization than conversion to a larger character set; on the other hand, because we have chosen a single character set that encompasses most languages, some of the need for locales is eliminated. (We have a utility, tcs, that translates between UTF and other character sets.)

There are several reasons why our library does not follow the ANSI design for wide and multi-byte characters. The ANSI model was designed by a committee, untried, almost as an afterthought, whereas we wanted to design as we built. (We made several major changes to the interface as we became familiar with the problems involved.) We disagree with ANSI C s handling of invalid multi-byte sequences. Also, the ANSI C library is incomplete: although it contains some crucial routines for handling wide and multi-byte characters, there are some serious omissions. For example, our software can exploit the fact that UTF preserves ASCII characters in the byte stream. We could remove that assumption by replacing all calls to `strchr` with `utfrune` and so on. (Because of the weaker properties of the original UTF, we have actually done so.) ANSI C cannot: the standard says nothing about the representation, so portable code should *never* call `strchr`, yet there is no ANSI equivalent to `utfrune`. ANSI C simultaneously invalidates `strchr` and offers no replacement.

Finally, ANSI did nothing to integrate wide characters into the I/O system: it gives no method for printing wide characters. We therefore needed to invent some things and decided to invent everything. In the end, some of our entry points do correspond closely to ANSI routines for example `chartorune` and `runetochar` are similar to `mbtowc` and `wctomb` but Plan 9 s library defines more functionality, enough to write real applications comfortably.

Converting the tools

The source for our tools and applications had already been converted to work with Latin-1, so it was 8-bit safe, but the conversion to the Unicode Standard and UTF is more involved. Some programs needed no change at all: `cat`, for instance, interprets its argument strings, delivered in UTF, as file names that it passes uninterpreted to the `open` system call, and then just copies bytes from its input to its output; it never makes decisions based on the values of the bytes. (Plan 9 `cat` has no options such as `-v` to complicate matters.) Most programs, however, needed modest change.

It is difficult to find automatically the places that need attention, but `grep` helps. Software that uses the libraries conscientiously can be searched for calls to library routines that examine bytes as characters: `strchr`, `strrchr`, `strstr`, etc. Replacing these by calls to `utfrune`, `utfrrune`, and `utfutf` is enough to fix many programs. Few tools actually need to operate on runes internally; more typically they need only to look for the final slash in a file name and similar trivial tasks. Of the 170 C source programs in the top level of `/sys/src/cmd`, only 23 now contain the word `Rune`.

The programs that *do* store runes internally are mostly those whose *raison d'être* is character manipulation: `sam` (the text editor), `sed`, `sort`, `tr`, `troff`, 8½ (the window system and terminal emulator), and so on. To decide whether to compute using runes or UTF-encoded byte strings requires balancing the cost of converting the data when read and written against the cost of converting relevant text on demand. For programs such as editors that run a long time with a relatively constant dataset, runes are the better choice. There are space considerations too, but they are more complicated: plain ASCII text grows when converted to runes; UTF-encoded Japanese shrinks.

Again, it is hard to automate the conversion of a program from `char`s to `Rune`s. It is not enough just to change the type of variables; the assumption that bytes and characters are equivalent can be insidious. For instance, to clear a character array by

```
memset(buf, 0, BUFSIZE)
```

becomes wrong if `buf` is changed from an array of `char`s to an array of `Rune`s. Any program that indexes tables based on character values needs rethinking. Consider `tr`, which originally used multiple 256-byte arrays for the mapping. The naïve conversion would yield multiple 1,114,112-rune arrays. Instead Plan 9 `tr` saves space by building in effect a run-encoded version of the map.

Sort has related problems. The cooperation of UTF and strcmp means that a simple sort one with no options can be done on the original UTF strings using strcmp. With sorting options enabled, however, sort may need to convert its input to runes: for example, option -t± requires searching for alphas in the input text to crack the input into fields. The field specifier +3.2 refers to 2 runes beyond the third field. Some of the other options are hopelessly provincial: consider the case-folding and dictionary order options (Japanese doesn t even have an official dictionary order) or -M which compares by case-insensitive English month name. Handling these options involves the larger issues of internationalization and is beyond the scope of this paper and our expertise. Plan 9 sort works sensibly with options that make sense relative to the input. The simple and most important options are, however, usually meaningful. In particular, sort sorts UTF into the same order that look expects.

Regular expression-matching algorithms need rethinking to be applied to UTF text. Deterministic automata are usually applied to bytes; converting them to operate on variable-sized byte sequences is awkward. On the other hand, converting the input stream to runes adds measurable expense and the state tables expand from size 256 to 1,114,112; it can be expensive just to generate them. For simple string searching, the Boyer-Moore algorithm works with UTF provided the input is guaranteed to be only valid UTF strings; however, it does not work with the old UTF encoding. At a more mundane level, even character classes are harder: the usual bit-vector representation within a non-deterministic automaton is unwieldy with 1,114,112 characters in the alphabet.

We compromised. An existing library for compiling and executing regular expressions was adapted to work on runes, with two entry points for searching in arrays of runes and arrays of chars (the pattern is always UTF text). Character classes are represented internally as runs of runes; the reserved value FFFF marks the end of the class. Then *all* utilities that use regular expressions editors, grep, awk, etc. except the shell, whose notation was grandfathered, were converted to use the library. For some programs, there was a concomitant loss of performance, but there was also a strong advantage. To our knowledge, Plan 9 is the only Unix-like system that has a single definition and implementation of regular expressions; patterns are written and interpreted identically by all the programs in the system.

A handful of programs have the notion of character built into them so strongly as to confuse the issue of what they should do with UTF input. Such programs were treated as individual special cases. For example, wc is, by default, unchanged in behavior and output; a new option, -r, counts the number of correctly encoded runes valid UTF sequences in its input; -b, the number of invalid sequences.

It took us several months to convert all the software in the system to the Unicode Standard and the old UTF. When we decided to convert from that to the new UTF, only three things needed to be done. First, we rewrote the library routines to encode and decode the new UTF. This took an evening. Next, we converted all the files containing UTF to the new encoding. We wrote a trivial program to look for non-ASCII bytes in text files and used a Plan 9 program called tcs (translate character set) to change encodings. Finally, we recompiled all the system software; the library interface was unchanged, so recompilation was sufficient to effect the transformation. The second two steps were done concurrently and took an afternoon. We concluded that the actual encoding is relatively unimportant to the software; the adoption of large characters and a byte-stream encoding *per se* are much deeper issues.

Graphics and fonts

Plan 9 provides only minimal support for plain text terminals. It is instead designed to be used with all character input and output mediated by a window system such as 8½. The window system and related software are responsible for the display of UTF text as Unicode character images. For plain text, the window system must provide a user-settable *font* that provides a (possibly empty) picture for each Unicode character. Fancier applications that use bold and Italic characters need multiple fonts storing multiple pictures for each Unicode value. All the issues are apparent, though, in just the problem of displaying a single image for each character, that is, the Unicode equivalent of a plain text terminal. With 128 or even 256 characters, a font can be just an array of bitmaps. With 1,114,112 characters, a more sophisticated design is necessary. To store the ideographs for just Japanese as 16×16×1 bit images, the smallest they can reasonably be, takes over a quarter of a megabyte. Make the images a little larger, store more bits per pixel, and hold a copy in every running application, and the memory cost becomes unreasonable.

The structure of the bitmap graphics services is described at length elsewhere [Pike91]. In summary, the memory holding the bitmaps is stored in the same machine that has the display, mouse, and keyboard: the terminal in Plan 9 terminology, the workstation in others. Access to that memory and associated services is provided by device files served by system software on the terminal. One of those files, /dev/bitblt , interprets messages written upon it as requests for actions corresponding to entry points in the graphics library: allocate a bitmap, execute a raster operation, draw a text string, etc. The window system acts as a multiplexer that mediates access to the services and resources of the terminal by simulating in each client window a set of files mirroring those provided by the system. That is, each window has a distinct /dev/mouse , /dev/bitblt , and so on through which applications drive graphical input and output.

One of the resources managed by 8½ and the terminal is the set of active *subfonts.* Each subfont holds the bitmaps and associated data structures for a sequential set of Unicode characters. Subfonts are stored in files and loaded into the terminal by 8½ or an application. For example, one subfont might hold the images of the first 256 characters of the Unicode space, corresponding to the Latin-1 character set; another might hold the standard phonetic character set, Unicode characters with value 0250 to 02E9. These files are collected in directories corresponding to typefaces: /lib/font/bit/pelm contains the Pellucida Monospace character set, with subfonts holding the Latin-1, Greek, Cyrillic and other components of the typeface. A suffix on subfont files encodes (in a subfont-specific way) the size of the images: /lib/font/bit/pelm/latin1.9 contains the Latin-1 Pellucida Monospace characters with lower case letters 9 pixels high; /lib/font/bit/jis/jis5400.16 contains 16-pixel high ideographs starting at Unicode value 5400.

The subfonts do not identify which portion of the Unicode space they cover. Instead, a font file, in plain text, describes how to assemble subfonts into a complete character set. The font file is presented as an argument to the window system to determine how plain text is displayed in text windows and applications. Here is the beginning of the font file /lib/font/bit/pelm/jis.9.font , which describes the layout of a font covering that portion of the Unicode Standard for which we have characters of typical display size, using Japanese characters to cover the Han space:

```
18      14
0x0000  0x00FF  latin1.9
0x0100  0x017E  latineur.9
0x0250  0x02E9  ipa.9
0x0386  0x03F5  greek.9
0x0400  0x0475  cyrillic.9
0x2000  0x2044  ../misc/genpunc.9
0x2070  0x208E  supsub.9
0x20A0  0x20AA  currency.9
0x2100  0x2138  ../misc/letterlike.9
0x2190  0x21EA  ../misc/arrows
0x2200  0x227F  ../misc/math1
0x2280  0x22F1  ../misc/math2
0x2300  0x232C  ../misc/tech
0x2500  0x257F  ../misc/chart
0x2600  0x266F  ../misc/ding

0x3000  0x303f  ../jis/jis3000.16
0x30a1  0x30fe  ../jis/katakana.16
0x3041  0x309e  ../jis/hiragana.16
0x4e00  0x4fff  ../jis/jis4e00.16
0x5000  0x51ff  ../jis/jis5000.16
...
```

The first two numbers set the interline spacing of the font (18 pixels) and the distance from the baseline to the top of the line (14 pixels). When characters are displayed, they are placed so as best to fit within those constraints; characters too large to fit will be truncated. The rest of the file associates subfont files with portions of Unicode space. The first four such files are in the Pellucida Monospace typeface and directory; others reside in other directories. The file names are relative to the font file's own location.

There are several advantages to this two-level structure. First, it simultaneously breaks the huge Unicode space into manageable components and provides a unifying architecture for assembling fonts from disjoint pieces. Second, the structure promotes sharing. For example, we have only one set of Japanese characters but dozens of typefaces for the Latin-1 characters, and this structure permits us to store only one copy of the Japanese set but use it with any Roman typeface. Also, customization is easy. English-speaking users who don't need Japanese characters but may want to read an on-line Oxford English Dictionary can assemble a custom font with the Latin-1 (or even just ASCII) characters and the International Phonetic Alphabet (IPA). Moreover, to do so requires just editing a plain text file, not using a special font editing tool. Finally, the structure guides the design of caching protocols to improve performance and memory usage.

To load a complete Unicode character set into each application would consume too much memory and, particularly on slow terminal lines, would take unreasonably long. Instead, Plan 9 assembles a multi-level cache structure for each font. An application opens a font file, reads and parses it, and allocates a data structure. A message written to /dev/bitblt allocates an associated structure held in the terminal, in particular, a bitmap to act as a cache for recently used character images. Other messages copy these images to bitmaps such as the screen by loading characters from subfonts into the cache on demand and from there to the destination bitmap. The protocol to draw characters is in terms of cache indices, not Unicode character number or UTF sequences. These details are hidden from the application, which instead sees only a subroutine to draw a string in a bitmap from a given font, functions to discover character size information, and routines to allocate and to free fonts.

As needed, whole subfonts are opened by the graphics library, read, and then downloaded to the terminal. They are held open by the library in an LRU-replacement list. Even when the program closes a subfont, it is retained in the terminal for later use. When the application opens the subfont, it asks the terminal if it already has a copy to avoid reading it from the file server if possible. This level of cache has the property that the bitmaps for, say, all the Japanese characters are stored only once, in the terminal; the applications read only size and width information from the terminal and share the images.

The sizes of the character and subfont caches held by the application are adaptive. A simple algorithm monitors the cache miss rate to enlarge and shrink the caches as required. The size of the character cache is limited to 2048 images maximum, which in practice seems enough even for Japanese text. For plain ASCII-like text it naturally stays around 128 images.

This mechanism sounds complicated but is implemented by only about 500 lines in the library and considerably less in each of the terminals graphics driver and 8½. It has the advantage that only characters that are being used are loaded into memory. It is also efficient: if the characters being drawn are in the cache the extra overhead is negligible. It works particularly well for alphabetic character sets, but also adapts on demand for ideographic sets. When a user first looks at Japanese text, it takes a few seconds to read all the font data, but thereafter the text is drawn almost as fast as regular text (the images are larger, so draw a little slower). Also, because the bitmaps are remembered by the terminal, if a second application then looks at Japanese text it starts faster than the first.

We considered building a font server to cache character images and associated data for the applications, the window system, and the terminal. We rejected this design because, although isolating many of the problems of font management into a separate program, it didnt simplify the applications. Moreover, in a distributed system such as Plan 9 it is easy to have too many special purpose servers. Making the management of the fonts the concern of only the essential components simplifies the system and makes bootstrapping less intricate.

Input

A completely different problem is how to type Unicode characters as input to the system. We selected an unused key on our ASCII keyboards to serve as a prefix for multi-keystroke sequences that generate Unicode characters. For example, the character ü is generated by the prefix key (typically ALT or Compose) followed by a double quote and a lower-case u. When that character is read by the application, from the file /dev/cons, it is of course presented as its UTF encoding. Such sequences generate characters from an arbitrary set that includes all of Latin-1 plus a selection of mathematical and technical characters. An arbitrary Unicode character may be generated by typing the prefix, an upper case X, and four hexadecimal digits that identify the Unicode value.

These simple mechanisms are adequate for most of our day-to-day needs: its easy to remember to type ALT 1 2 for ½ or ALT accent letter for accented Latin letters. For the occasional unusual character, the cut and paste features of 8½ serve well. A program called (perhaps misleadingly) unicode takes as argument a hexadecimal value, and prints the UTF representation of that character, which may then be picked up with the mouse and used as input.

These methods are clearly unsatisfactory when working in a non-English language. In the native country of such a language the appropriate keyboard is likely to be at hand. But its also reasonable especially now that the system handles Unicode characters to work in a language foreign to the keyboard.

For alphabetic languages such as Greek or Russian, it is straightforward to construct a program that does phonetic substitution, so that, for example, typing a Latin a yields the Greek ± . Within Plan 9, such a program can be inserted transparently between the real keyboard and a program such as the window system, providing a manageable input device for such languages.

For ideographic languages such as Chinese or Japanese the problem is harder. Native users of such languages have adopted methods for dealing with Latin keyboards that involve a hybrid technique based on phonetics to generate a list of possible symbols followed by menu selection to choose the desired one. Such methods can be effective, but their design must be rooted in information about the language unknown to non-native speakers. (Cxterm , a Chinese terminal emulator built by and for Chinese programmers, employs such a technique [Pong and Zhang].) Although the technical problem of implementing such a device is easy in Plan 9 it is just an elaboration of the technique for alphabetic languages our lack of familiarity with such languages has restrained our enthusiasm for building one.

The input problem is technically the least interesting but perhaps emotionally the most important of the problems of converting a system to an international character set. Beyond that remain the deeper problems of internationalization such as multi-lingual error messages and command names, problems we are not qualified to solve. With the ability to treat text of most languages on an equal footing, though, we can begin down that path. Perhaps people in non-English speaking countries will consider adopting Plan 9, solving the input problem locally perhaps just by plugging in their local terminals and begin to use a system with at least the capacity to be international.

Acknowledgements

Dennis Ritchie provided consultation and encouragement. Bob Flandrena converted most of the standard tools to UTF. Brian Kernighan suffered cheerfully with several inadequate implementations and converted troff to UTF. Rich Drechsler converted his Postscript driver to UTF. John Hobby built the Postscript ⊕. We thank them all.

References

[ANSIC] *American National Standard for Information Systems Programming Language C*, American National Standards Institute, Inc., New York, 1990.

[ISO10646] ISO/IEC DIS 10646-1:1993 *Information technology Universal Multiple-Octet Coded Character Set (UCS) Part 1: Architecture and Basic Multilingual Plane* .

[Pike90] R. Pike, D. Presotto, K. Thompson, H. Trickey, Plan 9 from Bell Labs , UKUUG Proc. of the Summer 1990 Conf., London, England, 1990.

[Pike91] R. Pike, 8½, The Plan 9 Window System , USENIX Summer Conf. Proc., Nashville, 1991, reprinted in this volume.

[Pike92] R. Pike, How to Use the Plan 9 C Compiler, this volume.

[Pong and Zhang] Man-Chi Pong and Yongguang Zhang, cxterm: A Chinese Terminal Emulator for the X Window System , *Software Practice and Experience*, Vol 22(1), 809-926, October 1992.

[Unicode] *The Unicode Standard, Worldwide Character Encoding, Version 1.0, Volume 1*, The Unicode Consortium, Addison Wesley, New York, 1991.

Plan 9 C Compilers

Ken Thompson
ken@plan9.bell-labs.com

ABSTRACT

This paper describes the overall structure and function of the Plan 9 C compilers. A more detailed implementation document for any one of the compilers is yet to be written.

1. Introduction

There are many compilers in the series. Six of the compilers (Intel 386, AMD64, PowerPC, PowerPC 64-bit, ARM, MIPS R3000) are considered active and are used to compile current versions of Plan 9. One of the compilers (SPARC) is maintained but is for older machines for which we have no current ports of Plan 9; we are unlikely to port to any SPARC machines. The DEC Alpha and Motorola 68020 compilers have been retired. Several others (Motorola 68000, Intel 960, AMD 29000) have had only limited use, such as to program peripherals or experimental devices. Any of the disused compilers could be restored if needed.

2. Structure

The compiler is a single program that produces an object file. Combined in the compiler are the traditional roles of preprocessor, lexical analyzer, parser, code generator, local optimizer, and first half of the assembler. The object files are binary forms of assembly language, similar to what might be passed between the first and second passes of an assembler.

Object files and libraries are combined by a loader program to produce the executable binary. The loader combines the roles of second half of the assembler, global optimizer, and loader. The names of the compilers, loaders, and assemblers are as follows:

SPARC	kc	kl	ka
PowerPC	qc	ql	qa
MIPS	vc	vl	va
MIPS little-endian	0c	0l	0a
ARM	5c	5l	5a
AMD64	6c	6l	6a
Intel 386	8c	8l	8a
PowerPC 64-bit	9c	9l	9a

There is a further breakdown in the source of the compilers into object-independent and object-dependent parts. All of the object-independent parts are combined into source files in the directory /sys/src/cmd/cc. The object-dependent parts are collected in a separate directory for each compiler, for example /sys/src/cmd/vc. All of the code, both object-independent and object-dependent, is machine-independent

Originally appeared, in a different form, in *Proceedings of the Summer 1990 UKUUG Conference*, pp. 41-51, London, 1990.

and may be cross-compiled and executed on any of the architectures.

3. The Language

The compiler implements ANSI C with some restrictions and extensions [ANSI90]. Most of the restrictions are due to personal preference, while most of the extensions were to help in the implementation of Plan 9. There are other departures from the standard, particularly in the libraries, that are beyond the scope of this paper.

3.1. Register, volatile, const

The keyword `register` is recognized syntactically but is semantically ignored. Thus taking the address of a `register` variable is not diagnosed. The keyword `volatile` disables all optimizations, in particular registerization, of the corresponding variable. The keyword `const` generates warnings (if warnings are enabled by the compiler's `-w` option) of non-constant use of the variable, but does not affect the generated code.

3.2. The preprocessor

The C preprocessor is probably the biggest departure from the ANSI standard.

The preprocessor built into the Plan 9 compilers does not support `#if`, although it does handle `#ifdef` and `#include`. If it is necessary to be more standard, the source text can first be run through the separate ANSI C preprocessor, `cpp`.

3.3. Unnamed substructures

The most important and most heavily used of the extensions is the declaration of an unnamed substructure or subunion. For example:

```
typedef
struct      lock
{
      int      locked;
} Lock;

typedef
struct      node
{
      int   type;
      union
      {
            double dval;
            float    fval;
            long     lval;
      };
      Lock;
} Node;

Lock*       lock;
Node*       node;
```

The declaration of `Node` has an unnamed substructure of type `Lock` and an unnamed subunion. One use of this feature allows references to elements of the subunit to be accessed as if they were in the outer structure. Thus `node->dval` and `node->locked` are legitimate references.

When an outer structure is used in a context that is only legal for an unnamed substructure, the compiler promotes the reference to the unnamed substructure. This is true for references to structures and to references to pointers to structures. This happens in assignment statements and in argument passing where prototypes have been declared. Thus, continuing with the example,

```
lock = node;
```

would assign a pointer to the unnamed `Lock` in the `Node` to the variable `lock`. Another example,

```
extern void lock(Lock*);
func(...)
{
        ...
        lock(node);
        ...
}
```

will pass a pointer to the `Lock` substructure.

Finally, in places where context is insufficient to identify the unnamed structure, the type name (it must be a `typedef`) of the unnamed structure can be used as an identifier. In our example, `&node->Lock` gives the address of the anonymous `Lock` structure.

3.4. Structure displays

A structure cast followed by a list of expressions in braces is an expression with the type of the structure and elements assigned from the corresponding list. Structures are now almost first-class citizens of the language. It is common to see code like this:

```
r = (Rectangle){point1, (Point){x,y+2}};
```

3.5. Initialization indexes

In initializers of arrays, one may place a constant expression in square brackets before an initializer. This causes the next initializer to assign the indicated element. For example:

```
enum errors
{
        Etoobig,
        Ealarm,
        Egreg
};
char* errstrings[] =
{
        [Ealarm]    "Alarm call",
        [Egreg]     "Panic: out of mbufs",
        [Etoobig] "Arg list too long",
};
```

In the same way, individual structure members may be initialized in any order by preceding the initialization with `.tagname`. Both forms allow an optional `=`, to be compatible with a proposed extension to ANSI C.

3.6. External register

The declaration `extern register` will dedicate a register to a variable on a global basis. It can be used only under special circumstances. External register variables must be identically declared in all modules and libraries. The feature is not intended for efficiency, although it can produce efficient code; rather it represents a unique storage class that would be hard to get any other way. On a shared-memory multi-processor, an external register is one-per-processor and neither one-per-procedure (automatic) or one-per-system (external). It is used for two variables in the Plan 9 kernel, u and m. U is a pointer to the structure representing the currently running process and m is a pointer to the per-machine data structure.

3.7. Long long

The compilers accept long long as a basic type meaning 64-bit integer. On some of the machines this type is synthesized from 32-bit instructions.

3.8. Pragma

The compilers accept #pragma lib *libname* and pass the library name string uninterpreted to the loader. The loader uses the library name to find libraries to load. If the name contains $O, it is replaced with the single character object type of the compiler (e.g., v for the MIPS). If the name contains $M, it is replaced with the architecture type for the compiler (e.g., mips for the MIPS). If the name starts with / it is an absolute pathname; if it starts with . then it is searched for in the loader's current directory. Otherwise, the name is searched from /$M/lib. Such #pragma statements in header files guarantee that the correct libraries are always linked with a program without the need to specify them explicitly at link time.

They also accept #pragma packed on (or yes or 1) to cause subsequently declared data, until #pragma packed off (or no or 0), to be laid out in memory tightly packed in successive bytes, disregarding the usual alignment rules. Accessing such data can cause faults.

Similarly, #pragma profile off (or no or 0) causes subsequently declared functions, until #pragma profile on (or yes or 1), to be marked as unprofiled. Such functions will not be profiled when profiling is enabled for the rest of the program.

Two #pragma statements allow type-checking of print-like functions. The first, of the form

```
#pragma varargck argpos error 2
```

tells the compiler that the second argument to error is a print format string (see the manual page *print*(2)) that specifies how to format error's subsequent arguments. The second, of the form

```
#pragma varargck type "s" char*
```

says that the print format verb s processes an argument of type char*. If the compiler's -F option is enabled, the compiler will use this information to report type violations in the arguments to print, error, and similar routines.

4. Object module conventions

The overall conventions of the runtime environment are important to runtime efficiency. In this section, several of these conventions are discussed.

4.1. Register saving

In the Plan 9 compilers, the caller of a procedure saves the registers. With caller-saves, the leaf procedures can use all the registers and never save them. If you spend a lot of time at the leaves, this seems preferable. With callee-saves, the saving of the registers is done in the single point of entry and return. If you are interested in space, this seems preferable. In both, there is a degree of uncertainty about what registers need to be saved. Callee-saved registers make it difficult to find variables in registers in debuggers. Callee-saved registers also complicate the implementation of longjmp. The convincing argument is that with caller-saves, the decision to registerize a variable can include the cost of saving the registers across calls. For a further discussion of caller- vs. callee-saves, see the paper by Davidson and Whalley [Dav91].

In the Plan 9 operating system, calls to the kernel look like normal procedure calls, which means the caller has saved the registers and the system entry does not have to. This makes system calls considerably faster. Since this is a potential security hole, and can lead to non-determinism, the system may eventually save the registers on entry, or more likely clear the registers on return.

4.2. Calling convention

Older C compilers maintain a frame pointer, which is at a known constant offset from the stack pointer within each function. For machines where the stack grows towards zero, the argument pointer is at a known constant offset from the frame pointer. Since the stack grows down in Plan 9, the Plan 9 compilers keep neither an explicit frame pointer nor an explicit argument pointer; instead they generate addresses relative to the stack pointer.

On some architectures, the first argument to a subroutine is passed in a register.

4.3. Functions returning structures

Structures longer than one word are awkward to implement since they do not fit in registers and must be passed around in memory. Functions that return structures are particularly clumsy. The Plan 9 compilers pass the return address of a structure as the first argument of a function that has a structure return value. Thus

```
x = f(...)
```

is rewritten as

```
f(&x, ...)   .
```

This saves a copy and makes the compilation much less clumsy. A disadvantage is that if you call this function without an assignment, a dummy location must be invented.

There is also a danger of calling a function that returns a structure without declaring it as such. With ANSI C function prototypes, this error need never occur.

5. Implementation

The compiler is divided internally into four machine-independent passes, four machine-dependent passes, and an output pass. The next nine sections describe each pass in order.

5.1. Parsing

The first pass is a YACC-based parser [Joh79]. Declarations are interpreted immediately, building a block structured symbol table. Executable statements are put into a parse tree and collected, without interpretation. At the end of each procedure, the parse tree for the function is examined by the other passes of the compiler.

The input stream of the parser is a pushdown list of input activations. The preprocessor expansions of macros and `#include` are implemented as pushdowns. Thus there is no separate pass for preprocessing.

5.2. Typing

The next pass distributes typing information to every node of the tree. Implicit operations on the tree are added, such as type promotions and taking the address of arrays and functions.

5.3. Machine-independent optimization

The next pass performs optimizations and transformations of the tree, such as converting &*x and *&x into x. Constant expressions are converted to constants in this pass.

5.4. Arithmetic rewrites

This is another machine-independent optimization. Subtrees of add, subtract, and multiply of integers are rewritten for easier compilation. The major transformation is factoring: 4+8*a+16*b+5 is transformed into 9+8*(a+2*b). Such expressions arise from address manipulation and array indexing.

5.5. Addressability

This is the first of the machine-dependent passes. The addressability of a processor is defined as the set of expressions that is legal in the address field of a machine language instruction. The addressability of different processors varies widely. At one end of the spectrum are the 68020 and VAX, which allow a complex mix of incrementing, decrementing, indexing, and relative addressing. At the other end is the MIPS, which allows only registers and constant offsets from the contents of a register. The addressability can be different for different instructions within the same processor.

It is important to the code generator to know when a subtree represents an address of a particular type. This is done with a bottom-up walk of the tree. In this pass, the leaves are labeled with small integers. When an internal node is encountered, it is labeled by consulting a table indexed by the labels on the left and right subtrees. For example, on the 68020 processor, it is possible to address an offset from a named location. In C, this is represented by the expression *(&name+constant). This is marked addressable by the following table. In the table, a node represented by the left column is marked with a small integer from the right column. Marks of the form A_i are addressable while marks of the form N_i are not addressable.

Node	**Marked**	
name	A_1	
const	A_2	
$\&A_1$	A_3	
A_3+A_1	N_1	(note that this is not addressable)
$*N_1$	A_4	

Here there is a distinction between a node marked A_1 and a node marked A_4 because the address operator of an A_4 node is not addressable. So to extend the table:

Node	**Marked**
$\&A_4$	N_2
N_2+N_1	N_1

The full addressability of the 68020 is expressed in 18 rules like this, while the addressability of the MIPS is expressed in 11 rules. When one ports the compiler, this table is usually initialized so that leaves are labeled as addressable and nothing else. The code produced is poor, but porting is easy. The table can be extended later.

This pass also rewrites some complex operators into procedure calls. Examples include 64-bit multiply and divide.

In the same bottom-up pass of the tree, the nodes are labeled with a Sethi-Ullman complexity [Set70]. This number is roughly the number of registers required to compile the tree on an ideal machine. An addressable node is marked 0. A function call is marked infinite. A unary operator is marked as the maximum of 1 and the mark of its subtree. A binary operator with equal marks on its subtrees is marked with a subtree mark plus 1. A binary operator with unequal marks on its subtrees is marked with the maximum mark of its subtrees. The actual values of the marks are not too important, but the relative values are. The goal is to compile the harder (larger mark) subtree first.

5.6. Code generation

Code is generated by recursive descent. The Sethi-Ullman complexity completely guides the order. The addressability defines the leaves. The only difficult part is compiling a tree that has two infinite (function call) subtrees. In this case, one subtree is compiled into the return register (usually the most convenient place for a function call) and then stored on the stack. The other subtree is compiled into the return register and then the operation is compiled with operands from the stack and the return register.

There is a separate boolean code generator that compiles conditional expressions. This is fundamentally different from compiling an arithmetic expression. The result of the boolean code generator is the position of the program counter and not an expression. The boolean code generator makes extensive use of De Morgan's rule. The boolean code generator is an expanded version of that described in chapter 8 of Aho, Sethi, and Ullman [Aho87].

There is a considerable amount of talk in the literature about automating this part of a compiler with a machine description. Since this code generator is so small (less than 500 lines of C) and easy, it hardly seems worth the effort.

5.7. Registerization

Up to now, the compiler has operated on syntax trees that are roughly equivalent to the original source language. The previous pass has produced machine language in an internal format. The next two passes operate on the internal machine language structures. The purpose of the next pass is to reintroduce registers for heavily used variables.

All of the variables that can be potentially registerized within a procedure are placed in a table. (Suitable variables are any automatic or external scalars that do not have their addresses extracted. Some constants that are hard to reference are also considered for registerization.) Four separate data flow equations are evaluated over the procedure on all of these variables. Two of the equations are the normal set-behind and used-ahead bits that define the life of a variable. The two new bits tell if a variable life crosses a function call ahead or behind. By examining a variable over its lifetime, it is possible to get a cost for registerizing. Loops are detected and the costs are multiplied by three for every level of loop nesting. Costs are sorted and the variables are replaced by available registers on a greedy basis.

The 68020 has two different types of registers. For the 68020, two different costs are calculated for each variable life and the register type that affords the better cost is used. Ties are broken by counting the number of available registers of each type.

Note that externals are registerized together with automatics. This is done by evaluating the semantics of a call instruction differently for externals and automatics. Since a call goes outside the local procedure, it is assumed that a call references all externals. Similarly, externals are assumed to be set before an entry instruction and assumed to be referenced after a return instruction. This makes sure that externals are in memory across calls.

The overall results are satisfactory. It would be nice to be able to do this processing in a machine-independent way, but it is impossible to get all of the costs and side effects of different choices by examining the parse tree.

Most of the code in the registerization pass is machine-independent. The major machine-dependency is in examining a machine instruction to ask if it sets or references a variable.

5.8. Machine code optimization

The next pass walks the machine code for opportunistic optimizations. For the most part, this is highly specific to a particular processor. One optimization that is performed on all of the processors is the removal of unnecessary move instructions. Ironically, most of these instructions were inserted by the previous pass. There are two patterns that are repetitively matched and replaced until no more matches are found. The first tries to remove move instructions by relabeling variables.

When a move instruction is encountered, if the destination variable is set before the source variable is referenced, then all of the references to the destination variable can be renamed to the source and the move can be deleted. This transformation uses the reverse data flow set up in the previous pass.

An example of this pattern is depicted in the following table. The pattern is in the left column and the replacement action is in the right column.

```
MOVE a->b                   (remove)
(sequence with no mention of a)
USE   b                     USE   a
(sequence with no mention of a)
SET   b                     SET   b
```

Experiments have shown that it is marginally worthwhile to rename uses of the destination variable with uses of the source variable up to the first use of the source variable.

The second transform will do relabeling without deleting instructions. When a move instruction is encountered, if the source variable has been set prior to the use of the destination variable then all of the references to the source variable are replaced by the destination and the move is inverted. Typically, this transformation will alter two move instructions and allow the first transformation another chance to remove code. This transformation uses the forward data flow set up in the previous pass.

Again, the following is a depiction of the transformation where the pattern is in the left column and the rewrite is in the right column.

```
SET   a                     SET   b
(sequence with no use of b)
USE   a                     USE   b
(sequence with no use of b)
MOVE a->b                   MOVE b->a
```

Iterating these transformations will usually get rid of all redundant move instructions.

A problem with this organization is that the cost of registerization calculated in the previous pass must depend on how well this pass can detect and remove redundant instructions. Often, a fine candidate for registerization is rejected because of the cost of instructions that are later removed.

5.9. Writing the object file

The last pass walks the internal assembly language and writes the object file. The object file is reduced in size by about a factor of three with simple compression techniques. The most important aspect of the object file format is that it is independent of the compiling machine. All integer and floating numbers in the object code are converted to known formats and byte orders.

6. The loader

The loader is a multiple pass program that reads object files and libraries and produces an executable binary. The loader also does some minimal optimization and code rewriting. Many of the operations performed by the loader are machine-dependent.

The first pass of the loader reads the object modules into an internal data structure that looks like binary assembly language. As the instructions are read, code is reordered to remove unconditional branch instructions. Conditional branch instructions are inverted to prevent the insertion of unconditional branches. The loader will also make a copy of a few instructions to remove an unconditional branch.

The next pass allocates addresses for all external data. Typical of processors is the MIPS, which can reference ±32K bytes from a register. The loader allocates the register R30 as the static pointer. The value placed in R30 is the base of the data segment plus 32K. It is then cheap to reference all data in the first 64K of the data segment. External variables are allocated to the data segment with the smallest variables allocated first. If all of the data cannot fit into the first 64K of the data segment, then usually only a few large arrays need more expensive addressing modes.

For the MIPS processor, the loader makes a pass over the internal structures, exchanging instructions to try to fill delay slots with useful work. If a useful instruction cannot be found to fill a delay slot, the loader will insert noop instructions. This pass is very expensive and does not do a good job. About 40% of all instructions are in delay slots. About 65% of these are useful instructions and 35% are noops. The vendor-supplied assembler does this job more effectively, filling about 80% of the delay slots with useful instructions.

On the 68020 processor, branch instructions come in a variety of sizes depending on the relative distance of the branch. Thus the size of branch instructions can be mutually dependent. The loader uses a multiple pass algorithm to resolve the branch lengths [Szy78]. Initially, all branches are assumed minimal length. On each subsequent pass, the branches are reassessed and expanded if necessary. When no more expansions occur, the locations of the instructions in the text segment are known.

On the MIPS processor, all instructions are one size. A single pass over the instructions will determine the locations of all addresses in the text segment.

The last pass of the loader produces the executable binary. A symbol table and other tables are produced to help the debugger to interpret the binary symbolically.

The loader places absolute source line numbers in the symbol table. The name and absolute line number of all #include files is also placed in the symbol table so that the debuggers can associate object code to source files.

7. Performance

The following is a table of the source size of the MIPS compiler.

lines	module
509	machine-independent headers
1070	machine-independent YACC source
6090	machine-independent C source
545	machine-dependent headers
6532	machine-dependent C source
298	loader headers
5215	loader C source

The following table shows timing of a test program that plays checkers, running on a

MIPS R4000. The test program is 26 files totaling 12600 lines of C. The execution time does not significantly depend on library implementation. Since no other compiler runs on Plan 9, the Plan 9 tests were done with the Plan 9 operating system; the other tests were done on the vendors operating system. The hardware was identical in both cases. The optimizer in the vendor compiler is reputed to be extremely good.

```
  4.49s   Plan 9 vc -N compile time (opposite of -O)
  1.72s   Plan 9 vc -N load time
148.69s   Plan 9 vc -N run time

 15.07s   Plan 9 vc compile time (-O implicit)
  1.66s   Plan 9 vc load time
 89.96s   Plan 9 vc run time

 14.83s   vendor cc compile time
  0.38s   vendor cc load time
104.75s   vendor cc run time

 43.59s   vendor cc -O compile time
  0.38s   vendor cc -O load time
 76.19s   vendor cc -O run time

  8.19s   vendor cc -O3 compile time
 35.97s   vendor cc -O3 load time
 71.16s   vendor cc -O3 run time
```

To compare the Intel compiler, a program that is about 40% bit manipulation and about 60% single precision floating point was run on the same 33 MHz 486, once under Windows compiled with the Watcom compiler, version 10.0, in 16-bit mode and once under Plan 9 in 32-bit mode. The Plan 9 execution time was 27 sec while the Windows execution time was 31 sec.

8. Conclusions

The new compilers compile quickly, load slowly, and produce medium quality object code. The compilers are relatively portable, requiring but a couple of weeks work to produce a compiler for a different computer. For Plan 9, where we needed several compilers with specialized features and our own object formats, this project was indispensable. It is also necessary for us to be able to freely distribute our compilers with the Plan 9 distribution.

Two problems have come up in retrospect. The first has to do with the division of labor between compiler and loader. Plan 9 runs on multi-processors and as such compilations are often done in parallel. Unfortunately, all compilations must be complete before loading can begin. The load is then single-threaded. With this model, any shift of work from compile to load results in a significant increase in real time. The same is true of libraries that are compiled infrequently and loaded often. In the future, we may try to put some of the loader work back into the compiler.

The second problem comes from the various optimizations performed over several passes. Often optimizations in different passes depend on each other. Iterating the passes could compromise efficiency, or even loop. We see no real solution to this problem.

9. References

[Aho87] A. V. Aho, R. Sethi, and J.D. Ullman, *Compilers Principles, Techniques, and Tools*, Addison Wesley, Reading, MA, 1987.

[ANSI90] *American National Standard for Information Systems Programming Language C*, American National Standards Institute, Inc., New York, 1990.

[Dav91] J.W. Davidson and D. B. Whalley, Methods for Saving and Restoring Register Values across Function Calls, *Software Practice and Experience*, Vol 21(2), pp. 149-165, February 1991.

[Joh79] S. C. Johnson, YACC Yet Another Compiler Compiler, *UNIX Programmer's Manual, Seventh Ed., Vol. 2A*, AT&T Bell Laboratories, Murray Hill, NJ, 1979.

[Set70] R. Sethi and J.D. Ullman, The Generation of Optimal Code for Arithmetic Expressions, *Journal of the ACM*, Vol 17(4), pp. 715-728, 1970.

[Szy78] T. G. Szymanski, Assembling Code for Machines with Span-dependent Instructions, *Communications of the ACM*, Vol 21(4), pp. 300-308, 1978.

Adding Application Support for a New Architecture in Plan 9

Bob Flandrena
bobf@plan9.bell-labs.com

Introduction

Plan 9 has five classes of architecture-dependent software: headers, kernels, compilers and loaders, the libc system library, and a few application programs. In general, architecture-dependent programs consist of a portable part shared by all architectures and a processor-specific portion for each supported architecture. The portable code is often compiled and stored in a library associated with each architecture. A program is built by compiling the architecture-specific code and loading it with the library. Support for a new architecture is provided by building a compiler for the architecture, using it to compile the portable code into libraries, writing the architecture-specific code, and then loading that code with the libraries.

This document describes the organization of the architecture-dependent code and headers on Plan 9. The first section briefly discusses the layout of the headers and the source code for the kernels, compilers, loaders, and the system library, libc. The second section provides a detailed discussion of the structure of libmach, a library containing almost all architecture-dependent code used by application programs. The final section describes the steps required to add application program support for a new architecture.

Directory Structure

Architecture-dependent information for the new processor is stored in the directory tree rooted at /*m* where *m* is the name of the new architecture (e.g., mips). The new directory should be initialized with several important subdirectories, notably bin, include, and lib. The directory tree of an existing architecture serves as a good model for the new tree. The architecture-dependent mkfile must be stored in the newly created root directory for the architecture. It is easiest to copy the mkfile for an existing architecture and modify it for the new architecture. When the mkfile is correct, change the OS and CPUS variables in the /sys/src/mkfile.proto to reflect the addition of the new architecture.

Headers

Architecture-dependent headers are stored in directory /*m*/include where *m* is the name of the architecture (e.g., mips). Two header files are required: u.h and ureg.h. The first defines fundamental data types, bit settings for the floating point status and control registers, and va_list processing which depends on the stack model for the architecture. This file is best built by copying and modifying the u.h file from an architecture with a similar stack model. The ureg.h file contains a structure describing the layout of the saved register set for the architecture; it is defined by the kernel.

Header file /sys/include/a.out.h contains the definitions of the magic numbers used to identify executables for each architecture. When support for a new architecture is added, the magic number for the architecture must be added to this file.

The header format of a bootable executable is defined by each manufacturer. Header file /sys/include/bootexec.h contains structures describing the headers currently supported. If the new architecture uses a common header such as COFF, the header format is probably already defined, but if the bootable header format is non-standard, a structure defining the format must be added to this file.

Kernel

Although the kernel depends critically on the properties of the underlying hardware, most of the higher-level kernel functions, including process management, paging, pseudo-devices, and some networking code, are independent of processor architecture. The portable kernel code is divided into two parts: that implementing kernel functions and that devoted to the boot process. Code in the first class is stored in directory /sys/src/9/port and the portable boot code is stored in /sys/src/9/boot. Architecture-dependent kernel code is stored in the subdirectories of /sys/src/9 named for each architecture.

The relationship between the kernel code and the boot code is convoluted and subtle. The portable boot code is compiled into a library for each architecture. An architecture-specific main program is loaded with the appropriate library and the resulting executable is compiled into the kernel where it is executed as a user process during the final stages of kernel initialization. The boot process performs authentication, attaches the name space root to the appropriate file system and starts the init process.

The organization of the portable kernel source code differs from that of most other architecture-specific code. Instead of storing the portable code in a library and loading it with the architecture-specific code, the portable code is compiled directly into the directory containing the architecture-specific code and linked with the object files built from the source in that directory.

Compilers and Loaders

The compiler source code conforms to the usual organization: portable code is compiled into a library for each architecture and the architecture-dependent code is loaded with that library. The common compiler code is stored in /sys/src/cmd/cc. The mkfile in this directory compiles the portable source and archives the objects in a library for each architecture. The architecture-specific compiler source is stored in a subdirectory of /sys/src/cmd with the same name as the compiler (e.g., /sys/src/cmd/vc).

There is no portable code shared by the loaders. Each directory of loader source code is self-contained, except for a header file and an instruction name table included from the directory of the associated compiler.

Libraries

Most C library modules are portable; the source code is stored in directories /sys/src/libc/port and /sys/src/libc/9sys. Architecture-dependent library code is stored in the subdirectory of /sys/src/libc named the same as the target processor. Non-portable functions not only implement architecture-dependent operations but also supply assembly language implementations of functions where speed is critical. Directory /sys/src/libc/9syscall is unusual because it contains architecture-dependent information for all architectures. It holds only a header file defining the names and numbers of system calls and a mkfile. The mkfile executes an rc script that parses the header file, constructs assembler language functions implementing the system call for each architecture, assembles the code, and archives the object files in libc. The assembler language syntax and the system interface differ for each architecture. The rc script in this mkfile must be modified to support a new architecture.

Applications

Application programs process two forms of architecture-dependent information: executable images and intermediate object files. Almost all processing is on executable files. System library libmach provides functions that convert architecture-specific data to a portable format so application programs can process this data independent of its underlying representation. Further, when a new architecture is implemented almost all code changes are confined to the library; most affected application programs need only be reloaded. The source code for the library is /sys/src/libmach.

An application program running on one type of processor must be able to interpret architecture-dependent information for all supported processors. For example, a debugger must be able to debug the executables of all architectures, not just the architecture on which it is executing, since /proc may be imported from a different machine.

A small part of the application library provides functions to extract symbol references from object files. The remainder provides the following processing of executable files or memory images:

- Header interpretation.
- Symbol table interpretation.
- Execution context interpretation, such as stack traces and stack frame location.
- Instruction interpretation including disassembly and instruction size and follow-set calculations.
- Exception and floating point number interpretation.
- Architecture-independent read and write access through a relocation map.

Header file /sys/include/mach.h defines the interfaces to the application library. Manual pages *mach* (2), *symbol* (2), and *object* (2) describe the details of the library functions.

Two data structures, called Mach and Machdata, contain architecture-dependent parameters and a jump table of functions. Global variables mach and machdata point to the Mach and Machdata data structures associated with the target architecture. An application determines the target architecture of a file or executable image, sets the global pointers to the data structures associated with that architecture, and subsequently performs all references indirectly through the pointers. As a result, direct references to the tables for each architecture are avoided and the application code intrinsically supports all architectures (though only one at a time).

Object file processing is handled similarly: architecture-dependent functions identify and decode the intermediate files for the processor. The application indirectly invokes a classification function to identify the architecture of the object code and to select the appropriate decoding function. Subsequent calls then use that function to decode each record. Again, the layer of indirection allows the application code to support all architectures without modification.

Splitting the architecture-dependent information between the Mach and Machdata data structures allows applications to choose an appropriate level of service. Even though an application does not directly reference the architecture-specific data structures, it must load the architecture-dependent tables and code for all architectures it supports. The size of this data can be substantial and many applications do not require the full range of architecture-dependent functionality. For example, the size command does not require the disassemblers for every architecture; it only needs to decode the header. The Mach data structure contains a few architecture-specific parameters and a description of the processor register set. The size of the structure varies with the size of the register set but is generally small. The Machdata data structure contains a jump table of architecture-dependent functions; the amount of code and data referenced by this table is usually large.

Libmach Source Code Organization

The libmach library provides four classes of functionality:

Header and Symbol Table Decoding - Files executable.c and sym.c contain code to interpret the header and symbol tables of an executable file or executing image. Function crackhdr decodes the header, reformats the information into an Fhdr data structure, and points global variable mach to the Mach data structure of the target architecture. The symbol table processing uses the data in the Fhdr structure to decode the symbol table. A variety of symbol table access functions then support queries on the reformatted table.

Debugger Support - Files named *m*.c, where *m* is the code letter assigned to the architecture, contain the initialized Mach data structure and the definition of the register set for each architecture. Architecture-specific debugger support functions and an initialized Machdata structure are stored in files named *m*db.c. Files machdata.c and setmach.c contain debugger support functions shared by multiple architectures.

Architecture-Independent Access - Files map.c, access.c, and swap.c provide accesses through a relocation map to data in an executable file or executing image. Byte-swapping is performed as needed. Global variables mach and machdata must point to the Mach and Machdata data structures of the target architecture.

Object File Interpretation - These files contain functions to identify the target architecture of an intermediate object file and extract references to symbols. File obj.c contains code common to all architectures; file *m*obj.c contains the architecture-specific source code for the machine with code character *m*.

The Machdata data structure is primarily a jump table of architecture-dependent debugger support functions. Functions select the Machdata structure for a target architecture based on the value of the type code in the Fhdr structure or the name of the architecture. The jump table provides functions to swap bytes, interpret machine instructions, perform stack traces, find stack frames, format floating point numbers, and decode machine exceptions. Some functions, such as machine exception decoding, are idiosyncratic and must be supplied for each architecture. Others depend on the compiler run-time model and several architectures may share code common to a model. For example, many architectures share the code to process the fixed-frame stack model implemented by several of the compilers. Finally, some functions, such as byte-swapping, provide a general capability and the jump table need only select an implementation appropriate to the architecture.

Adding Application Support for a New Architecture

This section describes the steps required to add application-level support for a new architecture. We assume the kernel, compilers, loaders and system libraries for the new architecture are already in place. This implies that a code-character has been assigned and that the architecture-specific headers have been updated. With the exception of two programs, application-level changes are confined to header files and the source code in /sys/src/libmach.

1. Begin by updating the application library header file in /sys/include/mach.h. Add the following symbolic codes to the enum statement near the beginning of the file:

   The processor type code, e.g., MSPARC.

   The type of the executable. There are usually two codes needed: one for a bootable executable (i.e., a kernel) and one for an application executable.

   The disassembler type code. Add one entry for each supported disassembler for the architecture.

   A symbolic code for the object file.

2. In a file name /sys/src/libmach/*m*.c (where *m* is the identifier character assigned to the architecture), initialize Reglist and Mach data structures with values defining the register set and various system parameters. The source file for a similar architecture can serve as template. Most of the fields of the Mach data structure are obvious but a few require further explanation.

   kbase – This field contains the address of the kernel ublock. The debuggers assume the first entry of the kernel ublock points to the Proc structure for a kernel thread.

   ktmask – This field is a bit mask used to calculate the kernel text address from the kernel ublock address. The first page of the kernel text segment is calculated by ANDing the negation of this mask with kbase.

   kspoff – This field contains the byte offset in the Proc data structure to the saved kernel stack pointer for a suspended kernel thread. This is the offset to the sched.sp field of a Proc table entry.

   kpcoff – This field contains the byte offset into the Proc data structure of the program counter of a suspended kernel thread. This is the offset to field sched.pc in that structure.

   kspdelta and kpcdelta – These fields contain corrections to be added to the stack pointer and program counter, respectively, to properly locate the stack and next instruction of a kernel thread. These values bias the saved registers retrieved from the Label structure named sched in the Proc data structure. Most architectures require no bias and these fields contain zeros.

   scalloff – This field contains the byte offset of the scallnr field in the ublock data structure associated with a process. The scallnr field contains the number of the last system call executed by the process. The location of the field varies depending on the size of the floating point register set which precedes it in the ublock.

3. Add an entry to the initialization of the ExecTable data structure at the beginning of file /sys/src/libmach/executable.c. Most architectures require two entries: one for a normal executable and one for a bootable image. Each table entry contains:

   Magic Number The big-endian magic number assigned to the architecture in /sys/include/a.out.h.

   Name A string describing the executable.

   Executable type code The executable code assigned in /sys/include/mach.h.

   Mach pointer The address of the initialized Mach data structure constructed in Step 2. You must also add the name of this table to the list of Mach table definitions immediately preceding the ExecTable initialization.

   Header size The number of bytes in the executable file header. The size of a normal executable header is always sizeof(Exec). The size of a bootable header is determined by the size of the structure for the architecture defined in /sys/include/bootexec.h.

Byte-swapping function The address of beswal or leswal for big-endian and little-endian architectures, respectively.

Decoder function – The address of a function to decode the header. Function adotout decodes the common header shared by all normal (i.e., non-bootable) executable files. The header format of bootable executable files is defined by the manufacturer and a custom function is almost always required to decode it. Header file /sys/include/bootexec.h contains data structures defining the bootable headers for all architectures. If the new architecture uses an existing format, the appropriate decoding function should already be in executable.c. If the header format is unique, then a new function must be added to this file. Usually the decoding function for an existing architecture can be adopted with minor modifications.

4. Write an object file parser and store it in file /sys/src/libmach/*m*obj.c where *m* is the identifier character assigned to the architecture. Two functions are required: a predicate to identify an object file for the architecture and a function to extract symbol references from the object code. The object code format is obscure but it is often possible to adopt the code of an existing architecture with minor modifications. When these functions are in hand, insert their addresses in the jump table at the beginning of file /sys/src/libmach/obj.c.

5. Implement the required debugger support functions and initialize the parameters and jump table of the Machdata data structure for the architecture. This code is conventionally stored in a file named /sys/src/libmach/*m*db.c where *m* is the identifier character assigned to the architecture. The fields of the Machdata structure are:

bpinst and bpsize – These fields contain the breakpoint instruction and the size of the instruction, respectively.

swab – This field contains the address of a function to byte-swap a 16-bit value. Choose leswab or beswab for little-endian or big-endian architectures, respectively.

swal – This field contains the address of a function to byte-swap a 32-bit value. Choose leswal or beswal for little-endian or big-endian architectures, respectively.

ctrace – This field contains the address of a function to perform a C-language stack trace. Two general trace functions, risctrace and cisctrace, traverse fixed-frame and relative-frame stacks, respectively. If the compiler for the new architecture conforms to one of these models, select the appropriate function. If the stack model is unique, supply a custom stack trace function.

findframe – This field contains the address of a function to locate the stack frame associated with a text address. Generic functions riscframe and ciscframe process fixed-frame and relative-frame stack models.

ufixup – This field contains the address of a function to adjust the base address of the register save area. Currently, only the 68020 requires this bias to offset over the active exception frame.

excep – This field contains the address of a function to produce a text string describing the current exception. Each architecture stores exception information uniquely, so this code must always be supplied.

bpfix – This field contains the address of a function to adjust an address prior to laying down a breakpoint.

sftos – This field contains the address of a function to convert a single precision floating point value to a string. Choose leieeesftos for little-endian or beieeesftos for big-endian architectures.

dftos – This field contains the address of a function to convert a double precision floating point value to a string. Choose leieeedftos for little-endian or beieeedftos for big-endian architectures.

foll, das, hexinst, and instsize – These fields point to functions that interpret machine instructions. They rely on disassembly of the instruction and are unique to each architecture. Foll calculates the follow set of an instruction. Das disassembles a machine instruction to assembly language. Hexinst formats a machine instruction as a text string of hexadecimal digits. Instsize calculates the size in bytes, of an instruction. Once the disassembler is written, the other functions can usually be implemented as trivial extensions of it.

It is possible to provide support for a new architecture incrementally by filling the jump table entries of the Machdata structure as code is written. In general, if a jump table entry contains a zero, application programs requiring that function will issue an error message instead of attempting to call the function. For example, the foll, das, hexinst, and instsize jump table slots can be zeroed until a disassembler is written. Other capabilities, such as stack trace or variable inspection, can be supplied and will be available to the debuggers but attempts to use the disassembler will result in an error message.

6. Update the table named machines near the beginning of /sys/src/libmach/setmach.c. This table binds the file type code and machine name to the Mach and Machdata structures of an architecture. The names of the initialized Mach and Machdata structures built in steps 2 and 5 must be added to the list of structure definitions immediately preceding the table initialization. If both Plan 9 and native disassembly are supported, add an entry for each disassembler to the table. The entry for the default disassembler (usually Plan 9) must be first.
7. Add an entry describing the architecture to the table named trans near the end of /sys/src/cmd/prof.c.
8. Add an entry describing the architecture to the table named objtype near the start of /sys/src/cmd/pcc.c.
9. Recompile and install all application programs that include header file mach.h and load with libmach.a.

Lexical File Names in Plan 9
or
Getting Dot-Dot Right

Rob Pike
rob@plan9.bell-labs.com

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Symbolic links make the Unix file system non-hierarchical, resulting in multiple valid path names for a given file. This ambiguity is a source of confusion, especially since some shells work overtime to present a consistent view from programs such as pwd, while other programs and the kernel itself do nothing about the problem.

Plan 9 has no symbolic links but it does have other mechanisms that produce the same difficulty. Moreover, Plan 9 is founded on the ability to control a program s environment by manipulating its name space. Ambiguous names muddle the result of operations such as copying a name space across the network.

To address these problems, the Plan 9 kernel has been modified to maintain an accurate path name for every active file (open file, working directory, mount table entry) in the system. The definition of accurate is that the path name for a file is guaranteed to be the rooted, absolute name the program used to acquire it. These names are maintained by an efficient method that combines lexical processing such as evaluating . by just removing the last path name element of a directory with local operations within the file system to maintain a consistently, easily understood view of the name system. Ambiguous situations are resolved by examining the lexically maintained names themselves.

A new kernel call, fd2path, returns the file name associated with an open file, permitting the use of reliable names to improve system services ranging from pwd to debugging. Although this work was done in Plan 9, Unix systems could also benefit from the addition of a method to recover the accurate name of an open file or the current directory.

Motivation

Consider the following unedited transcript of a session running the Bourne shell on a modern Unix system:

```
% echo $HOME
/home/rob
% cd $HOME
% pwd
/n/bopp/v7/rob
% cd /home/rob
% cd /home/ken
% cd ../rob
../rob: bad directory
%
```

(The same output results from running tcsh; we'll discuss ksh in a moment.) To a neophyte being schooled in the delights of a hierarchical file name space, this behavior must be baffling. It is, of course, the consequence of a series of symbolic links intended to give users the illusion they share a disk, when in fact their files are scattered over several devices:

```
% ls -ld /home/rob /home/ken
lrwxr-xr-x  1 root  sys  14 Dec 26  1998 /home/ken  -> /n/bopp/v6/ken
lrwxr-xr-x  1 root  sys  14 Dec 23  1998 /home/rob  -> /n/bopp/v7/rob
%
```

The introduction of symbolic links has changed the Unix file system from a true hierarchy into a directed graph, rendering .. ambiguous and sowing confusion.

Unix popularized hierarchical naming, but the introduction of symbolic links made its naming irregular. Worse, the pwd command, through the underlying getwd library routine, uses a tricky, expensive algorithm that often delivers the wrong answer. Starting from the current directory, getwd opens the parent, .., and searches it for an entry whose i-number matches the current directory; the matching entry is the final path element of the ultimate result. Applying this process iteratively, getwd works back towards the root. Since getwd knows nothing about symbolic links, it will recover surprising names for directories reached by them, as illustrated by the example; the backward paths getwd traverses will not backtrack across the links.

Partly for efficiency and partly to make cd and pwd more predictable, the Korn shell ksh [Korn94] implements pwd as a builtin. (The cd command must be a builtin in any shell, since the current directory is unique to each process.) Ksh maintains its own private view of the file system to try to disguise symbolic links; in particular, cd and pwd involve some lexical processing (somewhat like the cleanname function discussed later in this paper), augmented by heuristics such as examining the environment for names like $HOME and $PWD to assist initialization of the state of the private view. [Korn00]

This transcript begins with a Bourne shell running:

```
% cd /home/rob
% pwd
/n/bopp/v7/rob
% ksh
$ pwd
/home/rob
$
```

This result is encouraging. Another example, again starting from a Bourne shell:

```
% cd /home/rob
% cd ../ken
../ken: bad directory
% ksh
$ pwd
/home/rob
$ cd ../ken
$ pwd
/home/ken
$
```

By doing extra work, the Korn shell is providing more sensible behavior, but it is easy to defeat:

```
% cd /home/rob
% pwd
/n/bopp/v7/rob
% cd bin
% pwd
/n/bopp/v7/rob/bin
% ksh
$ pwd
/n/bopp/v7/rob/bin
$ exit
% cd /home/ken
% pwd
/n/bopp/v6/ken
% ksh
$ pwd
/n/bopp/v6/ken
$
```

In these examples, ksh s built-in pwd failed to produce the results (/home/rob/bin and /home/ken) that the previous example might have led us to expect. The Korn shell is hiding the problem, not solving it, and in fact is not even hiding it very well.

A deeper question is whether the shell should even be trying to make pwd and cd do a better job. If it does, then the getwd library call and every program that uses it will behave differently from the shell, a situation that is sure to confuse. Moreover, the ability to change directory to ../ken with the Korn shell s cd command but not with the chdir system call is a symptom of a diseased system, not a healthy shell.

The operating system should provide names that work and make sense. Symbolic links, though, are here to stay, so we need a way to provide sensible, unambiguous names in the face of a non-hierarchical name space. This paper shows how the challenge was met on Plan 9, an operating system with Unix-like naming.

Names in Plan 9

Except for some details involved with bootstrapping, file names in Plan 9 have the same syntax as in Unix. Plan 9 has no symbolic links, but its name space construction operators, bind and mount , make it possible to build the same sort of non-hierarchical structures created by symbolically linking directories on Unix.

Plan 9 s mount system call takes a file descriptor and attaches to the local name space the file system service it represents:

```
mount(fd, "/dir", flags)
```

Here fd is a file descriptor to a communications port such as a pipe or network connection; at the other end of the port is a service, such as file server, that talks 9P, the Plan 9 file system protocol. After the call succeeds, the root directory of the service will be visible at the *mount point* /dir, much as with the mount call of Unix. The flag argument specifies the nature of the attachment: MREPL says that the contents of the root directory (appear to) replace the current contents of /dir; MAFTER says that the current contents of dir remain visible, with the mounted directory's contents appearing *after* any existing files; and MBEFORE says that the contents remain visible, with the mounted directory's contents appearing *before* any existing files. These multicomponent directories are called *union directories* and are somewhat different from union directories in 4.4BSD-Lite [PeMc95], because only the top-level directory itself is unioned, not its descendents, recursively. (Plan 9's union directories are used differently from 4.4BSD-Lite's, as will become apparent.)

For example, to bootstrap a diskless computer the system builds a local name space containing only the root directory, /, then uses the network to open a connection to the main file server. It then executes

```
mount(rootfd, "/", MREPL);
```

After this call, the entire file server's tree is visible, starting from the root of the local machine.

While mount connects a new service to the local name space, bind rearranges the existing name space:

```
bind("tofile", "fromfile", flags)
```

causes subsequent mention of the fromfile (which may be a plain file or a directory) to behave as though tofile had been mentioned instead, somewhat like a symbolic link. (Note, however, that the arguments are in the opposite order compared to ln -s). The flags argument is the same as with mount.

As an example, a sequence something like the following is done at bootstrap time to assemble, under the single directory /bin, all of the binaries suitable for this architecture, represented by (say) the string sparc:

```
bind("/sparc/bin", "/bin", MREPL);
bind("/usr/rob/sparc/bin", "/bin", MAFTER);
```

This sequence of binds causes /bin to contain first the standard binaries, then the contents of rob's private SPARC binaries. The ability to build such union directories obviates the need for a shell $PATH variable while providing opportunities for managing heterogeneity. If the system were a Power PC, the same sequence would be run with power textually substituted for sparc to place the Power PC binaries in /bin rather than the SPARC binaries.

Trouble is already brewing. After these bindings are set up, where does

```
% cd /bin
% cd ..
```

set the current working directory, to / or /sparc or /usr/rob/sparc? We will return to this issue.

There are some important differences between binds and symbolic links. First, symbolic links are a static part of the file system, while Plan 9 bindings are created at run time, are stored in the kernel, and endure only as long as the system maintains them; they are temporary. Since they are known to the kernel but not the file system, they must be set up each time the kernel boots or a user logs in; permanent bindings are created by editing system initialization scripts and user profiles rather than by building them in the file system itself.

The Plan 9 kernel records what bindings are active for a process, whereas symbolic links, being held on the Unix file server, may strike whenever the process evaluates a file name. Also, symbolic links apply to all processes that evaluate the affected file, whereas bind has a local scope, applying only to the process that executes it and possibly some of its peers, as discussed in the next section. Symbolic links cannot construct the sort of /bin directory built here; it is possible to have multiple directories point to /bin but not the other way around.

Finally, symbolic links are symbolic, like macros: they evaluate the associated names each time they are accessed. Bindings, on the other hand, are evaluated only once, when the bind is executed; after the binding is set up, the kernel associates the underlying files, rather than their names. In fact, the kernel's representation of a bind is identical to its representation of a mount; in effect, a bind is a mount of the tofile upon the fromfile. The binds and mounts coexist in a single *mount table*, the subject of the next section.

The Mount Table

Unix has a single global mount table for all processes in the system, but Plan 9's mount tables are local to each process. By default it is inherited when a process forks, so mounts and binds made by one process affect the other, but a process may instead inherit a copy, so modifications it makes will be invisible to other processes. The convention is that related processes, such as processes running in a single window, share a mount table, while sets of processes in different windows have distinct mount tables. In practice, the name spaces of the two windows will appear largely the same, but the possibility for different processes to see different files (hence services) under the same name is fundamental to the system, affecting the design of key programs such as the window system [Pike91].

The Plan 9 mount table is little more than an ordered list of pairs, mapping the fromfiles to the tofiles. For mounts, the tofile will be an item called a Channel, similar to a Unix vnode, pointing to the root of the file service, while for a bind it will be the Channel pointing to the tofile mentioned in the bind call. In both cases, the fromfile entry in the table will be a Channel pointing to the fromfile itself.

The evaluation of a file name proceeds as follows. If the name begins with a slash, start with the Channel for the root; otherwise start with the Channel for the current directory of the process. For each path element in the name, such as usr in /usr/rob, try to walk the Channel to that element [Pike93]. If the walk succeeds, look to see if the resulting Channel is the same as any fromfile in the mount table, and if so, replace it by the corresponding tofile. Advance to the next element and continue.

There are a couple of nuances. If the directory being walked is a union directory, the walk is attempted in the elements of the union, in order, until a walk succeeds. If none succeed, the operation fails. Also, when the destination of a walk is a directory for a purpose such as the chdir system call or the fromfile in a bind, once the final walk of the sequence has completed the operation stops; the final check through the mount table is not done. Among other things, this simplifies the management of union directories; for example, subsequent bind calls will append to the union associated with the underlying fromfile instead of what is bound upon it.

A Definition of Dot-Dot

The ability to construct union directories and other intricate naming structures introduces some thorny problems: as with symbolic links, the name space is no longer hierarchical, files and directories can have multiple names, and the meaning of .., the parent directory, can be ambiguous.

The meaning of .. is straightforward if the directory is in a locally hierarchical part of the name space, but if we ask what .. should identify when the current directory is a mount point or union directory or multiply symlinked spot (which we will henceforth call just a mount point, for brevity), there is no obvious answer. Name spaces have been part of Plan 9 from the beginning, but the definition of `..` has changed several times as we grappled with this issue. In fact, several attempts to clarify the meaning of `..` by clever coding resulted in definitions that could charitably be summarized as what the implementation gives.

Frustrated by this situation, and eager to have better-defined names for some of the applications described later in this paper, we recently proposed the following definition for `..`:

> The parent of a directory *X*, *X*`/..`, is the same directory that would obtain if we instead accessed the directory named by stripping away the last path name element of *X*.

For example, if we are in the directory `/a/b/c` and `chdir` to `..`, the result is *exactly* as if we had executed `chdir` to `/a/b`.

This definition is easy to understand and seems natural. It is, however, a purely *lexical* definition that flatly ignores evaluated file names, mount tables, and other kernel-resident data structures. Our challenge is to implement it efficiently. One obvious (and correct) implementation is to rewrite path names lexically to fold out `..`, and then evaluate the file name forward from the root, but this is expensive and unappealing. We want to be able to use local operations to evaluate file names, but maintain the global, lexical definition of dot-dot. It isn't too hard.

The Implementation

To operate lexically on file names, we associate a name with each open file in the kernel, that is, with each `Channel` data structure. The first step is therefore to store a `char*` with each `Channel` in the system, called its `Cname`, that records the *absolute* rooted file name for the `Channel`. `Cnames` are stored as full text strings, shared copy-on-write for efficiency. The task is to maintain each `Cname` as an accurate absolute name using only local operations.

When a file is opened, the file name argument in the `open` (or `chdir` or `bind` or ...) call is recorded in the `Cname` of the resulting `Channel`. When the file name begins with a slash, the name is stored as is, subject to a cleanup pass described in the next section. Otherwise, it is a local name, and the file name must be made absolute by prefixing it with the `Cname` of the current directory, followed by a slash. For example, if we are in `/home/rob` and `chdir` to `bin`, the `Cname` of the resulting `Channel` will be the string `/home/rob/bin`.

This assumes, of course, that the local file name contains no `..` elements. If it does, instead of storing for example `/home/rob/..` we delete the last element of the existing name and set the `Cname` to `/home`. To maintain the lexical naming property we must guarantee that the resulting `Cname`, if it were to be evaluated, would yield the identical directory to the one we actually do get by the local operation.

If the current directory is not a mount point, it is easy to maintain the lexical property. If it is a mount point, though, it is still possible to maintain it on Plan 9 because the mount table, a kernel-resident data structure, contains all the information about the non-hierarchical connectivity of the name space. (On Unix, by contrast, symbolic links are stored on the file server rather than in the kernel.) Moreover, the presence of a full file name for each `Channel` in the mount table provides the information necessary to resolve ambiguities.

The mount table is examined in the *from*→*to* direction when evaluating a name, but `..` points backwards in the hierarchy, so to evaluate `..` the table must be examined in the *to*→*from* direction. (How did we get here?)

The value of `..` is ambiguous when there are multiple bindings (mount points) that point to the directories involved in the evaluation of ... For example, return to our original script with /n/bopp/v6 (containing a home directory for ken) and /n/bopp/v7 (containing a home directory for rob) unioned into /home. This is represented by two entries in the mount table, from=/home, to=/n/bopp/v6 and from=/home, to=/n/bopp/v7. If we have set our current directory to /home/rob (which has landed us in the physical location /n/bopp/v7/rob) our current directory is not a mount point but its parent is. The value of .. is ambiguous: it could be /home, /n/bopp/v7, or maybe even /n/bopp/v6, and the ambiguity is caused by two tofiles bound to the same fromfile. By our definition, if we now evaluate .., we should acquire the directory /home; otherwise ../ken could not possibly result in ken's home directory, which it should. On the other hand, if we had originally gone to /n/bopp/v7/rob, the name ../ken should *not* evaluate to ken's home directory because there is no directory /n/bopp/v7/ken (ken's home directory is on v6). The problem is that by using local file operations, it is impossible to distinguish these cases: regardless of whether we got here using the name /home/rob or /n/bopp/v7/rob, the resulting directory is the same. Moreover, the mount table does not itself have enough information to disambiguate: when we do a local operation to evaluate .. and land in /n/bopp/v7, we discover that the directory is a tofile in the mount table; should we step back through the /home mount table entry or not?

The solution comes from the Cnames themselves. Whether to step back through the mount point from=/home, to=/n/bopp/v7 when evaluating .. in rob's directory is trivially resolved by asking the question, Does the Cname for the directory begin /home? If it does, then the path that was evaluated to get us to the current directory must have gone through this mount point, and we should back up through it to evaluate ..; if not, then this mount table entry is irrelevant.

More precisely, both *before* and *after* each .. element in the path name is evaluated, if the directory is a tofile in the mount table, the corresponding fromfile is taken instead, provided the Cname of the corresponding fromfile is the prefix of the Cname of the original directory. Since we always know the full name of the directory we are evaluating, we can always compare it against all the entries in the mount table that point to it, thereby resolving ambiguous situations and maintaining the lexical property of ... This check also guarantees we don't follow a misleading mount point, such as the entry pointing to /home when we are really in /n/bopp/v7/rob. Keeping the full names with the Channels makes it easy to use the mount table to decide how we got here and, therefore, how to get back.

In summary, the algorithm is as follows. Use the usual file system operations to walk to ..; call the resulting directory *d*. Lexically remove the last element of the initial file name. Examine all entries in the mount table whose tofile is *d* and whose fromfile has a Cname identical to the truncated name. If one exists, that fromfile is the correct result; by construction, it also has the right Cname. In our example, evaluating .. in /home/rob (really /n/bopp/v7/rob) will set *d* to /n/bopp/v7; that is a tofile whose fromfile is /home. Removing the /rob from the original Cname, we find the name /home, which matches that of the fromfile, so the result is the fromfile, /home.

Since this implementation uses only local operations to maintain its names, it is possible to confuse it by external changes to the file system. Deleting or renaming directories and files that are part of a Cname, or modifying the mount table, can introduce errors. With more implementation work, such mistakes could probably be caught, but in a networked environment, with machines sharing a remote file server, renamings and deletions made by one machine may go unnoticed by others. These problems, however, are minor, uncommon and, most important, easy to understand. The method maintains the lexical property of file names unless an external agent changes the name surreptitiously; within a stable file system, it is always maintained and right.

To recapitulate, maintaining the Channel's absolute file names lexically and using the names to disambiguate the mount table entries when evaluating .. at a mount point combine to maintain the lexical definition of .. efficiently.

Cleaning names

The lexical processing can generate names that are messy or redundant, ones with extra slashes or embedded ../ or ./ elements and other extraneous artifacts. As part of the kernel's implementation, we wrote a procedure, cleanname, that rewrites a name in place to canonicalize its appearance. The procedure is useful enough that it is now part of the Plan 9 C library and is employed by many programs to make sure they always present clean file names.

Cleanname is analogous to the URL-cleaning rules defined in RFC 1808 [Field95], although the rules are slightly different. Cleanname iteratively does the following until no further processing can be done:

1. Reduce multiple slashes to a single slash.
2. Eliminate . path name elements (the current directory).
3. Eliminate .. path name elements (the parent directory) and the non-. non-.., element that precedes them.
4. Eliminate .. elements that begin a rooted path, that is, replace /.. by / at the beginning of a path.
5. Leave intact .. elements that begin a non-rooted path.

If the result of this process is a null string, cleanname returns the string ".", representing the current directory.

The fd2path system call

Plan 9 has a new system call, fd2path, to enable programs to extract the Cname associated with an open file descriptor. It takes three arguments: a file descriptor, a buffer, and the size of the buffer:

```
int fd2path(int fd, char *buf, int nbuf)
```

It returns an error if the file descriptor is invalid; otherwise it fills the buffer with the name associated with fd. (If the name is too long, it is truncated; perhaps this condition should also draw an error.) The fd2path system call is very cheap, since all it does is copy the Cname string to user space.

The Plan 9 implementation of getwd uses fd2path rather than the tricky algorithm necessary in Unix:

```
char*
getwd(char *buf, int nbuf)
{
        int n, fd;

        fd = open(".", OREAD);
        if(fd < 0)
                return NULL;
        n = fd2path(fd, buf, nbuf);
        close(fd);
        if(n < 0)
                return NULL;
        return buf;
}
```

(The Unix specification of getwd does not include a count argument.) This version of getwd is not only straightforward, it is very efficient, reducing the performance advantage of a built-in `pwd` command while guaranteeing that all commands, not just `pwd`, see sensible directory names.

Here is a routine that prints the file name associated with each of its open file descriptors; it is useful for tracking down file descriptors left open by network listeners, text editors that spawn commands, and the like:

```
void
openfiles(void)
{
        int  i;
        char  buf[256];

        for(i=0;  i<NFD;  i++)
                if(fd2path(i,  buf,  sizeof  buf)  >=  0)
                        print("%d:  %s\n",  i,  buf);
}
```

Uses of good names

Although `pwd` was the motivation for getting names right, good file names are useful in many contexts and have become a key part of the Plan 9 programming environment. The compilers record in the symbol table the full name of the source file, which makes it easy to track down the source of buggy, old software and also permits the implementation of a program, `src`, to automate tracking it down. Given the name of a program, `src` reads its symbol table, extracts the file information, and triggers the editor to open a window on the program's source for its `main` routine. No guesswork, no heuristics.

The `openfiles` routine was the inspiration for a new file in the `/proc` file system [Kill84]. For process *n*, the file `/proc/`*n*`/fd` is a list of all its open files, including its working directory, with associated information including its open status, I/O offset, unique id (analogous to i-number) and file name. Here is the contents of the `fd` file for a process in the window system on the machine being used to write this paper:

```
% cat /proc/125099/fd
/usr/rob
  0 r  M 5141  00000001.00000000          0 /mnt/term/dev/cons
  1 w  M 5141  00000001.00000000         51 /mnt/term/dev/cons
  2 w  M 5141  00000001.00000000         51 /mnt/term/dev/cons
  3 r  M 5141  0000000b.00000000       1166 /dev/snarf
  4 rw M 5141  0ffffffc.00000000        288 /dev/draw/new
  5 rw M 5141  00000036.00000000    4266337 /dev/draw/3/data
  6 r  M 5141  00000037.00000000          0 /dev/draw/3/refresh
  7 r  c    0  00000004.00000000    6199848 /dev/bintime
%
```

(The Linux implementation of `/proc` provides a related service by giving a directory in which each file-descriptor-numbered file is a symbolic link to the file itself.) When debugging errant systems software, such information can be valuable.

Another motivation for getting names right was the need to extract from the system an accurate description of the mount table, so that a process's name space could be recreated on another machine, in order to move (or simulate) a computing environment across the network. One program that does this is Plan 9's `cpu` command, which recreates the local name space on a remote machine, typically a large fast multiprocessor. Without accurate names, it was impossible to do the job right; now `/proc` provides a description of the name space of each process, `/proc/`*n*`/ns`:

```
% cat /proc/125099/ns
bind / /
mount -aC #s/boot /
bind #c /dev
bind #d /fd
bind -c #e /env
bind #p /proc
bind -c #s /srv
bind /386/bin /bin
bind -a /rc/bin /bin
bind /net /net
bind -a #l /net
mount -a #s/cs /net
mount -a #s/dns /net
bind -a #D /net
mount -c #s/boot /n/emelie
bind -c /n/emelie/mail /mail
mount -c /net/il/134/data /mnt/term
bind -a /usr/rob/bin/rc /bin
bind -a /usr/rob/bin/386 /bin
mount #s/boot /n/emelieother other
bind -c /n/emelieother/rob /tmp
mount #s/boot /n/dump dump
bind /mnt/term/dev/cons /dev/cons
...
cd /usr/rob
%
```

(The # notation identifies raw device drivers so they may be attached to the name space.) The last line of the file gives the working directory of the process. The format of this file is that used by a library routine, newns, which reads a textual description like this and reconstructs a name space. Except for the need to quote # characters, the output is also a shell script that invokes the user-level commands bind and mount, which are just interfaces to the underlying system calls. However, files like /net/il/134/data represent network connections; to find out where they point, so that the corresponding calls can be reestablished for another process, they must be examined in more detail using the network device files [PrWi93]. Another program, ns, does this; it reads the /proc/ *n*/ns file, decodes the information, and interprets it, translating the network addresses and quoting the names when required:

```
...
mount -a '#s/dns' /net
...
mount -c il!135.104.3.100!12884 /mnt/term
...
```

These tools make it possible to capture an accurate description of a process's name space and recreate it elsewhere. And like the open file descriptor table, they are a boon to debugging; it is always helpful to know exactly what resources a program is using.

Adapting to Unix

This work was done for the Plan 9 operating system, which has the advantage that the non-hierarchical aspects of the name space are all known to the kernel. It should be possible, though, to adapt it to a Unix system. The problem is that Unix has nothing corresponding precisely to a Channel, which in Plan 9 represents the unique result of evaluating a name. The vnode structure is a shared structure that may represent a file known by several names, while the file structure refers only to open files, but for example the current working directory of a process is not open. Possibilities to address this discrepancy include introducing a Channel-like structure that connects a name and a vnode, or maintaining a separate per-process table that maps names to vnodes, disambiguating using the techniques described here. If it could be done the result would be an implementation that reduces the need for a built-in pwd in the shell and offers a consistent, sensible interpretation of the parent directory.

We have not done this adaptation, but we recommend that the Unix community try it.

Conclusions

It should be easy to discover a well-defined, absolute path name for every open file and directory in the system, even in the face of symbolic links and other non-hierarchical elements of the file name space. In earlier versions of Plan 9, and all current versions of Unix, names can instead be inconsistent and confusing.

The Plan 9 operating system now maintains an accurate name for each file, using inexpensive lexical operations coupled with local file system actions. Ambiguities are resolved by examining the names themselves; since they reflect the path that was used to reach the file, they also reflect the path back, permitting a dependable answer to be recovered even when stepping backwards through a multiply-named directory.

Names make sense again: they are sensible and consistent. Now that dependable names are available, system services can depend on them, and recent work in Plan 9 is doing just that. We the community of Unix and Unix-like systems should have done this work a long time ago.

Acknowledgements

Phil Winterbottom devised the ns command and the fd and ns files in /proc, based on an earlier implementation of path name management that the work in this paper replaces. Russ Cox wrote the final version of cleanname and helped debug the code for reversing the mount table. Ken Thompson, Dave Presotto, and Jim McKie offered encouragement and consultation.

References

[Field95] R. Fielding, Relative Uniform Resource Locators, *Network Working Group Request for Comments: 1808*, June, 1995.

[Kill84] T. J. Killian, Processes as Files, *Proceedings of the Summer 1984 USENIX Conference*, Salt Lake City, 1984, pp. 203-207.

[Korn94] David G. Korn, ksh: An Extensible High Level Language, *Proceedings of the USENIX Very High Level Languages Symposium*, Santa Fe, 1994, pp. 129-146.

[Korn00] David G. Korn, personal communication.

[PeMc95] Jan-Simon Pendry and Marshall Kirk McKusick, Union Mounts in 4.4BSD-Lite, *Proceedings of the 1995 USENIX Conference*, New Orleans, 1995.

[Pike91] Rob Pike, 8½, the Plan 9 Window System, *Proceedings of the Summer 1991 USENIX Conference*, Nashville, 1991, pp. 257-265.

[Pike93] Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, and Phil Winterbottom, The Use of Name Spaces in Plan 9, *Operating Systems Review*, **27**, 2, April 1993, pp. 72-76.

[PrWi93] Dave Presotto and Phil Winterbottom, The Organization of Networks in Plan 9, *Proceedings of the Winter 1993 USENIX Conference*, San Diego, 1993, pp. 43-50.

Process Sleep and Wakeup on a Shared-memory Multiprocessor

Rob Pike
Dave Presotto
Ken Thompson
Gerard Holzmann

rob,presotto,ken,gerard@plan9.bell-labs.com

ABSTRACT

The problem of enabling a sleeping process on a shared-memory multiprocessor is a difficult one, especially if the process is to be awakened by an interrupt-time event. We present here the code for sleep and wakeup primitives that we use in our multiprocessor system. The code has been exercised by years of active use and by a verification system.

Our problem is to synchronise processes on a symmetric shared-memory multiprocessor. Processes suspend execution, or *sleep*, while awaiting an enabling event such as an I/O interrupt. When the event occurs, the process is issued a *wakeup* to resume its execution. During these events, other processes may be running and other interrupts occurring on other processors.

More specifically, we wish to implement subroutines called sleep, callable by a process to relinquish control of its current processor, and wakeup, callable by another process or an interrupt to resume the execution of a suspended process. The calling conventions of these subroutines will remain unspecified for the moment.

We assume the processors have an atomic test-and-set or equivalent operation but no other synchronisation method. Also, we assume interrupts can occur on any processor at any time, except on a processor that has locally inhibited them.

The problem is the generalisation to a multiprocessor of a familiar and well-understood uniprocessor problem. It may be reduced to a uniprocessor problem by using a global test-and-set to serialise the sleeps and wakeups, which is equivalent to synchronising through a monitor. For performance and cleanliness, however, we prefer to allow the interrupt handling and process control to be multiprocessed.

Our attempts to solve the sleep/wakeup problem in Plan 9 [Pik90] prompted this paper. We implemented solutions several times over several months and each time convinced ourselves — wrongly — they were correct. Multiprocessor algorithms can be difficult to prove correct by inspection and formal reasoning about them is impractical. We finally developed an algorithm we trust by verifying our code using an empirical testing tool. We present that code here, along with some comments about the process by which it was designed.

Appeared in a slightly different form in *Proceedings of the Spring 1991 EurOpen Conference*, Tromsø, Norway, 1991, pp. 161-166.

History

Since processes in Plan 9 and the UNIX system have similar structure and properties, one might ask if UNIX `sleep` and `wakeup` [Bac86] could not easily be adapted from their standard uniprocessor implementation to our multiprocessor needs. The short answer is, no.

The UNIX routines take as argument a single global address that serves as a unique identifier to connect the wakeup with the appropriate process or processes. This has several inherent disadvantages. From the point of view of `sleep` and `wakeup`, it is difficult to associate a data structure with an arbitrary address; the routines are unable to maintain a state variable recording the status of the event and processes. (The reverse is of course easy we could require the address to point to a special data structure but we are investigating UNIX `sleep` and `wakeup`, not the code that calls them.) Also, multiple processes sleep on a given address, so `wakeup` must enable them all, and let process scheduling determine which process actually benefits from the event. This is inefficient; a queueing mechanism would be preferable but, again, it is difficult to associate a queue with a general address. Moreover, the lack of state means that `sleep` and `wakeup` cannot know what the corresponding process (or interrupt) is doing; `sleep` and `wakeup` must be executed atomically. On a uniprocessor it suffices to disable interrupts during their execution. On a multiprocessor, however, most processors can inhibit interrupts only on the current processor, so while a process is executing `sleep` the desired interrupt can come and go on another processor. If the wakeup is to be issued by another process, the problem is even harder. Some inter-process mutual exclusion mechanism must be used, which, yet again, is difficult to do without a way to communicate state.

In summary, to be useful on a multiprocessor, UNIX `sleep` and `wakeup` must either be made to run atomically on a single processor (such as by using a monitor) or they need a richer model for their communication.

The design

Consider the case of an interrupt waking up a sleeping process. (The other case, a process awakening a second process, is easier because atomicity can be achieved using an interlock.) The sleeping process is waiting for some event to occur, which may be modeled by a condition coming true. The condition could be just that the event has happened, or something more subtle such as a queue draining below some low-water mark. We represent the condition by a function of one argument of type `void*`; the code supporting the device generating the interrupts provides such a function to be used by `sleep` and `wakeup` to synchronise. The function returns `false` if the event has not occurred, and `true` some time after the event has occurred. The `sleep` and `wakeup` routines must, of course, work correctly if the event occurs while the process is executing `sleep`.

We assume that a particular call to `sleep` corresponds to a particular call to `wakeup`, that is, at most one process is asleep waiting for a particular event. This can be guaranteed in the code that calls `sleep` and `wakeup` by appropriate interlocks. We also assume for the moment that there will be only one interrupt and that it may occur at any time, even before `sleep` has been called.

For performance, we desire that multiple instances of `sleep` and `wakeup` may be running simultaneously on our multiprocessor. For example, a process calling `sleep` to await a character from an input channel need not wait for another process to finish executing `sleep` to await a disk block. At a finer level, we would like a process reading from one input channel to be able to execute `sleep` in parallel with a process reading from another input channel. A standard approach to synchronisation is to interlock the channel driver so that only one process may be executing in the channel code at once. This method is clearly inadequate for our purposes; we need fine-grained synchronisation, and in particular to apply interlocks at the level of individual channels rather than at the level of the channel driver.

Our approach is to use an object called a *rendezvous*, which is a data structure through which sleep and wakeup synchronise. (The similarly named construct in Ada is a control structure; ours is an unrelated data structure.) A rendezvous is allocated for each active source of events: one for each I/O channel, one for each end of a pipe, and so on. The rendezvous serves as an interlockable structure in which to record the state of the sleeping process, so that sleep and wakeup can communicate if the event happens before or while sleep is executing.

Our design for sleep is therefore a function

```
void sleep(Rendezvous *r, int (*condition)(void*), void *arg)
```

called by the sleeping process. The argument r connects the call to sleep with the call to wakeup, and is part of the data structure for the (say) device. The function condition is described above; called with argument arg, it is used by sleep to decide whether the event has occurred. Wakeup has a simpler specification:

```
void wakeup(Rendezvous *r).
```

Wakeup must be called after the condition has become true.

An implementation

The Rendezvous data type is defined as

```
typedef struct{
        Lock    l;
        Proc    *p;
}Rendezvous;
```

Our Locks are test-and-set spin locks. The routine lock(Lock *l) returns when the current process holds that lock; unlock(Lock *l) releases the lock.

Here is our implementation of sleep. Its details are discussed below. Thisp is a pointer to the current process on the current processor. (Its value differs on each processor.)

```
void
sleep(Rendezvous *r, int (*condition)(void*), void *arg)
{
	int s;

	s = inhibit();		/* interrupts */
	lock(&r->l);

	/*
	 * if condition happened, never mind
	 */
	if((*condition)(arg)){
		unlock(&r->l);
		allow();	/* interrupts */
		return;
	}

	/*
	 * now we are committed to
	 * change state and call scheduler
	 */
	if(r->p)
		error("double sleep %d %d", r->p->pid, thisp->pid);
	thisp->state = Wakeme;
	r->p = thisp;
	unlock(&r->l);
	allow(s);	/* interrupts */
	sched();	/* relinquish CPU */
}
```

Here is *wakeup*.

```
void
wakeup(Rendezvous *r)
{
	Proc *p;
	int s;

	s = inhibit();	/* interrupts; return old state */
	lock(&r->l);
	p = r->p;
	if(p){
		r->p = 0;
		if(p->state != Wakeme)
			panic("wakeup: not Wakeme");
		ready(p);
	}
	unlock(&r->l);
	if(s)
		allow();
}
```

Sleep and *wakeup* both begin by disabling interrupts and then locking the rendezvous structure. Because *wakeup* may be called in an interrupt routine, the lock must be set only with interrupts disabled on the current processor, so that if the interrupt comes during *sleep* it will occur only on a different processor; if it occurred on the processor executing *sleep*, the spin lock in *wakeup* would hang forever. At the end of each routine, the lock is released and processor priority returned to its previous value. (*Wakeup* needs to inhibit interrupts in case it is being called by a process; this is a no-op if called by an interrupt.)

Sleep checks to see if the condition has become true, and returns if so. Otherwise the process posts its name in the rendezvous structure where wakeup may find it, marks its state as waiting to be awakened (this is for error checking only) and goes to sleep by calling sched(). The manipulation of the rendezvous structure is all done under the lock, and wakeup only examines it under lock, so atomicity and mutual exclusion are guaranteed.

Wakeup has a simpler job. When it is called, the condition has implicitly become true, so it locks the rendezvous, sees if a process is waiting, and readies it to run.

Discussion

The synchronisation technique used here is similar to known methods, even as far back as Saltzer's thesis [Sal66]. The code looks trivially correct in retrospect: all access to data structures is done under lock, and there is no place that things may get out of order. Nonetheless, it took us several iterations to arrive at the above implementation, because the things that *can* go wrong are often hard to see. We had four earlier implementations that were examined at great length and only found faulty when a new, different style of device or activity was added to the system.

Here, for example, is an incorrect implementation of wakeup, closely related to one of our versions.

```
void
wakeup(Rendezvous *r)
{
        Proc *p;
        int s;

        p = r->p;
        if(p){
                s = inhibit();
                lock(&r->l);
                r->p = 0;
                if(p->state != Wakeme)
                        panic("wakeup: not Wakeme");
                ready(p);
                unlock(&r->l);
                if(s)
                        allow();
        }
}
```

The mistake is that the reading of r->p may occur just as the other process calls sleep, so when the interrupt examines the structure it sees no one to wake up, and the sleeping process misses its wakeup. We wrote the code this way because we reasoned that the fetch p = r->p was inherently atomic and need not be interlocked. The bug was found by examination when a new, very fast device was added to the system and sleeps and interrupts were closely overlapped. However, it was in the system for a couple of months without causing an error.

How many errors lurk in our supposedly correct implementation above? We would like a way to guarantee correctness; formal proofs are beyond our abilities when the subtleties of interrupts and multiprocessors are involved. With that in mind, the first three authors approached the last to see if his automated tool for checking protocols [Hol91] could be used to verify our new sleep and wakeup for correctness. The code was translated into the language for that system (with, unfortunately, no way of proving that the translation is itself correct) and validated by exhaustive simulation.

The validator found a bug. Under our assumption that there is only one interrupt, the bug cannot occur, but in the more general case of multiple interrupts synchronising through the same condition function and rendezvous, the process and interrupt can enter a peculiar state. A process may return from sleep with the condition function false if there is a delay between the condition coming true and wakeup being called, with the delay occurring just as the receiving process calls sleep. The condition is now true, so that process returns immediately, does whatever is appropriate, and then (say) decides to call sleep again. This time the condition is false, so it goes to sleep. The wakeup process then finds a sleeping process, and wakes it up, but the condition is now false.

There is an easy (and verified) solution: at the end of sleep or after sleep returns, if the condition is false, execute sleep again. This re-execution cannot repeat; the second synchronisation is guaranteed to function under the external conditions we are supposing.

Even though the original code is completely protected by interlocks and had been examined carefully by all of us and believed correct, it still had problems. It seems to us that some exhaustive automated analysis is required of multiprocessor algorithms to guarantee their safety. Our experience has confirmed that it is almost impossible to guarantee by inspection or simple testing the correctness of a multiprocessor algorithm. Testing can demonstrate the presence of bugs but not their absence [Dij72].

We close by claiming that the code above with the suggested modification passes all tests we have for correctness under the assumptions used in the validation. We would not, however, go so far as to claim that it is universally correct.

References

[Bac86] Maurice J. Bach, *The Design of the UNIX Operating System*, Prentice-Hall, Englewood Cliffs, 1986.

[Dij72] Edsger W. Dijkstra, The Humble Programmer 1972 Turing Award Lecture, *Comm. ACM*, 15(10), pp. 859-866, October 1972.

[Hol91] Gerard J. Holzmann, *Design and Validation of Computer Protocols*, Prentice-Hall, Englewood Cliffs, 1991.

[Pik90] Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, Plan 9 from Bell Labs, *Proceedings of the Summer 1990 UKUUG Conference*, pp. 1-9, London, July, 1990.

[Sal66] Jerome H. Saltzer, *Traffic Control in a Multiplexed Computer System*, MIT, Cambridge, Mass., 1966.

A Guide to the Lp Printer Spooler

Paul Glick
pg@plan9.bell-labs.com

ABSTRACT

Lp is a collection of programs used to provide an easy-to-use interface for printing a variety of document types on a variety of printers. *Lp* is the glue that connects various document language translators and printer communication programs together so that the users may have a consistent view of printers. Most of the glue is shell scripts, which can be easily modified. The user need not specify options to get sensible output in most cases. *Lp* is described here so that others may make additions and changes.

1. Introduction

Lp is used to format and print data on a variety of output devices. The need for *lp* was rooted in the inability of other printer spoolers to do simple tasks without a great deal of user specification of options. At the time *lp* was written, there were several printer languages, such as ImPress and PostScript, and an internally developed printer that would accept *troff* output. Now, all our printers take PostScript, but printers that use HPCL and HPGL abound and support for those printers may be added easily. A great deal of what underlies *lp* is taken from BSD's *lpr* and System V's *lp*. The important features of this system are that most of the programs are easily modified shell scripts and the user need not learn to use the large amount of underlying software developed by others. *Lp* runs under Plan 9 and several flavors of UNIX. This document deals with *lp* as it relates to Plan 9. *Lp* was developed using both Datakit and Ethernet to transport data between machines. Now only the Ethernet transport mechanism remains.

Text, graphics, and formatted text files are appropriately processed and placed into a spool directory from which they are taken to be printed by a daemon process. Additional functions include checking the status of a printer queue and removing jobs from the printer queue.

All the shell scripts (see *rc*(1)) associated with *lp* reside in the spool directory /sys/lib/lp except for the *lp* command itself, which resides in /rc/bin. Commands related to *lp* that are not shell scripts can most often be found in /$cputype/bin/aux. The directory where all the *lp* scripts reside is defined within *lp* by the shell variable LPLIB. In the remainder of this document, file names will be specified with this shell variable as their root.

2. Usage

Lp requires an output device to be specified before it will process input. This can be done in any of three ways described here.

1) The file $LPLIB/defdevice may contain the name of a default output device. This may not be practical for environments where there are many printers.

2) The user's environment variable LPDEST may be set to the name of the device to

be used. This is often a more practical solution when there are several printers available. This overrides the default device specification.

3) The -d *printer* option to the *lp* command specifies *printer* as the device to which output should be directed, overriding the previous two specifications.

If *printer* is ?, a list of printers and other information in the devices file is printed, as shown in Figure 1. Quote the question mark to prevent it from being interpreted by the shell language as a metacharacter.

```
% lp -d'?'
device     location    host              class
fn         2C-501      helix             post/2+600dpi+duplex
pcclone    -           -                 post+nohead
peacock    2C-501      cetus             post/2+300dpi+nohead+color
ps83       st8_fl3     rice              post+300dpi+reverse
psu        2C-501      cetus             post/2+1200dpi
     .
     .
     .
%
```

Figure 1. Sample listing of installed printers

Normally, *lp* uses the file command to figure out what type of input it is receiving. This is done within the generic process which is discussed later in this paper in the **Process directory** section. To select a specific input processor the -p *process* option is used where *process* is one of the shell scripts in the process directory.

Troff output can be printed, in this case, on printer *fn* with

```
% troff -ms lp.ms | lp -dfn
```

A file can be converted to PostScript using the pseudo-printer stdout:

```
% troff -ms lp.ms | lp -dstdout > lp.ps
```

LaTeX (and analogously TeX) documents are printed in two steps:

```
% latex lp.tex
     .
     .
% lp lp.dvi
     .
     .
%
```

LaTeX produces a .dvi file and does not permit the use of a pipe connection to the standard input of *lp*. To look at the status and queue of a device, use -q:

```
% lp -dpsu -q
daemon status:
: 67.17% sent
printer status:
%%[ status: busy; source: lpd ]%%

queue on cetus:
job              user     try     size
rice29436.1      pg       0       17454
slocum17565.1    ches     1       49995
%
```

This command can print the status and queue of the local and remote hosts. Administrators should be advised that working in an environment where the *lp* spool directory is shared among the local and remote hosts, no spooling should be done on the local hosts. The format of the status and queue printout is up to the administrator. The job started above can be killed with -k:

```
$ lp -dpsu -k rice29436.1
rice29436.1   removed  from  psu  queue  on  cetus
```

3. Options

There are options available to modify the way in which a job is handled. It is the job of the *lp* programs to convert the option settings so they may be used by each of the different translation and interface programs. Not all options are applicable to all printer environments. Table 1 lists the standard *lp* options, the shell variable settings, and description of the options.

4. Devices file

The devices file is found in the spool directory. Each line in the file is composed of 12 fields, separated by tabs or spaces, that describe the attributes of the printer and how it should be serviced. Within the lp command, a shell variable is set for each attribute; the following list describes them:

LPDEST is the name of the device as given to *lp* with the -d option or as specified by the shell environment variable LPDEST or as specified by the file $LPLIB/defdevice . This name is used in creating directories and log files that are associated with the printers operation.

LOC just describes where the printer is physically located.

DEST_HOST is the host from which the files are printed. Files may be spooled on other machines before being transferred to the destination host.

OUT_DEV is the physical device name or network address needed by the printer daemon to connect to the printer. This field depends on the requirements of the daemon and may contain a if not required.

SPEED is the baud rate setting for the port. This field depends on the requirements of the daemon and may contain a if not required.

LPCLASS is used to encode minor printer differences. The keyword reverse is used by some of the preprocessors to reverse the order the pages are printed to accommodate different output trays (either face up or face down). The keyword nohead is used to suppress the header page. This is used for special and color printers. The keyword duplex is used to coax double sided output from duplex printers.

LPPROC is the command from the LPLIB/process directory to be used to convert input to a format that will be accepted by the device. The preprocessor is invoked by the spooler.

SPOOLER is the command from the LPLIB/spooler directory which will select files using the SCHED command and invoke the LPPROC command, putting its output into the remote spool directory. The output is sent directly to the spool directory on the destination machine to avoid conflicts when client and server machines share spool directories.

STAT is the command from the LPLIB/stat directory that prints the status of the device and the list of jobs waiting on the queue for the device. The status information depends on what is available from the printer and interface software. The queue information should be changed to show information

option	shell variable name	default	set	action
-D	DEBUG	N	1	turn on debugging mode.
-H	NOHEADER	N	1	suppress header page.
-L	LAND	N	1	make long page dimension horizontal.
-M *mach*	LPMACHID	N	*mach*	set the source machine name.
-Q	QONLY	N	1	do not execute daemon; for debugging.
-c *n*	COPIES	N	*n*	number of copies to be printed.
-d *printer*	LPDEST	U	*printer*	set job destination, override other settings.
-f *font.pt*	FONT POINT	N N	*font* *pt*	set font style and point size for printing.
-i *n*	IBIN	N	*n*	select input paper tray options. The argument given is dependent on the printer type. A number can be given to select a particular tray and/or simplex or duplex may be used to get single or double sided output, where applicable. Multiple options should be separated by commas.
-k	KILLFLAG	0	1	take non-option arguments as job numbers to be removed from queue.
-l *n*	LINES	N	*n*	for printed data, the number of lines per logical page.
-m *f*	MAG	N	*f*	magnify the image by a factor *f*. The factor should be a positive real number.
-n *n*	NPAG	N	*n*	put *n* logical pages on a single physical page. A simple algorithm is used to pack the pages.
-o *list*	OLIST	N	*list*	print only those pages specified in the list. The list may be a sequence of numbers or ranges separated by commas. A range is a pair of numbers separated by a hyphen.
-p *proc*	LPPROC	L	*proc*	use the preprocessor *proc* instead of the preprocessor given in the devices file for this printer.
-q	LPQ	N	1	print the status and queue.
-r	REVERSE	L	1	this toggles the REVERSE flag, changing whether or not page reversal should occur in preprocessing. Page reversal is needed if a printer delivers pages face up. The keyword reverse can be placed in the *lpclass* field of the devices file. If a document has already been processed this flag has no effect.
-u *user*	LPUSERID	U	*user*	change the user id that appears on the cover page.
-x *offset*	XOFF	N	*offset*	move the image *offset* inches to the right. A negative *offset* will move the image to the left. The *offset* may be any reasonable real number.
-y *offset*	YOFF	N	*offset*	same as for -x except a positive offset will move the image down.

default	setting definition
N	set to the null string ( ) initially in *lp*.
L	set from printer entry in devices file.
U	set from the users environment.

Table 1. Lp Option List

	useful in tracking down problems. The SCHED command is used to show the jobs in the order in which they will be printed.
KILL	is the command from the LPLIB/kill that removes jobs from the queue. The jobs to be removed are given as arguments to the *lp* command. When possible, it should also abort the currently running job if it has to be killed.
DAEMON	is the command from the LPLIB/daemon that is meant to run asynchronously to remove jobs from the queue. Jobs may either be passed on to another host or sent to the printing device. *Lp* always tries to start a daemon process when one is specified.
SCHED	is the command from the LPLIB/sched that is used to present the job names to the daemon and stat programs in some order, e.g., first-in-first-out, smallest first.

5. Support programs

The following sections describe the basic functions of the programs that are found in the subdirectories of $LPLIB . The programs in a specific directory vary with the type of output device or networks that have to be used.

5.1. Process directory

The generic preprocessor is the default preprocessor for most printers. It uses the *file* (1) command to determine the format of the input file. The appropriate preprocessor is then selected to transform the file to a format suitable for the printer.

Here is a list of some of the preprocessors and a description of their function. A complete list of preprocessors and their descriptions can be found in the manual page *lp* (8).

dvipost	Converts TeX or LaTeX output (dvi files) to PostScript
ppost	Converts UTF text to PostScript. The default font is Courier with Lucida fonts filling in the remainder of the (available) Unicode character space.
tr2post	Converts (device independent) troff output for the device type utf . See /sys/lib/troff/font/devutf directory for troff font width table descriptions. See also the /sys/lib/postscript/troff directory for mappings of troff UTF character space to PostScript font space.
p9bitpost	Converts Plan 9 bitmaps (see *bitfile* (9.6)) to PostScript.
g3post	Converts fax (CCITT-G31 format) to PostScript.
hpost	Does header page processing and page reversal processing, if necessary. Page reversal is done here so the header page always comes out at the beginning of the job. Header page processing is very location-dependent.

5.2. Spool directory

The generic spooler is responsible for executing the preprocessor and directing its output to a file in the printer's queue. An additional file is created containing information such as the system name, userid, job number, and number of times this job was attempted.

Certain printer handling programs do not require separate preprocessing and spooling. For such circumstances a nospool spooler is available that just executes the preprocessing program. The processing and spooling functions are assumed by this program and the output is sent to OUT_DEV or standard output if OUT_DEV is - .

The pcclone spooler is used to send print jobs directly to a printer connected to a 386 compatible printer port (See *lpt*(3)).

5.3. Stat directory

The function of the shell scripts in the stat directory is to present status information about the printer and its queue. When necessary, the stat scripts may be designed to return information about the local queue as well as the remote queue. This is not done on Plan 9 because many systems share the same queue directory. The scheduler is used to print the queue in the order in which the jobs will be executed.

5.4. Kill directory

The kill scripts receive command line arguments passed to them by *lp* and remove the job and id files which match the arguments for the particular queue. When a job is killed, the generic kill procedure:

1) kills the daemon for this queue if the job being killed is first in the queue,

2) removes the files associated with the job from the queue,

3) attempts to restart the daemon.

5.5. Daemon directory

The daemon shell scripts are the last to be invoked by *lp* if the -Q option has not been given. The daemon process is executed asynchronously with its standard output and standard error appended to the printer log file. The log file is described in a subsequent section. Because the daemon runs asynchronously, it must catch signals that could cause it to terminate abnormally. The daemon first checks to see that it is the only one running by using the LOCK program found in the /$cputype/bin/aux directory. The LOCK command creates a LOCK file in the printer's queue directory. The daemon then executes the scheduler to obtain the name of the next job on the queue.

The processing of jobs may entail transfer to another host or transmission to a printer. The details of this are specific to the individual daemons. If a job is processed without error, it is removed from the queue. If a job does not succeed, the associated files may be moved to a printer specific directory in $LPLIB/prob. In either case, the daemon can make an entry in the printer's log file. Before exiting, the daemon should clean up lock files by calling UNLOCK.

Several non-standard daemon programs have been designed to suit various requirements and whims. One such program announces job completion and empty paper trays by causing icons to appear in peoples' seemail window. Another, using a voice synthesizer, makes verbal announcements. Other daemons may be designed to taste.

5.6. Sched directory

The scheduler must decide which job files should be executed and in what order. The most commonly used scheduler program is FIFO, which looks like this:

```
ls -tr $* | sed -n -e 's/.*   *//' \
  -e '/^[0-9][0-9]*.[1-9][0-9]*$/p'
```

This lists all the job files in this printer's queue in modification time order. Jobs entering the queue have a dot (.) prefixed to their name to keep the scheduler from selecting them before they are complete.

6. Where Things Go Wrong

There are four directories *lp* writes files. On the Plan 9 release these directories may be found in a directory on a scratch filesystem that is not backed-up. This directory is /n/emelieother/lp. It is built on top of a file system other that is mounted on the fileserver emelie. The four directories in this scratch directory are log, prob, queue, and tmp. *Lp* binds (see *bind*(1)) the first three into the directory /sys/lib/lp for its processes and their children. The tmp directory is bound to the /tmp directory so that the lp daemons, which run as user none, may write into this directory.

On any new installation, it is important that these directories be set up and that the */rc/bin/lp* command be editted to reflect the change. If you do not have a scratch filesystem for these directories, create the four directories log, prob, queue, and tmp in $LPLIB (/sys/lib/lp) so that they are writable by anyone.

6.1. Log directory

The log files for a particular *printer* appear in a subdirectory of the spool directory log/*printer*. There are currently two types of log files. One is for the daemon to log errors and successful completions of jobs. These are named *printer.day* where *day* is the three letter abbreviation for the day of the week. These are overwritten once a week to avoid the need for regular cleanup. The other type of log file contains the status of the printer and is written by the program that communicates with the printer itself. These are named *printer*.st. These are overwritten with each new job and are saved in the $LPLIB/prob directory along with the job under circumstances described below. When a printer does not appear to be functioning these files are the place to look first.

6.2. Prob directory

When a job fails to produce output, the log files should be checked for any obvious problems. If none can be found, a directory with full read and write permissions should be created with the name of the printer in the $LPLIB/prob directory. Subsequent failure of a job will cause the daemon to leave a copy of the job and the printer communication log in $LPLIB/prob/*printer* directory. It is common for a printer to enter states from which it cannot be rescued except by manually cycling the power on the printer. After this is done the print daemon should recover by itself (give it a minute). If it does not recover, remove the LOCK file from the printer's spool directory to kill the daemon. The daemon will have to be restarted by sending another job to the printer. For PostScript printers just use:

```
echo '%!PS' | lp
```

6.3. Repairing Stuck Daemons

There are conditions that occur which are not handled by the daemons. One such problem can only be described as the printer entering a comatose state. The printer does not respond to any messages sent to it. The daemon should recover from the reset and an error message will appear in the log files. If all else fails, one can kill the first job in the queue or remove the LOCK file from the queue directory. This will kill the daemon, which will have to be restarted.

7. Interprocessor Communication

A Plan 9 CPU server can be set up as a printer's spooling host. That is, the machine where jobs are spooled and from which those jobs are sent directly to the printer. To do this, the CPU must listen on TCP port 515 which is the well known port for the BSD line printer daemon. The file /rc/bin/service/tcp515 is executed when a call comes in on that port. The Plan 9 lpdaemon will accept jobs sent from

BSD LPR/LPD systems. The /$cputype/bin/aux/lpdaemon command is executed from the service call and it accepts print jobs, requests for status, and requests to kill jobs. The command /$cputype/bin/aux/lpsend is used to send jobs to other Plan 9 machines and is usually called from within a spooler or daemon script. The command /$cputype/bin/aux/lpdsend is used to send jobs to machines and printers that use the BSD LPR/LPD protocol and is also usually called from within a spooler or daemon script.

8. Acknowledgements

Special thanks to Rich Drechsler for supplying and maintaining most of the PostScript translation and interface programs, without which *lp* would be an empty shell. Tomas Rokicki provided the TeX to PostScript translation program.

9. References

[Camp86] Ralph Campbell, 4.3BSD Line Printer Spooler Manual , UNIX System Manager s Manual, May, 1986, Berkeley, CA
[RFC1179] Request for Comments: 1179, Line Printer Daemon Protocol, Aug 1990
[Sys5] System V manual, date unknown

Troff User's Manual

Joseph F. Ossanna
Brian W. Kernighan

bwk@research.bell-labs.com

Introduction

Troff and *nroff* are text processors that format text for typesetter- and typewriter-like terminals, respectively. They accept lines of text interspersed with lines of format control information and format the text into a printable, paginated document having a user-designed style. *Troff* and *nroff* offer unusual freedom in document styling: arbitrary style headers and footers; arbitrary style footnotes; multiple automatic sequence numbering for paragraphs, sections, etc; multiple column output; dynamic font and point-size control; arbitrary horizontal and vertical local motions at any point; and a family of automatic overstriking, bracket construction, and line-drawing functions.

Troff produces its output in a device-independent form, although parameterized for a specific device; *troff* output must be processed by a driver for that device to produce printed output.

Troff and *nroff* are highly compatible with each other and it is almost always possible to prepare input acceptable to both. Conditional input is provided to enable the user to embed input expressly destined for either program. *Nroff* can prepare output directly for a variety of terminal types and is capable of utilizing the full resolution of each terminal. *Nroff* is the same program as *troff*; in fact, on Plan 9 *nroff* is a shell script that calls *troff* with the -N argument.

Background to the Plan 9 Edition

The primary change to *troff* and *nroff* for Plan 9 is support of the Unicode Standard, which was added during 1992 and 1993. There are two results. First, there is much less need for the myriad of two-character names that are so much a part of *troff* lore; in Plan 9, for example, one naturally uses the Unicode character ½ instead of *troff*'s \(12. Second, the output device, though called utf, is almost always a form of PostScript printer; the panoply of special drivers for different typesetters has largely disappeared. Unfortunately, not all PostScript printers can cope with Unicode characters, so there remains a need for programs that synthesize PostScript characters from bitmaps; this is especially true for Asian languages.

Background to the Second Edition

Troff was originally written by the late Joe Ossanna in about 1973, in assembly language for the PDP-11, to drive the Graphic Systems CAT typesetter. It was rewritten in C around 1975, and underwent slow but steady evolution until Ossanna's death late in 1977.

In 1979, Brian Kernighan modified *troff* so that it would produce output for a variety of typesetters, while retaining its input specifications. Over the decade from 1979 to 1989, the internals have been modestly revised, though much of the code remains as it was when Ossanna wrote it.

Troff reads parameter files each time it is invoked, to set values for machine resolution, legal type sizes and fonts, and character names, character widths and the like. *Troff* output is ASCII characters in a simple language that describes where each character is to be placed and in what size and font. A post-processor must be written for each device to convert this typesetter-independent language into specific instructions for that device.

The output language contains information that was not readily identifiable in the older output. In the newer language, the beginning of each page, line, and word is marked, so post-processors can do device-specific optimizations such as sorting the data vertically or printing it boustrophedonically, independent of *troff*.

Capabilities for graphics have been added: *troff* recognizes commands for drawing diagonal lines, circles, ellipses, circular arcs, and quadratic B-splines. There are also ways to pass arbitrary information to the output, unprocessed by *troff*.

A number of limitations have been eased or eliminated. A document may have an arbitrary number of fonts on any page (if the output device permits it, of course). Fonts may be accessed merely by naming them; mounting is no longer necessary. There are no limits on the number of characters. Character height and slant may be set independently of width.

The remainder of this document contains a description of usage and command-line options; a summary of requests, escape sequences, and pre-defined number registers; a reference manual; tutorial examples; and a list of commonly-available characters.

Acknowledgements

Joe Ossanna's *troff* remains a remarkable accomplishment. For more than twenty years, it has proven a robust tool, taking unbelievable abuse from a variety of preprocessors and being forced into uses that were never conceived of in the original design, all with considerable grace under fire.

Recent versions of *troff* have profited from significant code improvements by Jaap Akkerhuis, Dennis Ritchie, Ken Thompson, and Molly Wagner. UTF facilities owe much to Jaap Akkerhuis. Andrew Hume, Doug McIlroy, Peter Nelson and Ravi Sethi made valuable suggestions on the manual. I fear that the remaining bugs are my fault.

Usage

Troff or *nroff* is invoked as

```
troff    options   files
nroff    options   files
```

where *options* represent any of a number of option arguments and *files* represent the list of files containing the document to be formatted. An argument consisting of a single minus – represents standard input. If no filenames are given input is taken from the standard input. The options, which may appear in any order so long as they appear before the files, are:

-m *name*	Read the macro file /sys/lib/tmac.*name* before the input *files*.
-T *name*	Specifies the type of the output device. Specific devices are site-dependent. For *troff*, the most useful name is utf. For *nroff*, useful names include 37 for the (default) Model 37 Teletype, lp for dumb line printer terminals (no half-line motions, no reverse motions), and think for the HP ThinkJet printer.
-i	Read standard input after the input files are exhausted.
-o *list*	Print only pages whose page numbers appear in *list*, which consists of comma-separated numbers and number ranges. A number range has the form *N*–*M* and means pages *N* through *M*; a initial –*N* means from the beginning to page *N*; and a final *N*– means from *N* to the end.
-n *N*	Number first generated page *N*.
-r *aN*	Set number register *a* (one-character) to *N*.
-s *N*	Stop every *N* pages. *Nroff* will halt prior to every *N* pages (default *N*=1) to allow paper loading or changing, and will resume upon receipt of a newline. *Troff* will include a pause code every *N* pages; its meaning, if any, depends on the output device.
-u *N*	Set amount of emboldening for the bd request to *N*.
-F *path*	Look in directory *path* for font information; the defaults are /sys/lib/troff/font and /sys/lib/troff/term for *troff* and *nroff* respectively.
	troff **Only**
-a	Send a printable approximation of the results to the standard output.
	nroff **Only**
-e	Produce equally-spaced words in adjusted lines, using full terminal resolution.
-h	Use tabs instead of spaces to speed up printing.
-q	Invoke the simultaneous input-output mode of the rd request.

Each option is a separate argument; for example,

```
troff  -Tutf  -ms  -mpictures   -o4,6,8-10   file1 file2
```

requests formatting of pages 4, 6, and 8 through 10 of a document contained in the files named *file1* and *file2*, specifies the output in UTF, and invokes the macro packages -ms and -mpictures.

Various pre- and post-processors are available for use with *nroff* and *troff*. These include the equation preprocessor *eqn* (for *troff* only), the table-construction preprocessor *tbl*, and *pic* and *grap* for various forms of graphics.

Request Summary

In the following table, the notation ± *N* in the *Request Form* column means that the forms *N*, +*N*, or −*N* are permitted, to set the parameter to *N*, increment it by *N*, or decrement it by *N*, respectively. Plain *N* means that the value is used to set the parameter. *Initial Values* separated by ; are for *troff* and *nroff* respectively. In the *Notes* column,

B	Request normally causes a break. The use of ' as control character (instead of .) suppresses the break function.
D	Mode or relevant parameters associated with current diversion level.
E	Relevant parameters are a part of the current environment.
O	Must stay in effect until logical output.
P	Mode must be still or again in effect at the time of physical output.
T	*troff* only; no effect in *nroff*.
v, **p**, **m**, **u**	Default scale indicator; if not specified, scale indicators are ignored.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
1. **General Information**				
2. **Font and Character Size Control**				
.ps ±*N*	10 point	previous	E,T	Point size; also \s ± *N*.
.ss *N*	12/36 **m**	ignored	E,T	Space-character size set to *N*/36 em.
.cs *F N M*	off	-	P,T	Constant character space (width) mode (font *F*).
.bd *F N*	off	-	P,T	Embolden font *F* by *N*−1 units.
.bd S *F N*	off	-	P,T	Embolden Special Font when current font is *F*.
.ft *F*	Roman	previous	E	Change to font *F*; also \f *x*, \f(*xx*, \f *N*.
.fp *N F L*	R,I,B,...,S	ignored	-	Mount font named *F* on physical position *N*≤1; long name is *L* if given.
3. **Page Control**				
.pl ±*N*	11i	11i	**v**	Page length.
.bp ±*N*	*N*=1	-	B,**v**	Eject current page; next page number *N*.
.pn ±*N*	*N*=1	ignored	-	Next page number *N*.
.po ±*N*	1i;0	previous	**v**	Page offset.
.ne *N*	-	*N*=1v	D,**v**	Need *N* vertical space.
.mk *R*	none	internal	D	Mark current vertical place in register *R*.
.rt ±*N*	none	internal	D,**v**	Return (upward only) to marked vertical place.
4. **Text Filling, Adjusting, and Centering**				
.br	-	-	B	Break.
.fi	fill	-	B,E	Fill output lines.
.nf	fill	-	B,E	No filling or adjusting of output lines.
.ad *c*	adj,both	adjust	E	Adjust output lines with mode *c*; *c*=l,r,c,b,*none*
.na	adjust	-	E	No output line adjusting.
.ce *N*	off	*N*=1	B,E	Center next *N* input text lines.
5. **Vertical Spacing**				
.vs *N*	12p; 1/6i	previous	E,**p**	Vertical baseline spacing (*V*).
.ls *N*	*N*=1	previous	E	Output *N*−1 **v** s after each text output line.
.sp *N*	-	*N*=1v	B,**v**	Space vertical distance *N* in either direction.
.sv *N*	-	*N*=1v	**v**	Save vertical distance *N*.
.os	-	-	-	Output saved vertical distance.
.ns	space	-	D	Turn no-space mode on.
.rs	-	-	D	Restore spacing; turn no-space mode off.

6. Line Length and Indenting

.ll ±*N*	6.5i	previous	E,**m**	Line length.
.in ±*N*	*N*=0	previous	B,E,**m**	Indent.
.ti ±*N*	-	ignored	B,E,**m**	Temporary indent.

7. Macros, Strings, Diversion, and Position Traps

.de *xx yy*	-	.*yy*=..	-	Define or redefine macro *xx*; end at call of *yy*.
.am *xx yy*	-	.*yy*=..	-	Append to a macro.
.ds *xx string*	-	ignored	-	Define a string *xx* containing *string* .
.as *xx string*	-	ignored	-	Append *string* to string *xx*.
.rm *xx*	-	ignored	-	Remove request, macro, or string.
.rn *xx yy*	-	ignored	-	Rename request, macro, or string *xx* to *yy*.
.di *xx*	-	end	D	Divert output to macro *xx*.
.da *xx*	-	end	D	Divert and append to *xx*.
.wh *N xx*	-	-	**v**	Set location trap; negative is w.r.t. page bottom.
.ch *xx N*	-	-	**v**	Change trap location.
.dt *N xx*	-	off	D,**v**	Set a diversion trap.
.it *N xx*	-	off	E	Set an input-line count trap.
.em *xx*	none	none	-	End macro is *xx*.

8. Number Registers

.nr *R* ±*N M*		-	**u**	Define and set number register *R*; auto-increment by *M*.
.af *R c*	arabic	-	-	Assign format to register *R* (*c*=1, i, I, a, A).
.rr *R*	-	-	-	Remove register *R*.

9. Tabs, Leaders, and Fields

.ta *Nt* ...	0.5i; 0.8n	none	E,**m**	Tab settings; left-adjusting, unless *t*=R (right), C (centered).
.tc *c*	none	none	E	Tab repetition character.
.lc *c*	.	none	E	Leader repetition character.
.fc *a b*	off	off	-	Set field delimiter *a* and pad character *b*.

10. Input and Output Conventions and Character Translations

.ec *c*	\	\	-	Set escape character.
.eo	on	-	-	Turn off escape character mechanism.
.lg *N*	on; -	on	T	Ligature mode on if *N*>0.
.ul *N*	off	*N*=1	E	Underline (italicize in *troff*) *N* input lines.
.cu *N*	off	*N*=1	E	Continuous underline in *nroff* ; in *troff* , like ul .
.uf *F*	Italic	Italic	-	Underline font set to *F* (to be switched to by ul).
.cc *c*	.	.	E	Set control character to *c*.
.c2 *c*	'	'	E	Set no-break control character to *c*.
.tr *abcd....*	none	-	O	Translate *a* to *b*, etc. on output.

11. Local Horizontal and Vertical Motions, and the Width Function

12. Overstrike, Bracket, Line-drawing, Graphics, and Zero-width Functions

13. Hyphenation.

.nh	hyphenate	-	E	No hyphenation.
.hy *N*	hyphenate	hyphenate	E	Hyphenate; *N* = mode.
.hc *c*	\%	\%	E	Hyphenation indicator character *c*.
.hw *word* ...		ignored	-	Add words to hyphenation dictionary.

14. Three-Part Titles.

.tl *'l'c'r'*		-	-	Three-part title; delimiter may be any character.
.pc *c*	%	off	-	Page number character.
.lt ±*N*	6.5i	previous	E,**m**	Length of title.

15. Output Line Numbering.

.nm ±*N M S I*		off	E	Number mode on or off, set parameters.
.nn *N*	-	*N*=1	E	Do not number next *N* lines.

16. **Conditional Acceptance of Input**

.if *c any*		–	–	If condition *c* true, accept *any* as input; for multi-line use \{ *any* \}.
.if !*c any*		–	–	If condition *c* false, accept *any*.
.if *N any*		–	**u**	If expression $N > 0$, accept *any*.
.if !*N any*		–	**u**	If expression *N* f 0 [sic], accept *any*.
.if ′*s*1′*s*2′ *any*		–	–	If string *s*1 identical to *s*2, accept *any*.
.if !′*s*1′*s*2′ *any*		–	–	If string *s*1 not identical to *s*2, accept *any*.
.ie *c any*		–	**u**	If portion of if-else; all above forms (like if).
.el *any*		–	–	Else portion of if-else.

17. **Environment Switching**

.ev *N*	*N*=0	previous	–	Environment switch (push down).

18. **Insertions from the Standard Input**

.rd *prompt*	–	*prompt*=BEL	–	Read insertion.
.ex	–	–	–	Exit.

19. **Input/Output File Switching**

.so *filename*		–	–	Switch source file (push down).
.nx *filename*		end-of-file	–	Next file.
.sy *string*		–	–	Execute program *string*. Output not interpolated.
.pi *string*		–	–	Pipe output to program *string*.
.cf *filename*		–	–	Copy file contents to *troff* output.

20. **Miscellaneous**

.mc *c N*	–	off	E,**m**	Set margin character *c* and separation *N*.
.tm *string*	–	newline	–	Print *string* on terminal (standard error).
.ab *string*	–	newline	–	Print *string* on standard error, exit program.
.ig *yy*	–	.*yy*=..	–	Ignore input until call of *yy*.
.lf *N f*		–	–	Set input line number to *N* and filename to *f*.
.pm *t*	–	all	–	Print macro names, sizes; if *t* present, print total.
.fl	–	–	B	Flush output buffer.

21. **Output and Error Messages**

22. **Output Language**

23. **Device and Font Description Files**

Alphabetical Request and Section Number Cross Reference

ab	20	ce	4	ec	10	ft	2	lg	10	nh	13	pm	20	so	19	tr	10
ad	4	cf	19	el	16	hc	13	lf	20	nm	15	pn	3	sp	5	uf	10
af	8	ch	7	em	7	hw	13	ll	6	nn	15	po	3	ss	2	ul	10
am	7	cs	2	eo	10	hy	13	ls	5	nr	8	ps	2	sv	5	vs	5
as	7	cu	10	ev	17	ie	16	lt	14	ns	5	rd	18	sy	19	wh	7
bd	2	da	7	ex	18	if	16	mc	20	nx	19	rm	7	ta	9		
bp	3	de	7	fc	9	ig	20	mk	3	os	5	rn	7	tc	9		
br	4	di	7	fi	4	in	6	na	4	pc	14	rr	8	ti	6		
c2	10	ds	7	fl	20	it	7	ne	3	pi	19	rs	5	tl	14		
cc	10	dt	7	fp	2	lc	9	nf	4	pl	3	rt	3	tm	20		

Escape Sequences for Characters, Indicators, and Functions

Section Reference	*Escape Sequence*	*Meaning*
10.1	\\	\ prevents or delays the interpretation of \
10.1	\e	Printable version of the current escape character.
2.1	\′	´ (acute accent); equivalent to \(aa
2.1	\‘	‘ (grave accent); equivalent to \(ga
2.1	\	Minus sign in the current font
7.	\.	Period (dot) (see de)
11.1	\ *space*	Unpaddable space-size space character
11.1	\0	Digit width space
11.1	|	1/6 em narrow space character (zero width in *nroff*)
11.1	\^	1/12 em half-narrow space character (zero width in *nroff*)
4.1	\&	Non-printing, zero width character
10.6	\!	Transparent line indicator
10.8	\"	Beginning of comment; continues to end of line
13.	\%	Default optional hyphenation character
2.1	\(*xx*	Character named *xx*
7.1	* *x*, *(*xx*	Interpolate string *x* or *xx*
7.3	\$ *N*	Interpolate argument 1 ≤ *N* ≤ 9
9.1	\a	Non-interpreted leader character
12.3	\b′ *abc...*′	Bracket building function
4.2	\c	Connect to next input text
2.1	\C′ *xyz*′	Character named *xyz*
11.1	\d	Downward 1/2 em vertical motion (1/2 line in *nroff*)
12.5	\D′ *c...*′	Draw graphics function *c* with parameters ...; *c* = l , c , e , a , ~
2.2	\f *x*, \f(*xx*, \f *N*	Change to font named *x* or *xx*, or position *N*
8.	\g *x*, \g(*xx*	Format of number register *x* or *xx*
11.1	\h′ *N*′	Local horizontal motion; move right *N* (negative left)
2.3	\H′ *N*′	Height of current font is *N*
11.3	\k *x*	Mark horizontal input place in register *x*
12.4	\l′ *Nc*′	Horizontal line drawing function (optionally with *c*)
12.4	\L′ *Nc*′	Vertical line drawing function (optionally with *c*)
8.	\n *x*, \n(*xx*	Contents of number register *x* or *xx*
2.1	\N′ *N*′	Character number *N* on current font
12.1	\o′ *abc...*′	Overstrike characters *a*, *b*, *c*, ...
4.1	\p	Break and spread output line
11.1	\r	Reverse 1 em vertical motion (reverse line in *nroff*)
2.3	\s *N*, \s ±*N*	Point-size change function; also \s(*nn*, \s ±(*nn*
2.2	\S′ *N*′	Slant output *N* degrees
9.1	\t	Non-interpreted horizontal tab
11.1	\u	Reverse (up) 1/2 em vertical motion (1/2 line in *nroff*)
11.1	\v′ *N*′	Local vertical motion; move down N (negative up)
11.2	\w′ *string* ′	Width of *string*
5.2	\x′ *N*′	Extra line-space function (negative before, positive after)
10.7	\X′ *string* ′	Output *string* as device control function
12.2	\z *c*	Print *c* with zero width (without spacing)
16.	\{	Begin conditional input
16.	\}	End conditional input
10.8	\ *newline*	Concealed (ignored) newline
-	\ *Z*	*Z*, any character not listed above

The escape sequences \\ , \. , \" , \$, * , \a , \n , \t , \g , and \ *newline* are interpreted in copy mode (§7.2).

Predefined Number Registers

Section Reference	*Register Name*	*Description*
3.	%	Current page number.
11.2	ct	Character type (set by \w function).
7.4	dl	Width (maximum) of last completed diversion.
7.4	dn	Height (vertical size) of last completed diversion.
-	dw	Current day of the week (1-7).
-	dy	Current day of the month (1-31).
15.	ln	Output line number.
-	mo	Current month (1-12).
4.1	nl	Vertical position of last printed text base line.
11.2	sb	Depth of string below base line (generated by \w function).
11.2	st	Height of string above base line (generated by \w function).
-	yr	Last two digits of current year.

Predefined Read-Only Number Registers

Section Reference	*Register Name*	*Description*
19.	$$	Process id of *troff* or *nroff*.
7.3	.$	Number of arguments available at the current macro level.
5.2	.a	Post-line extra line-space most recently used in \x'*N*'.
-	.A	Set to 1 in *troff*, if -a option used; always 1 in *nroff*.
2.3	.b	Emboldening level.
20.	.c	Number of lines read from current input file.
7.4	.d	Current vertical place in current diversion; equal to nl, if no diversion.
2.2	.f	Current font number.
20.	.F	Current input file name [sic].
4.	.h	Text base line high-water mark on current page or diversion.
11.1	.H	Available horizontal resolution in basic units.
6.	.i	Current indent.
4.2	.j	Current ad mode.
4.1	.k	Current output horizontal position.
6.	.l	Current line length.
5.1	.L	Current ls value.
4.	.n	Length of text portion on previous output line.
3.	.o	Current page offset.
3.	.p	Current page length.
7.5	.R	Number of unused number registers.
-	.T	Set to 1 in *nroff*, if -T option used; always 0 in *troff*.
2.3	.s	Current point size.
7.5	.t	Distance to the next trap.
4.1	.u	Equal to 1 in fill mode and 0 in nofill mode.
5.1	.v	Current vertical line spacing.
11.1	.V	Available vertical resolution in basic units.
11.2	.w	Width of previous character.
-	.x	Reserved version-dependent register.
-	.y	Reserved version-dependent register.
7.4	.z	Name [sic] of current diversion.

Reference Manual

1. General Explanation

1.1. Form of input. Input consists of *text lines*, which are destined to be printed, interspersed with *control lines*, which set parameters or otherwise control subsequent processing. Control lines begin with a *control character* normally . (period) or ' (single quote) followed by a one or two character name that specifies a basic *request* or the substitution of a user-defined *macro* in place of the control line. The control character ' suppresses the *break* function the forced output of a partially filled line caused by certain requests. The control character may be separated from the request/macro name by white space (spaces and/or tabs) for aesthetic reasons. Names should be followed by either space or newline. Control lines with unrecognized names are ignored.

Various special functions may be introduced anywhere in the input by means of an *escape* character, normally \. For example, the function \n*R* causes the interpolation of the contents of the *number register R* in place of the function; here *R* is either a single character name as in \n*x*, or a two-character name introduced by a left-parenthesis, as in \n(*xx*.

1.2. Formatter and device resolution. *Troff* internally stores and processes dimensions in units that correspond to the particular device for which output is being prepared; values from 300 to 1200/inch are typical. See §23. *Nroff* internally uses 240 units/inch, corresponding to the least common multiple of the horizontal and vertical resolutions of various typewriter-like output devices. *Troff* rounds horizontal/vertical numerical parameter input to the actual horizontal/vertical resolution of the output device indicated by the -T option (default post). *Nroff* similarly rounds numerical input to the actual resolution of its output device (default Model 37 Teletype).

1.3. Numerical parameter input. Both *nroff* and *troff* accept numerical input with the appended scale indicators shown in the following table, where *S* is the current type size in points and *V* is the current vertical line spacing in basic units.

Scale Indicator	Meaning
i	Inch
c	Centimeter
P	Pica = 1/6 inch
m	Em = *S* points
n	En = Em/2
p	Point = 1/72 inch
u	Basic unit
v	Vertical line space *V*
none	Default, see below

In *nroff*, both the em and the en are taken to be equal to the nominal character width, which is output-device dependent; common values are 1/10 and 1/12 inch. Actual character widths in *nroff* need not be all the same and constructed characters such as -> (´) are often extra wide. The default scaling is m for the horizontally-oriented requests and functions ll, in, ti, ta, lt, po, mc, \h, \l, and horizontal coordinates of \D; v for the vertically-oriented requests and functions pl, wh, ch, dt, sp, sv, ne, rt, \v, \x, \L, and vertical coordinates of \D; p for the vs request; and u for the requests nr, if, and ie. *All* other requests ignore any scale indicators. When a number register containing an already appropriately scaled number is interpolated to provide numerical input, the unit scale indicator u may need to be appended to prevent an additional inappropriate default scaling. The number, *N*, may be specified in decimal-fraction form but the parameter finally stored is rounded to an integer number of basic units. Internal computations are performed in integer arithmetic.

The *absolute position* indicator | may be prefixed to a number *N* to generate the distance to the vertical or horizontal place *N*. For vertically-oriented requests and functions, |*N* becomes the distance in basic units from the current vertical place on the page or in a *diversion* (§7.4) to the vertical place *N*. For *all* other requests and functions, |*N* becomes the distance from the current horizontal place on the *input* line to the horizontal place *N*. For example,

```
.sp |3.2c
```

will space in the required direction to 3.2 centimeters from the top of the page.

1.4. Numerical expressions. Wherever numerical input is expected, an expression involving parentheses, the arithmetic operators +, −, /, *, % (mod), and the logical operators <, >, <=, >=, = (or ==), & (and), : (or) may be used. Except where controlled by parentheses, evaluation of expressions is left-to-right; there is no operator precedence. In the case of certain requests, an initial + or − is stripped and interpreted as an increment or decrement indicator respectively. In the presence of default scaling, the desired scale indicator must be attached to *every* number in an expression for which the desired and default scaling differ. For example, if the number register x contains 2 and the current point size is 10, then

```
.ll (4.25i+\nxP+3)/2u
```

will set the line length to 1/2 the sum of 4.25 inches + 2 picas + 3 ems.

1.5. Notation. Numerical parameters are indicated in this manual in two ways. ±*N* means that the argument may take the forms *N*, +*N*, or −*N* and that the corresponding effect is to set the parameter to *N*, to increment it by *N*, or to decrement it by *N* respectively. Plain *N* means that an initial algebraic sign is *not* an increment indicator, but merely the sign of *N*. Generally, unreasonable numerical input is either ignored or truncated to a reasonable value. For example, most requests expect to set parameters to non-negative values; exceptions are sp, wh, ch, nr, and if. The requests ps, ft, po, vs, ls, ll, in, and lt restore the previous parameter value in the absence of an argument.

Single character arguments are indicated by single lower case letters and one/two character arguments are indicated by a pair of lower case letters. Character string arguments are indicated by multi-character mnemonics.

2. Font and Character Size Control

2.1. Character set. The *troff* character set is defined by a description file specific to each output device (§23). There are normally several regular fonts and one or more special fonts. Characters are input as themselves, as \(*xx*, as \C'*name*', or as \N'*n*'. The form \C'*name*' permits a name of any length; the form \N'*n*' refers to the *n*-th character on the current font, whether named or not.

Normally the input characters `, ', and - are printed as ‘, ’, and - respectively; \`, \', and \- produce `, ´, and −. If the character does not exist in the font, *troff* assumes the width is 1 em and outputs the character with a C name as defined in Section 22. (This is independent of how the device handles characters unknown to it.)

Nroff has an analogous, but different, mechanism for defining legal characters and how to print them. By default all characters are valid. There are such additional characters as may be available on the output device, such characters as may be constructed by overstriking or other combination, and those that can reasonably be mapped into other printable characters. The exact behavior is determined by a driving table prepared for each device.

2.2. Fonts. *Troff* begins execution by reading information for a set of defaults fonts, said to be *mounted*; conventionally, the first four are Times Roman (R), Times Italic (I), Times Bold (B), and Times Bold Italic (BI), and the last is a Special font (S) containing miscellaneous characters. (This document uses Lucida Sans in place of Times.) The set of fonts and positions is determined by the device description file, described in §23.

The current font, initially Roman, may be changed by the ft request, or by embedding at any desired point \f*x*, \f(*xx*, or \f*N*, where *x* and *xx* are the name of a font and *N* is a numerical font position.

It is not necessary to change to the Special font; characters on that font are automatically handled as if they were physically part of the current font. The Special font may actually be several fonts; the name S is reserved and is generally used for one of these. All special fonts must be mounted after regular fonts.

Troff can be informed that any particular font is mounted by use of the fp request. The list of known fonts is installation dependent. In the subsequent discussion of font-related requests, *F* represents either a one/two-character font name or the numerical font position. The current font is available (as a numerical position) in the read-only number register .f.

A request for a named but not-mounted font is honored if the font description information exists. In this way, there is no limit on the number of fonts that may be printed in any part of a document. Mounted fonts may be handled more efficiently, and they may be referred to by their mount positions, but there is no other difference. Mention of an unmounted font loads it temporarily at font position zero, which serves as a one-font cache.

The function \S'±*N*' causes the current font to be slanted by ±*N* degrees. Not all devices support slanting.

Nroff understands font control and normally underlines italic characters (see §10.5).

2.3. Character size. Character point sizes available depend on the specific output device; a typical (historical) set of values is 6, 7, 8, 9, 10, 11, 12, 14, 16, 18, 20, 22, 24, 28, and 36. This is a range of 1/12 inch to 1/2 inch. The ps request is used to change or restore the point size. Alternatively the point size may be changed between any two characters by embedding a \s*N* at the desired point to set the size to *N*, or a \s±*N* (1 ≤ *N* ≤ 9) to increment/decrement the size by *N*; \s0 restores the previous size. Requested point size values that are between two valid sizes yield the larger of the two.

Note that through an accident of history, a construction like \s39 is parsed as size 39, and thus converted to size 36 (given the sizes above), while \s40 is parsed as size 4 followed by 0. The forms \s(*nn* and \s±(*nn* permit specification of sizes that would otherwise be ambiguous.

The current size is available in the .s register. *Nroff* ignores type size requests.

The function \H'±*N*' sets the height of the current font to *N*, or increments it by +*N*, or decrements it by −*N*; if *N*=0, the height is restored to the current point size. In each case, the width is unchanged. Not all devices support independent height and width for characters.

Request Form	*Initial Value*	*If No Argument*	*Notes*
.ps ±*N**	10 point	previous	E

Point size set to ±*N*. Alternatively embed \s*N* or \s±*N*. Any positive size value may be requested; if invalid, the next larger valid size will result, with a maximum of 36. A paired sequence +*N*, −*N* will work because the

*The fields have the same meaning as described earlier in the Request Summary.

previous requested value is also remembered. Ignored in *nroff*.

.ss *N* 12/36 em ignored E

Space-character size (i.e., inter-word gap) is set to *N*/36 ems. This size is the minimum word spacing in adjusted text. Ignored in *nroff*.

.cs *F N M* off - P

Constant character space (width) mode is set on for font *F* (if mounted); the width of every character will be taken to be *N*/36 ems. If *M* is absent, the em is that of the character's point size; if *M* is given, the em is *M* points. All affected characters are centered in this space, including those with an actual width larger than this space. Special Font characters occurring while the current font is *F* are also so treated. If *N* is absent, the mode is turned off. The mode must be in effect when the characters are physically printed. Ignored in *nroff*.

.bd *F N* off - P

The characters in font *F* will be artificially emboldened by printing each one twice, separated by $N-1$ basic units. A reasonable value for *N* is 3 when the character size is near 10 points. If *N* is missing the embolden mode is turned off. The emboldening value *N* is in the .b register.

This paragraph is printed with .bd R 3. The mode must be in effect when the characters are physically printed. Ignored in *nroff*.

.bd S *F N* off - P

The characters in the Special font will be emboldened whenever the current font is *F*. The mode must be in effect when the characters are physically printed. Ignored in *nroff*.

.ft *F* Roman previous E

Font changed to *F*. Alternatively, embed \f*F*. The font name P is reserved to mean the previous font, and the name S for the special font.

.fp *N F L* R,I,B,...,S ignored -

Font position. This is a statement that a font named *F* is associated with position *N*. It is a fatal error if *F* is not known. For fonts with names longer than two characters, *L* refers to the long name, and *F* becomes a synonym. There is generally a limit of about 10 mounted fonts.

3. Page control

Top and bottom margins are not automatically provided; it is conventional to define two *macros* and to set *traps* for them at vertical positions 0 (top) and $-N$ (distance *N* up from the bottom). See §7 and Tutorial Examples §T2. A pseudo-page transition onto the first page occurs either when the first *break* occurs or when the first *non-diverted* text processing occurs. Arrangements for a trap to occur at the top of the first page must be completed before this transition. In the following, references to the *current diversion* (§7.4) mean that the mechanism being described works during both ordinary and diverted output (the former considered as the top diversion level).

The limitations on *troff* and *nroff* output dimensions are device dependent.

.pl ±*N* 11 in 11 in **v**

Page length set to ±*N*. The current page length is available in the .p register.

.bp ±*N* *N*=1 - B,**v**

Begin page. The current page is ejected and a new page is begun. If ±*N* is given, the new page number will be ±*N*. Also see request ns.

.pn ±*N*   *N*=1   ignored   -

Page number. The next page (when it occurs) will have the page number ±*N*. A pn must occur before the initial pseudo-page transition to affect the page number of the first page. The current page number is in the % register.

.po ±*N*   1 in; 0   previous   **v**

Page offset. The current *left margin* is set to ±*N*. The *troff* initial value provides 1 inch of paper margin on a typical device. The current page offset is available in the .o register.

.ne *N*   -   *N*=1 *V*   D,**v**

Need *N* vertical space. If the distance *D* to the next trap position (see §7.5) is less than *N*, a forward vertical space of size *D* occurs, which will spring the trap. If there are no remaining traps on the page, *D* is the distance to the bottom of the page. If $D < V$, another line could still be output and spring the trap. In a diversion, *D* is the distance to the *diversion trap*, if any, or is very large.

.mk *R*   none   internal   D

Mark the current vertical place in an internal register (both associated with the current diversion level), or in register *R*, if given. See rt request.

.rt ±*N*   none   internal   D,**v**

Return *upward only* to a marked vertical place in the current diversion. If ±*N* (with respect to current place) is given, the place is ±*N* from the top of the page or diversion or, if *N* is absent, to a place marked by a previous mk. The sp request (§5.3) may be used instead of rt by spacing to the absolute place stored in a explicit register, e.g., using .mk *R* ... .sp |\n*R*u; this also works when the motion is downwards.

4. Text Filling, Adjusting, and Centering

4.1. Filling and adjusting. Normally, words are collected from input text lines and assembled into a output text line until some word does not fit. An attempt is then made to hyphenate the word to put part of it into the output line. The spaces between the words on the output line are then increased to spread out the line to the current *line length* minus any current *indent*. A *word* is any string of characters delimited by the *space* character or the beginning/end of the input line. Any adjacent pair of words that must be kept together (neither split across output lines nor spread apart in the adjustment process) can be tied together by separating them with the *unpaddable space* character \ (backslash-space). The adjusted word spacings are uniform in *troff* and the minimum interword spacing can be controlled with the ss request (§2). In *nroff*, they are normally nonuniform because of quantization to character-size spaces; however, the command line option −e causes uniform spacing with full output device resolution. Filling, adjustment, and hyphenation (§13) can all be prevented or controlled. The text length on the last line output is available in the .n register, and text baseline position on the page for this line is in the nl register. The text baseline high-water mark (lowest place) on the current page is in the .h register. The current horizontal output position is in the .k register.

An input text line *ending* with ., ?, or !, optionally followed by any number of ", ',),], *, or †, is taken to be the end of a sentence, and an additional space character is automatically provided during filling. To prevent this, add \& to the end of the input line. Multiple inter-word space characters found in the input are retained, except for trailing spaces; initial spaces also cause a break.

When filling is in effect, a \p may be embedded or attached to a word to cause a break at the end of the word and have the resulting output line spread out to fill the current line length.

A text input line that happens to begin with a control character can be made not to look like a control line by prefixing it with the non-printing, zero-width filler character \&. Still another way is to specify output translation of some convenient character into the control character using tr (§10.5).

4.2. Interrupted text. The copying of a input line in *nofill* (non-fill) mode can be interrupted by terminating the partial line with a \c. The next encountered input text line will be considered to be a continuation of the same line of input text. Similarly, a word within *filled* text may be interrupted by terminating the word (and line) with \c; the next encountered text will be taken as a continuation of the interrupted word. If the intervening control lines cause a break, any partial line will be forced out along with any partial word.

.br - - B

Break. The filling of the line currently being collected is stopped and the line is output without adjustment. Text lines beginning with space characters (but not tabs) and empty text lines (blank lines) also cause a break.

.fi fill on - B,E

Fill subsequent output lines. The register .u is 1 in fill mode and 0 in nofill mode.

.nf fill on - B,E

Nofill. Subsequent output lines are neither filled nor adjusted. Input text lines are copied directly to output lines without regard for the current line length.

.ad *c* adj, both adjust E

Line adjustment is begun. If fill mode is not on, adjustment will be deferred until fill mode is back on. If the type indicator *c* is present, the adjustment type is changed as shown in the following table.

Indicator	Adjust Type
l	adjust left margin only
r	adjust right margin only
c	center
b or n	adjust both margins
absent	unchanged

The number register .j contains the current value of the ad setting; its value can be recorded and used subsequently to set adjustment.

.na adjust - E

Noadjust. Adjustment is turned off; the right margin will be ragged. The adjustment type for ad is not changed. Output line filling still occurs if fill mode is on.

.ce *N* off *N*=1 B,E

Center the next *N* input text lines within the current available horizontal space (line-length minus indent). If *N*=0, any residual count is cleared. A break occurs after each of the *N* input lines. If the input line is too long, it will be left adjusted.

5. Vertical Spacing

5.1. Baseline spacing. The vertical spacing (V) between the baselines of successive output lines can be set using the vs request. V should be large enough to accommodate the character sizes on the affected output lines. For the common type sizes (9-12 points), usual typesetting practice is to set V to 2 points greater than the point size; *troff* default is 10-point type on a 12-point spacing (as in this document). The current V is available in the .v register. Multiple-V line separation (e.g. double spacing) may be requested with ls, but it is better to use a large vs instead; certain preprocessors assume single spacing. The current line spacing is available in the .L register.

5.2. Extra line-space. If a word contains a tall construct requiring the output line containing it to have extra vertical space before and/or after it, the *extra-line-space* function \x'N' can be embedded in or attached to that word. If N is negative, the output line containing the word will be preceded by N extra vertical space; if N is positive, the output line containing the word will be followed by N extra vertical space. If successive requests for extra space apply to the same line, the maximum values are used. The most recently utilized post-line extra line-space is available in the .a register.

In \x'...' and other functions having a pair of delimiters around their parameter, the delimiter choice (here ') is arbitrary, except that it can not look like the continuation of a number expression for N.

5.3. Blocks of vertical space. A block of vertical space is ordinarily requested using sp, which honors the *no-space* mode and which does not space past a trap. A contiguous block of vertical space may be reserved using sv.

.vs N — 12pts; 1/6in — previous — E,**p**

Set vertical baseline spacing size V. Transient extra vertical space is available with \x'N' (see above).

.ls N — N=1 — previous — E

Line spacing set to $\pm N$. $N-1$ Vs (blank lines) are appended to each output text line. Appended blank lines are omitted, if the text or previous appended blank line reached a trap position.

.sp N — - — N=1 V — B,**v**

Space vertically in either direction. If N is negative, the motion is backward (upward) and is limited to the distance to the top of the page. Forward (downward) motion is truncated to the distance to the nearest trap. (Recall the use of .sp |N from §1.3.) If the no-space mode is on, no spacing occurs (see ns and rs below).

.sv N — - — N=1 V — **v**

Save a contiguous vertical block of size N. If the distance to the next trap is greater than N, N vertical space is output. No-space mode has no effect. If this distance is less than N, no vertical space is immediately output, but N is remembered for later output (see os). Subsequent sv requests will overwrite any still remembered N.

.os — - — - — -

Output saved vertical space. No-space mode has no effect. Used to finally output a block of vertical space requested by an earlier sv request.

.ns — space — - — D

No-space mode turned on. When on, no-space mode inhibits sp requests and bp requests *without* a next page number. No-space mode is turned off when a line of output occurs, or with rs.

.rs — space — - — D

Restore spacing. The no-space mode is turned off.

Blank text line.   -   B

Causes a break and output of a blank line exactly like .sp 1.

6. Line Length and Indenting

The maximum line length for fill mode may be set with ll . The indent may be set with in ; an indent applicable to only the next output line may be set with ti . The line length includes indent space but not page offset space. The line length minus the indent is the basis for centering with ce . The effect of ll , in , or ti is delayed, if a partially collected line exists, until after that line is output. In fill mode the length of text on an output line is less than or equal to the line length minus the indent. The current line length and indent are available in registers .l and .i respectively. The length of *three-part titles* produced by tl (see §14) is independently set by lt.

.ll ±*N*   6.5in   previous   E,m

Line length is set to ±*N*.

.in ±*N*   *N*=0   previous   B,E,m

Indent is set to ±*N*. The indent is prefixed to each output line.

.ti ±*N*   -   ignored   B,E,m

Temporary indent. The next output text line will be indented a distance ±*N* with respect to the current indent. The resulting total indent may not be negative. The current indent is not changed.

7. Macros, Strings, Diversion, and Position Traps

7.1. Macros and strings. A *macro* is a named set of arbitrary *lines* that may be invoked by name or with a *trap* . A *string* is a named string of *characters* , not including a newline character, that may be interpolated by name at any point. Request, macro, and string names share the same name list. Macro and string names may be one or two characters long and may usurp previously defined request, macro, or string names; this implies that built-in operations may be (irrevocably) redefined. Any of these entities may be renamed with rn or removed with rm .

Macros are created by de and di , and appended to by am and da ; di and da cause normal output to be stored in a macro. A macro is invoked in the same way as a request; a control line beginning .*xx* will interpolate the contents of macro *xx*. The remainder of the line may contain up to nine *arguments* .

Strings are created by ds and appended to by as . The strings *x* and *xx* are interpolated at any desired point with *x and *(*xx* respectively. String references and macro invocations may be nested.

7.2. Copy mode input interpretation. During the definition and extension of strings and macros (not by diversion) the input is read in *copy mode* . In copy mode, input is copied without interpretation except that:

The contents of number registers indicated by \n are interpolated.
Strings indicated by * are interpolated.
Arguments indicated by \$ are interpolated.
Concealed newlines indicated by \\(*newline*) are eliminated.
Comments indicated by \" are eliminated.
\t and \a are interpreted as ASCII horizontal tab and SOH respectively (§9).
\\ is interpreted as \.
\. is interpreted as ".".

These interpretations can be suppressed by prefixing a \ . For example, since \\ maps into a \ , \\n will copy as \n , which will be interpreted as a number register indicator when the macro or string is reread.

7.3. Arguments. When a macro is invoked by name, the remainder of the line is taken to contain up to nine arguments. The argument separator is the space character (not tab), and arguments may be surrounded by double quotes to permit embedded space characters. Pairs of double quotes may be embedded in double-quoted arguments to represent a single double-quote character. The argument "" is explicitly null. If the desired arguments won't fit on a line, a concealed newline may be used to continue on the next line. A trailing double quote may be omitted.

When a macro is invoked the *input level* is *pushed down* and any arguments available at the previous level become unavailable until the macro is completely read and the previous level is restored. A macro's own arguments can be interpolated at any point within the macro with \$*N*, which interpolates the *N*th argument ($1 \le N \le 9$). If an invoked argument does not exist, a null string results. For example, the macro *xx* may be defined by

```
.de xx   \" begin definition
Today is \\$1 the \\$2.
..       \" end definition
```

and called by

```
.xx Monday 14th
```

to produce the text

```
Today is Monday the 14th.
```

Note that each \$ was concealed in the definition with a prefixed \. The number of arguments is in the .$ register.

No arguments are available at the top (non-macro) level, within a string, or within a trap-invoked macro.

Arguments are copied in copy mode onto a stack where they are available for reference. It is advisable to conceal string references (with an extra \) to delay interpolation until argument reference time.

7.4. Diversions. Processed output may be diverted into a macro for purposes such as footnote processing (see Tutorial §T5) or determining the horizontal and vertical size of some text for conditional changing of pages or columns. A single diversion trap may be set at a specified vertical position. The number registers dn and dl respectively contain the vertical and horizontal size of the most recently ended diversion. Processed text that is diverted into a macro retains the vertical size of each of its lines when reread in *nofill* mode regardless of the current *V*. Constant-spaced (cs) or emboldened (bd) text that is diverted can be reread correctly only if these modes are again or still in effect at reread time. One way to do this is to embed in the diversion the appropriate cs or bd requests with the *transparent* mechanism described in §10.6.

Diversions may be nested and certain parameters and registers are associated with the current diversion level (the top non-diversion level may be thought of as the 0th diversion level). These are the diversion trap and associated macro, no-space mode, the internally-saved marked place (see mk and rt), the current vertical place (.d register), the current high-water text base-line (.h register), and the current diversion name (.z register).

7.5. Traps. Three types of trap mechanisms are available—page traps, a diversion trap, and an input-line-count trap. Macro-invocation traps may be planted using wh at any page position including the top. This trap position may be changed using ch. Trap positions at or below the bottom of the page have no effect unless or until moved to within the page or rendered effective by an increase in page length. Two traps may be planted at the same position only by first planting them at different positions and then moving one of the traps; the first planted trap will conceal the second unless and until the first one is moved (see Tutorial Examples). If the first one is moved back, it again conceals the second trap. The macro associated with a page trap is automatically invoked when a line of text is output whose vertical size reaches or sweeps past the trap position. Reaching the bottom of a page springs the top-of-page trap, if any, provided there is a next page. The distance to the next trap position is available in the .t register; if there are no traps between the current position and the bottom of the page, the distance returned is the distance to the page bottom.

A macro-invocation trap effective in the current diversion may be planted using dt. The .t register works in a diversion; if there is no subsequent trap a large distance is returned. For a description of input-line-count traps, see it below.

.de *xx yy* - *.yy=..* -

Define or redefine the macro *xx*. The contents of the macro begin on the next input line. Input lines are copied in *copy mode* until the definition is terminated by a line beginning with .*yy*, whereupon the macro *yy* is called. In the absence of *yy*, the definition is terminated by a line beginning with .. . A macro may contain de requests provided the terminating macros differ or the contained definition terminator is concealed. .. can be concealed as \\.. which will copy as \.. and be reread as .. .

.am *xx yy* - *.yy=..* -

Append to macro *xx* (append version of de).

.ds *xx string* - ignored -

Define a string *xx* containing *string*. Any initial double quote in *string* is stripped off to permit initial blanks.

.as *xx string* - ignored -

Append *string* to string *xx* (append version of ds).

.rm *xx* - ignored -

Remove request, macro, or string. The name *xx* is removed from the name list and any related storage space is freed. Subsequent references will have no effect. If many macros and strings are being created dynamically, it may become necessary to remove unused ones to recapture internal storage space for newer registers.

.rn *xx yy* - ignored -

Rename request, macro, or string *xx* to *yy*. If *yy* exists, it is first removed.

.di *xx* - end D

Divert output to macro *xx*. Normal text processing occurs during diversion except that page offsetting is not done. The diversion ends when the request di or da is encountered without an argument; extraneous requests of this type should not appear when nested diversions are being used.

.da *xx* - end D

Divert, appending to macro *xx* (append version of di).

.wh *N xx* - - **v**

Install a trap to invoke *xx* at page position *N*; a negative N will be interpreted as a distance from the page bottom. Any macro previously planted at *N* is replaced by *xx*. A zero *N* refers to the top of a page. In the absence of *xx*, the first trap found at *N*, if any, is removed.

.ch *xx N* - - **v**

Change the trap position for macro *xx* to be *N*. In the absence of *N*, the trap, if any, is removed.

.dt *N xx*   -   off   D,**v**

Install a diversion trap at position *N* in the *current* diversion to invoke macro *xx*. Another dt will redefine the diversion trap. If no arguments are given, the diversion trap is removed.

.it *N xx*   -   off   E

Set an input-line-count trap to invoke the macro *xx* after *N* lines of *text* input have been read (control or request lines do not count). The text may be inline text or text interpolated by inline or trap-invoked macros.

.em *xx*   none   none   -

The macro *xx* will be invoked when all input has ended. The effect is almost as if the contents of *xx* had been at the end of the last file processed, but all processing ceases at the next page eject.

8. Number Registers

A variety of parameters are available to the user as predefined *number registers* (see Summary, page 7). In addition, users may define their own registers. Register names are one or two characters long and do not conflict with request, macro, or string names. Except for certain predefined read-only registers, a number register can be read, written, automatically incremented or decremented, and interpolated into the input in a variety of formats. One common use of user-defined registers is to automatically number sections, paragraphs, lines, etc. A number register may be used any time numerical input is expected or desired and may be used in numerical *expressions* (§1.4).

Number registers are created and modified using nr, which specifies the name, numerical value, and the auto-increment size. Registers are also modified, if accessed with an auto-incrementing sequence. If the registers *x* and *xx* both contain *N* and have the auto-increment size *M*, the following access sequences have the effect shown:

Sequence	Effect on Register	Value Interpolated
\n*x*	none	*N*
\n(*xx*	none	*N*
\n+*x*	*x* incremented by *M*	*N+M*
\n−*x*	*x* decremented by *M*	*N−M*
\n+(*xx*	*xx* incremented by *M*	*N+M*
\n−(*xx*	*xx* decremented by *M*	*N−M*

When interpolated, a number register is converted to decimal (default), decimal with leading zeros, lower-case Roman, upper-case Roman, lower-case sequential alphabetic, or upper-case sequential alphabetic according to the format specified by af.

.nr *R*±*N M*   -   **u**

The number register *R* is assigned the value ±*N* with respect to the previous value, if any. The increment for auto-incrementing is set to *M*.

.af *R c*   arabic   -   -

Assign format *c* to register *R*. The available formats are:

Format	Numbering Sequence
1	0, 1, 2, 3, 4, 5, ...
001	000, 001, 002, 003, 004, 005, ...
i	0, i, ii, iii, iv, v, ...
I	0, I, II, III, IV, V, ...
a	0, a, b, c, ..., z, aa, ab, ..., zz, aaa, ...
A	0, A, B, C, ..., Z, AA, AB, ..., ZZ, AAA, ...

An arabic format having *N* digits specifies a field width of *N* digits (example 2 above). The read-only registers and the width function \w (§11.2) are always arabic. Warning: the value of a number register in a non-Arabic format is not numeric, and will not produce the expected results in expressions.

The function \g*x* or \g(*xx* returns the format of a number register in a form suitable for af; it returns nothing if the register has not been used.

.rr *R* — ignored —

Remove number register *R*. If many registers are being created dynamically, it may become necessary to remove unused registers to recapture internal storage space for newer registers. The register .R contains the number of number registers still available.

9. Tabs, Leaders, and Fields

9.1. Tabs and leaders. The ASCII horizontal tab character and the ASCII SOH (control-A, hereafter called the *leader* character) can both be used to generate either horizontal motion or a string of repeated characters. The length of the generated entity is governed by internal *tab stops* specifiable with ta . The default difference is that tabs generate motion and leaders generate a string of periods; tc and lc offer the choice of repeated character or motion. There are three types of internal tab stops *left* adjusting, *right* adjusting, and *centering* . In the following table, *D* is the distance from the current position on the *input* line (where a tab or leader was found) to the next tab stop, *next-string* consists of the input characters following the tab (or leader) up to the next tab (or leader) or end of line, and *W* is the width of *next-string* .

Tab type	Length of motion or repeated characters	Location of *next-string*
Left	*D*	Following *D*
Right	*D−W*	Right adjusted within *D*
Centered	*D−W*/2	Centered on right end of *D*

The length of generated motion is allowed to be negative, but that of a repeated character string cannot be. Repeated character strings contain an integer number of characters, and any residual distance is prepended as motion. Tabs or leaders found after the last tab stop are ignored, but may be used as *next-string* terminators.

Tabs and leaders are not interpreted in copy mode. \t and \a always generate a non-interpreted tab and leader respectively, and are equivalent to actual tabs and leaders in copy mode.

9.2. Fields. A *field* is contained between a pair of *field delimiter* characters, and consists of substrings separated by *padding* indicator characters. The field length is the distance on the *input* line from the position where the field begins to the next tab stop. The difference between the total length of all the substrings and the field length is incorporated as horizontal padding space that is divided among the indicated padding places. The incorporated padding is allowed to be negative. For example, if the field delimiter is # and the padding indicator is ^, #^*xxx*^*right* # specifies a right-adjusted string with the string *xxx* centered in the remaining space.

.ta *Nt* ...   0.8; 0.5in   none   E,**m**

Set tab stops and types. *t*=R, right adjusting; *t*=C, centering; *t* absent, left adjusting. *Troff* tab stops are preset every 0.5in.; *nroff* every 0.8in. The stop values are separated by spaces, and a value preceded by + is treated as an increment to the previous stop value.

.tc *c*   none   none   E

The tab repetition character becomes *c*, or is removed, thus specifying motion.

.lc *c*   .   none   E

The leader repetition character becomes *c*, or is removed, thus specifying motion.

.fc *a b*   off   off   -

The field delimiter is set to *a*; the padding indicator is set to the space character or to *b*, if given. In the absence of arguments the field mechanism is turned off.

10. Input and Output Conventions and Character Translations

10.1. Input character translations. Ways of inputting the valid character set were discussed in §2.1. The ASCII control characters horizontal tab (§9.1), SOH (§9.1), and backspace (§10.3) are discussed elsewhere. The newline delimits input lines. In addition, STX, ETX, ENQ, ACK, and BEL are accepted, and may be used as delimiters or translated into a graphic with .tr (§10.5). All others are ignored.

The *escape* character \ introduces *escape sequences*, which cause the following character to mean another character, or to indicate some function. A complete list of such sequences is given in the Summary on page 7. The escape character \ should not be confused with the ASCII control character ESC. The escape character \ can be input with the sequence \\. The escape character can be changed with ec, and all that has been said about the default \ becomes true for the new escape character. \e can be used to print whatever the current escape character is. The escape mechanism may be turned off with eo, and restored with ec.

.ec *c*   \   \   -

Set escape character to \, or to *c*, if given.

.eo   on   -   -

Turn escape mechanism off.

10.2. Ligatures. The set of available ligatures is device and font dependent, but is often a subset of **fi**, **fl**, **ff**, **ffi**, and **ffl**. They may be input by \(fi, \(fl, \(ff, \(Fi, and \(Fl respectively. The ligature mode is normally on in *troff*, and automatically invokes ligatures during input.

.lg *N*   on; off   on   -

Ligature mode is turned on if *N* is absent or non-zero, and turned off if *N*=0. If *N*=2, only the two-character ligatures are automatically invoked. Ligature mode is inhibited for request, macro, string, register, or file names, and in copy mode. No effect in *nroff*.

10.3. Backspacing, underlining, overstriking, etc. Unless in copy mode, the ASCII backspace character is replaced by a backward horizontal motion having the width of the space character. Underlining as a form of line-drawing is discussed in §12.4. A generalized overstriking function is described in §12.1.

Nroff automatically underlines characters in the *underline* font, specifiable with uf, normally that on font position 2. In addition to ft and \f*F*, the underline font may be selected by ul and cu. Underlining is restricted to an output-device-dependent subset of reasonable characters.

.ul *N*   off   *N*=1   E

Italicize in *troff* (underline in *nroff*) the next *N* input text lines. Actually, switch to underline font, saving the current font for later restoration; other font changes within the span of a ul will take effect, but the restoration will undo the last change. Output generated by tl (§14) is affected by the font change, but does not decrement *N*. If *N*>1, there is the risk that a trap interpolated macro may provide text lines within the span; environment switching can prevent this.

.cu *N*   off   *N*=1   E

Continuous underline. A variant of ul that causes *every* character to be underlined in *nroff*. Identical to ul in *troff*.

.uf *F*   Italic   Italic   -

Underline font set to *F*. In *nroff*, *F* may not be on position 1.

10.4. Control characters. Both the control character . and the *no-break* control character ′ may be changed. Such a change must be compatible with the design of any macros used in the span of the change, and particularly of any trap-invoked macros.

.cc *c*   .   .   E

The basic control character is set to *c*, or reset to ".".

.c2 *c*   ′   ′   E

The *no-break* control character is set to *c*, or reset to "′".

10.5. Output translation. One character can be made a stand-in for another character using tr. All text processing (e.g., character comparisons) takes place with the input (stand-in) character, which appears to have the width of the final character. The graphic translation occurs at the moment of output (including diversion).

.tr *abcd....*   none   -   O

Translate *a* into *b*, *c* into *d*, etc. If an odd number of characters is given, the last one will be mapped into the space character. To be consistent, a particular translation must stay in effect from *input* to *output* time.

10.6. Transparent throughput. An input line beginning with a \! is read in copy mode and *transparently* output (without the initial \!); the text processor is otherwise unaware of the line's presence. This mechanism may be used to pass control information to a post-processor or to embed control lines in a macro created by a diversion.

10.7. Transparent output The sequence \X′*anything*′ copies *anything* to the output, as a device control function of the form x X *anything* (§22). Escape sequences in *anything* are processed.

10.8. Comments and concealed newlines. An uncomfortably long input line that must stay one line (e.g., a string definition, or nofilled text) can be split into several physical lines by ending all but the last one with the escape \. The sequence *newline* is always ignored, except in a comment. Comments may be embedded at the end of any line by prefacing them with \". The newline at the end of a comment cannot be concealed. A line beginning with \" will appear as a blank line and behave like .sp 1; a comment can be on a line by itself by beginning the line with .\".

11. Local Horizontal and Vertical Motions, and the Width Function

11.1. Local Motions. The functions \v'*N*' and \h'*N*' can be used for *local* vertical and horizontal motion respectively. The distance *N* may be negative; the positive directions are rightward and downward. A local motion is one contained within a line. To avoid unexpected vertical dislocations, it is necessary that the net vertical local motion within a word in filled text and otherwise within a line balance to zero. The escape sequences providing local motion are summarized in the following table.

Vertical Local Motion	Effect in *troff*	Effect in *nroff*	Horizontal Local Motion	Effect in *troff*	Effect in *nroff*
\v'*N*'	Move distance *N*		\ *space*	Unpaddable space-size space	
\h'*N*'	Move distance *N*		\0	Digit-size space	
\u	½ em up	½ line up	\\|	1/6 em space	ignored
\d	½ em down	½ line down	\^	1/12 em space	ignored
\r	1 em up	1 line up			

As an example, E^2 could be generated by a sequence of size changes and motions: E\s−2\v'−0.4m'2\v'0.4m'\s+2 ; note that the 0.4 em vertical motions are at the smaller size.

11.2. Width Function. The *width* function \w'*string*' generates the numerical width of *string* (in basic units). Size and font changes may be embedded in *string*, and will not affect the current environment. For example, .ti −\w'\fB1. 'u could be used to temporarily indent leftward a distance equal to the size of the string **1.**

The width function also sets three number registers. The registers st and sb are set respectively to the highest and lowest extent of *string* relative to the baseline; then, for example, the total height of the string is \n(stu−\n(sbu . In *troff* the number register ct is set to a value between 0 and 3. The value 0 means that all of the characters in *string* were short lower case characters without descenders (like); 1 means that at least one character has a descender (like y); 2 means that at least one character is tall (like H); and 3 means that both tall characters and characters with descenders are present.

11.3. Mark horizontal place. The function \k*x* causes the current horizontal position in the *input line* to be stored in register *x*. For example, the construction \kx*word*\h'|\nxu+3u'*word* will embolden *word* by backing up to almost its beginning and overprinting it, resulting in ***word***.

12. Overstrike, Bracket, Line-drawing, Graphics, and Zero-width Functions

12.1. Overstriking. Automatically centered overstriking of up to nine characters is provided by the *overstrike* function \o'*string*'. The characters in *string* are overprinted with centers aligned; the total width is that of the widest character. *string* may not contain local vertical motion. As examples, \o'e\´' produces é, and \o'\(mo\(sl' produces ∉.

12.2. Zero-width characters. The function \z*c* will output *c* without spacing over it, and can be used to produce left-aligned overstruck combinations. As examples, \z¡+ will produce †, and \(br\z\(rn\(ul\(br will produce a small badly constructed box ⎣⎦.

12.3. Large Brackets. The Special Font usually contains a number of bracket construction pieces ⎡ ⎣ ⎤ ⎦ ⎨ ⎬ | ⎣ ⎦ ⎡ ⎤ that can be combined into various bracket styles. The function \b'*string*' may be used to pile up vertically the characters in *string* (the first character on top and the last at the bottom); the characters are vertically separated by 1 em and the total pile is centered 1/2 em above the current baseline (½ line in *nroff*). For example,

```
\b'\(lc\(lf'E\b'\(rc\(rf'\x'-0.5m'\x'0.5m'
```

produces [E].

12.4. Line drawing. The function \l'*Nc*' (backslash-ell) draws a string of repeated *c*'s towards the right for a distance *N*. If *c* looks like a continuation of an expression for *N*, it may be insulated from *N* with \&. If *c* is not specified, the _ (baseline rule) is used (underline character in *nroff*). If *N* is negative, a backward horizontal motion of size *N* is made before drawing the string. Any space resulting from *N*/(size of *c*) having a remainder is put at the beginning (left end) of the string. If *N* is less than the width of *c*, a single *c* is centered on a distance *N*. In the case of characters that are designed to be connected, such as baseline-rule _, under-rule _, and root-en ‾, the remainder space is covered by overlapping. As an example, a macro to underscore a string can be written

```
.de us
\\$1\  l'|0\(ul'
..
```

or one to draw a box around a string

```
.de bx
\(br\|\\$1\|\(br\    l'|0\(rn'\  l'|0\(ul'
..
```

such that

```
.ul "underlined words"
```

and

```
.bx "words in a box"
```

yield underlined words and |words in a box|.

The function \L'*Nc*' draws a vertical line consisting of the (optional) character *c* stacked vertically apart 1 em (1 line in *nroff*), with the first two characters overlapped, if necessary, to form a continuous line. The default character is the *box rule* | (\(br); the other suitable character is the *bold vertical* | (\(bv). The line is begun without any initial motion relative to the current baseline. A positive *N* specifies a line drawn downward and a negative *N* specifies a line drawn upward. After the line is drawn no compensating motions are made; the instantaneous baseline is at the end of the line.

The horizontal and vertical line drawing functions may be used in combination to produce large boxes. The zero-width *box-rule* and the ½-em wide *under-rule* were designed to form corners when using 1-em vertical spacings. For example the macro

```
.de eb
.sp -1    \"compensate for next automatic baseline spacing
.nf       \"avoid possibly overflowing word buffer
\h'-.5n'\L'|\\nau-1'\l'\\n(.lu+1n\(ul'\L'-|\\nau+1'\l'|0u-.5n\(ul'
.fi
..
```

will draw a box around some text whose beginning vertical place was saved in number register *a* (e.g., using .mk a) as was done for this paragraph.

12.5. Graphics. The function \D'*c...*' draws a graphic object of type *c* according to a sequence of parameters, which are generally pairs of numbers.

\D'l *dh dv*'	draw line from current position by *dh*, *dv*
\D'c *d*'	draw circle of diameter *d* with left side at current position
\D'e d_1 d_2'	draw ellipse of diameters d_1 and d_2

\D'a dh_1 dv_1 dh_2 dv_2' draw arc from current position to dh_1+dh_2, dv_1+dv_2, with center at dh_1, dv_1 from current position

\D'~ dh_1 dv_1 dh_2 dv_2 ...' draw B-spline from current position by dh_1, dv_1, then by dh_2, dv_2, then ...

For example, \D'e0.2i 0.1i' draws the ellipse, and \D'l .2i -.1i'\D'l .1i .1i' the line. A \D with an unknown *c* is processed and copied through to the output for unspecified interpretation; coordinates are interpreted alternately as horizontal and vertical values.

Numbers taken as horizontal (first, third, etc.) have default scaling of ems; vertical numbers (second, fourth, etc.) have default scaling of *V*s (§1.3). The position after a graphical object has been drawn is at its end; for circles and ellipses, the end is at the right side.

13. Hyphenation.

Automatic hyphenation may be switched off and on. When switched on with hy, several variants may be set. A *hyphenation indicator* character may be embedded in a word to specify desired hyphenation points, or may be prefixed to suppress hyphenation. In addition, the user may specify a small list of exception words.

Only words that consist of a central alphabetic string surrounded by (usually null) non-alphabetic strings are candidates for automatic hyphenation. Words that contain hyphens (minus), em-dashes (\(em), or hyphenation indicator characters are always subject to splitting after those characters, whether automatic hyphenation is on or off.

.nh   hyphenate   -   E

Automatic hyphenation is turned off.

.hy *N*   on, $N=1$ on, $N=1$ E

Automatic hyphenation is turned on for $N \ge 1$, or off for $N=0$. If $N=2$, last lines (ones that will cause a trap) are not hyphenated. For $N=4$ and 8, the last and first two characters respectively of a word are not split off. These values are additive; i.e., $N=14$ will invoke all three restrictions.

.hc *c*   \%   \%   E

Hyphenation indicator character is set to *c* or to the default \%. The indicator does not appear in the output.

.hw *word ...*   ignored   -

Specify hyphenation points in words with embedded minus signs. Versions of a word with terminal *s* are implied; *i.e.*, dig-it implies dig-its. This list is examined initially and after each suffix stripping. The space available is small.

14. Three-Part Titles.

The titling function tl provides for automatic placement of three fields at the left, center, and right of a line with a title length specifiable with lt. tl may be used anywhere, and is independent of the normal text collecting process. A common use is in header and footer macros.

.tl ' *left* ' *center* ' *right* '   -   -

The strings *left*, *center*, and *right* are respectively left-adjusted, centered, and right-adjusted in the current title length. Any of the strings may be empty, and overlapping is permitted. If the page-number character (initially %) is found within any of the fields it is replaced by the current page number in the format assigned to register %. Any character may be used in place of ' as the string delimiter.

`.pc` *c*   %   off   -

The page number character is set to *c*, or removed. The page number register remains %.

`.lt` ±*N*   6.5in   previous   E,**m**

Length of title is set to ± *N*. The line length and the title length are independent. Indents do not apply to titles; page offsets do.

15. Output Line Numbering.

Automatic sequence numbering of output lines may be requested with `nm`. When in effect, a three-digit, arabic number plus a digit-space is prefixed to output text lines. The text lines are thus offset by four digit-spaces, and otherwise retain their line length; a reduction in line length may be desired to keep the right margin aligned with an earlier margin. Blank lines, other vertical spaces, and lines generated by `tl` are not numbered. Numbering can be temporarily suspended with `nn`, or with an `.nm` followed by a later `.nm +0`. In addition, a line number indent *I*, and the number-text separation *S* may be specified in digit-spaces. Further, it can be specified that only those line numbers that are multiples of some number *M* are to be printed (the others will appear as blank number fields).

`.nm` ±*N M S I*   off   E

Line number mode. If ± *N* is given, line numbering is turned on, and the next output line numbered is numbered ± *N*. Default values are *M*=1, *S*=1, and *I*=0. Parameters corresponding to missing arguments are unaffected; a non-numeric argument is considered missing. In the absence of all arguments, numbering is turned off; the next line number is preserved for possible further use in number register `ln`.

`.nn` *N*   -   *N*=1   E

The next *N* text output lines are not numbered.

As an example, the paragraph portions of this section are numbered with *M*=3: `.nm 1 3` was placed at the beginning; `.nm` was placed at the end of the first paragraph; and `.nm +0` was placed in front of this paragraph; and `.nm` finally placed at the end. Line lengths were also changed (by `\w'0000'u`) to keep the right side aligned. Another example is `.nm +5 5 x 3`, which turns on numbering with the line number of the next line to be 5 greater than the last numbered line, with *M*=5, with spacing *S* untouched, and with the indent *I* set to 3.

16. Conditional Acceptance of Input

In the following *c* is a one-character built-in *condition* name, `!` signifies *not*, *N* is a numerical expression, *string1* and *string2* are strings delimited by any non-blank, non-numeric character not in the strings, and *anything* represents what is conditionally accepted.

`.if` *c anything*   -   -

If condition *c* true, accept *anything* as input; in multi-line case use `\{`*anything* `\}`.

`.if !` *c anything*   -   -

If condition *c* false, accept *anything*.

`.if` *N anything*   -   **u**

If expression *N* > 0, accept *anything*.

`.if !` *N anything*   -   **u**

If expression *N* ≤ 0 [sic], accept *anything*.

.if ' *string1* ' *string2* ' *anything* –

If *string1* identical to *string2* , accept *anything* .

.if !' *string1* ' *string2* ' *anything* –

If *string1* not identical to *string2* , accept *anything* .

.ie *c anything* – **u**

If portion of if-else; all of the if forms above are valid.

.el *anything* – –

Else portion of if-else.

The built-in condition names are:

Condition Name	True If
o	Current page number is odd
e	Current page number is even
t	Formatter is *troff*
n	Formatter is *nroff*

If the condition **c** is true, or if the number *N* is greater than zero, or if the strings compare identically (including motions and character size and font), *anything* is accepted as input. If a ! precedes the condition, number, or string comparison, the sense of the acceptance is reversed.

Any spaces between the condition and the beginning of *anything* are skipped over. The *anything* can be either a single input line (text, macro, or whatever) or a number of input lines. In the multi-line case, the first line must begin with a left delimiter \{ and the last line must end with a right delimiter \}.

The request ie (if-else) is identical to if except that the acceptance state is remembered. A subsequent and matching el (else) request then uses the reverse sense of that state. ie-el pairs may be nested.

Some examples are:

```
.if e .tl 'Even Page %'''
```

which outputs a title if the page number is even; and

```
.ie \n%>1 \{\
'       sp 0.5i
.       tl 'Page %'''
'       sp |1.2i \}
.el .sp |2.5i
```

which treats page 1 differently from other pages.

17. Environment Switching.

A number of the parameters that control the text processing are gathered together into an *environment* , which can be switched by the user. The environment parameters are those associated with requests noting E in their *Notes* column; in addition, partially collected lines and words are in the environment. Everything else is global; examples are page-oriented parameters, diversion-oriented parameters, number registers, and macro and string definitions. All environments are initialized with default parameter values.

.ev *N* *N*=0 previous –

Environment switched to environment $0 \le N \le 2$. Switching is done in push-down fashion so that restoring a previous environment *must* be done with

.ev rather than specific reference. Note that what is pushed down and restored is the environment *number*, not its contents.

18. Insertions from the Standard Input

The input can be temporarily switched to the system standard input with rd, which will switch back when two consecutive newlines are found (the extra blank line is not used). This mechanism is intended for insertions in form-letter-like documentation. The standard input can be the user keyboard, a pipe, or a file.

.rd *prompt* - *prompt*= BEL -

Read insertion from the standard input until two newlines in a row are found. If the standard input is the users keyboard, *prompt* (or a BEL) is written onto the standard output. rd behaves like a macro, and arguments may be placed after *prompt*.

.ex - - -

Exit from *nroff*/*troff*. Text processing is terminated exactly as if all input had ended.

If insertions are to be taken from the terminal keyboard while output is being printed on the terminal, the command line option -q will turn off the echoing of keyboard input and prompt only with BEL. The regular input and insertion input cannot simultaneously come from the standard input.

As an example, multiple copies of a form letter may be prepared by entering the insertions for all the copies in one file to be used as the standard input, and causing the file containing the letter to reinvoke itself with nx (§19); the process would ultimately be ended by an ex in the insertion file.

19. Input/Output File Switching

.so *filename* - -

Switch source file. The top input (file reading) level is switched to *filename*. When the new file ends, input is again taken from the original file. so's may be nested.

.nx *filename* end-of-file -

Next file is *filename*. The current file is considered ended, and the input is immediately switched to *filename*.

.sy *string* - -

Execute program from *string*, which is the rest of the input line. The output is not collected automatically. The number register $$, which contains the process id of the *troff* process, may be useful in generating unique filenames for output.

.pi *string* - -

Pipe output to *string*, which is the rest of the input line. This request must occur before any printing occurs; typically it is the first line of input.

.cf *filename* - -

Copy contents of file *filename* to output, completely unprocessed. The file is assumed to contain something meaningful to subsequent processes.

20. Miscellaneous

.mc *c N* — off E,**m**

Specifies that a *margin* character *c* appear a distance *N* to the right of the right margin after each non-empty text line (except those produced by tl). If the output line is too long (as can happen in nofill mode) the character will be appended to the line. If *N* is not given, the previous *N* is used; the initial *N* is 0.2 inches in *nroff* and 1 em in *troff*. The margin character used with this paragraph was a 12-point box-rule.

.tm *string* — newline —

After skipping initial blanks, *string* (rest of the line) is read in copy mode and written on the standard error.

.ab *string* — newline —

After skipping initial blanks, *string* (rest of the line) is read in copy mode and written on the standard error. *Troff* or *nroff* then exit.

.ig *yy* — *.yy*=.. —

Ignore input lines. ig behaves exactly like de (§7) except that the input is discarded. The input is read in copy mode, and any auto-incremented registers will be affected.

.lf *N filename* — —

Set line number to *N* and filename to *filename* for purposes of subsequent error messages, etc. The number register [sic] .F contains the name of the current input file, as set by command line argument, so, nx, or lf. The number register .c contains the number of input lines read from the current file, again perhaps as modified by

.pm *t* — all —

Print macros. The names and sizes of all of the defined macros and strings are printed on the standard error; if *t* is given, only the total of the sizes is printed. The sizes is given in blocks of 128 characters.

.fl — — B

Flush output buffer. Force output, including any pending position information.

21. Output and Error Messages.

The output from tm, pm, and the prompt from rd, as well as various error messages, are written onto the standard error. The latter is different from the standard output, where formatted text goes. By default, both are written onto the users terminal, but they can be independently redirected.

Various error conditions may occur during the operation of *nroff* and *troff*. Certain less serious errors having only local impact do not cause processing to terminate. Two examples are *word overflow*, caused by a word that is too large to fit into the word buffer (in fill mode), and *line overflow*, caused by an output line that grew too large to fit in the line buffer. In both cases, a message is printed, the offending excess is discarded, and the affected word or line is marked at the point of truncation with a in *nroff* and a ☜ in *troff*. Processing continues if possible, on the grounds that output useful for debugging may be produced. If a serious error occurs, processing terminates, and a message is printed, along with a list of the macro names currently active. Examples of serious errors include the inability to create, read, or write files, and the exceeding of certain internal limits that make future output unlikely to be useful.

22. Output Language

Troff produces its output in a language that is independent of any specific output device, except that the numbers in it have been computed on the basis of the resolution of the device, and the sizes, fonts, and characters that that device can print. Nevertheless it is quite possible to interpret that output on a different device, within the latter's capabilities.

s *n*	set point size to *n*
f *n*	set font to *n*
c *c*	print character *c*
C *name*	print the character called *name*; terminate *name* by white space
N *n*	print character *n* on current font
H *n*	go to absolute horizontal position *n* ($n \geq 0$)
V *n*	go to absolute vertical position *n* ($n \geq 0$, down is positive)
h *n*	go *n* units horizontally; $n < 0$ is to the left
v *n*	go *n* units vertically; $n < 0$ is up
nnc	move right *nn*, then print UTF character *c*; *nn* must be exactly 2 digits
p *n*	new page *n* begins set vertical position to 0
n *b a*	end of line (information only—no action); *b* = space before line, *a* = after
w	paddable word space (information only—no action)
D *c* ...\n	graphics function *c*; see below
x ...\n	device control functions; see below
# ...\n	comment

All position values are in units. Sequences that end in digits must be followed by a non-digit. Blanks, tabs and newlines may occur as separators in the input, and are mandatory to separate constructions that would otherwise be confused. Graphics functions, device control functions, and comments extend to the end of the line they occur on.

The device control and graphics commands are intended as open-ended families, to be expanded as needed. The graphics functions coincide directly with the \D sequences:

Dl dh dv	draw line from current position by dh, dv
Dc d	draw circle of diameter d with left side here
De dh_1 dv_2	draw ellipse of diameters dh_1 and dv_2
Da dh_1 dv_1 dh_2 dv_2	draw arc from current position to $dh_1 + dh_2$, $dv_1 + dv_2$, center at dh_1, dv_1 from current position
D~ dh_1 dv_1 dh_2 dv_2 ...	draw B-spline from current position to dh_1, dv_1, then to dh_2, dv_2, then to ...
Dz dh_1 dv_1 dh_2 dv_2 ...	for any other z is uninterpreted

In all of these, dh, dv is an increment on the current horizontal and vertical position, with down and right positive. All distances and dimensions are in units.

The device control functions begin with x, then a command, then other parameters.

x T *s*	name of typesetter is *s*
x r *n h v*	resolution is *n* units/inch; *h* = minimum horizontal motion, *v* = minimum vertical
x i	initialize
x f *n s*	mount font *s* on font position *n*
x p	pause can restart
x s	stop done forever
x t	generate trailer information, if any
x H *n*	set character height to *n*
x S *n*	set slant to *n*

x X *any*   generated by the \X function
x *any*   to be ignored if not recognized

Subcommands like x i may be spelled out like x init.

The commands x T, x r ..., and x i must occur first; fonts must be mounted before they can be used; x s comes last. There are no other order requirements.

The following is the output from hello, world for a typical printer, as described in §23:

```
x T utf
x res 720 1 1
x init
V0
p1

x font 1 R
x font 2 I
x font 3 B
x font 4 BI
x font 5 CW
x font 6 H
x font 7 HB
x font 8 HX
x font 9 S1
x font 10 S

s10
f1
H0
s10
f1
V0
H720
V120
ch
50e44l28l28o50,w58w72o50r33l28dn120 0
x trailer
V7920
x stop
```

Troff output is normally not redundant; size and font changes and position information are not included unless needed. Nevertheless, each page is self-contained, for the benefit of postprocessors that re-order pages or process only a subset.

23. Device and Font Description Files

The parameters that describe a output device *name* are read from the directory /sys/lib/troff/font/dev*name*, each time *troff* is invoked. The device name is provided by default, by the environment variable TYPESETTER, or by a command-line argument -T*name*. The default device name is utf, for UTF-encoded Unicode characters. The pre-defined string .T contains the name of the device. The -F command-line option may be used to change the default directory.

23.1. Device description file. General parameters of the device are stored, one per line, in the file /sys/lib/troff/font/dev*name*/DESC, as a sequence of names and values. *Troff* recognizes these parameters, and ignores any others that may be present for specific drivers:

fonts n F_1 F_2 ... F_n
sizes s_1 s_2 ... 0

```
res n
hor n
vert n
unitwidth n
charset
```

list of multi-character character names (optional)

The F_i are font names to be initially mounted. The list of sizes is a set of integers representing some or all of the legal sizes the device can produce, terminated by a zero. The res parameter gives the resolution of the machine in units per inch; hor and ver give the minimum number of units that can be moved horizontally and vertically.

Character widths for each font are assumed to be given in machine units at point size unitwidth. (In other words, a character with a width of *n* is *n* units wide at size unitwidth.) All widths are integers at all sizes.

A list of valid character names may be introduced by charset; the list of names is optional.

A line whose first non-blank character is # is a comment. Except that charset must occur last, parameters may appear in any order.

Here is a subset of the DESC file for a typical Postscript printer:

```
# Description file for Postscript printers.

fonts 10 R I B BI CW H HB HX S1 S
sizes 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
   24 25 26 27 28 29 30 31 32 33 34 35 36 38 40 44 48 54 60 72 0
res 720
hor 1
vert 1
unitwidth 10
charset
hy ct fi fl ff Fi Fl dg em 14 34 12 en aa
ga ru sc dd -> br Sl ps cs cy as os =. ld
rd le ge pp -+ ob vr
sq bx ci fa te ** pl mi eq ~= *A *B *X *D
*E *F *G *Y *I *K *L *M *N *O *P *R *H *S *T *U *W
*C *Q *Z ul rn *a *b *x *d *e *f *g *y *i *k
*l *m *n *o *p *h *r *s *t *u *w *c *q *z
```

23.2. Font description files. Each font is described by an analogous description file, which begins with parameters of the font, one per line, followed by a list of characters and widths. The file for font *f* is /usr/sys/lib/troff/font/dev*name*/*f*.

```
name str             name of font is str
ligatures ... 0      list of ligatures
spacewidth n         width of a space on this font
special              this is a special font
charset
```

list of character name, width, ascender/descender, code, tab separated

The name and charset fields are mandatory; charset must be last. Comments are permitted, as are other unrecognized parameters.

Each line following charset describes one character: its name, its width in units as described above, ascender/descender information, and a decimal, octal or hexadecimal value by which the output device knows it (the \N number of the character). The character name is arbitrary, except that --- signifies an unnamed character. If the width field contains ", the name is a synonym for the previous character. The ascender/descender field is 1 if the character has a descender (hangs below the baseline, like y), is 2 if it has an ascender (is tall, like Y), is 3 if both, and is 0 if neither. The value is returned in the ct register, as computed by the \w function (§11.2).

Here are excerpts from a typical font description file for the same Postscript printer.

hy	33	0	45	hyphen \(hy
-	"			- is a synonym for \(hy
Q	72	3	81	
a	44	0	97	
b	50	2	98	
c	44	0	99	
d	50	2	100	
y	50	1	121	
em	100	0	208	
---	44	2	220	Pound symbol £, \N'220'
---	36	0	221	centered dot \N'221'

This says, for example, that the width of the letter a is 44 units at point size 10, the value of unitwidth. Point sizes are scaled linearly and rounded, so the width of a will be 44 at size 10, 40 at size 9, 35 at size 8, and so on.

Tutorial Examples

Introduction

It is almost always necessary to prepare at least a small set of macro definitions to describe a document. Such common formatting needs as page margins and footnotes are deliberately not built into *nroff* and *troff*. Instead, the macro and string definition, number register, diversion, environment switching, page-position trap, and conditional input mechanisms provide the basis for user-defined implementations.

For most uses, a standard package like -ms or -mm is the right choice. The next stage is to augment that, or to selectively replace macros from the standard package. The last stage, much harder, is to write one s own from scratch. This is not a task for the novice.

The examples discussed here are intended to be useful and somewhat realistic, but will not necessarily cover all relevant contingencies. Explicit numerical parameters are used in the examples to make them easier to read and to illustrate typical values. In many cases, number registers would be used to reduce the number of places where numerical information is kept, and to concentrate conditional parameter initialization like that which depends on whether *troff* or *nroff* is being used.

Page Margins

As discussed in §3, header and footer macros are usually defined to describe the top and bottom page margin areas respectively. A trap is planted at page position 0 for the header, and at $-N$ (N from the page bottom) for the footer. The simplest such definitions might be

```
.de  hd    \"define   header
'sp  1i
..         \"end  definition
.de  fo    \"define   footer
'bp
..         \"end  definition
.wh  0  hd
.wh  -1i  fo
```

which provide blank 1 inch top and bottom margins. The header will occur on the *first* page only if the definition and trap exist prior to the initial pseudo-page transition (§3). In fill mode, the output line that springs the footer trap was typically forced out because some part or whole word didnt fit on it. If anything in the footer and header that follows causes a break, that word or part word will be forced out. In this and other examples, requests like bp and sp that normally cause breaks are invoked using the no-break control character ' to avoid this. When the header/footer design contains material requiring independent text processing, the environment may be switched, avoiding most interaction with the running text.

A more realistic example would be

```
.de hd  \"header
.if \\n%>1 \{\
'sp  0.5i-1      \"tl base at 0.5i
.tl ''- % -''    \"centered page number
.ps      \"restore size
.ft      \"restore font
.vs  \}  \"restore vs
'sp  1.0i        \"space to 1.0i
.ns      \"turn on no-space mode
..
.de fo  \"footer
.ps 10  \"set footer/header size
.ft R   \"set font
.vs 12p \"set baseline spacing
.if \\n%=1 \{\
'sp  \\n(.pu-0.5i-1    \"tl base 0.5i up
.tl ''- % -'' \}  \"first page number
'bp
..
.wh 0 hd
.wh -1i fo
```

which sets the size, font, and baseline spacing for the header/footer material, and ultimately restores them. The material in this case is a page number at the bottom of the first page and at the top of the remaining pages. The sp's refer to absolute positions to avoid dependence on the baseline spacing. Another reason for doing this in the footer is that the footer is invoked by printing a line whose vertical spacing swept past the trap position by possibly as much as the baseline spacing. No-space mode is turned on at the end of hd to render ineffective accidental occurrences of sp at the top of the running text.

This method of restoring size, font, etc. presupposes that such requests (that set *previous* value) are *not* used in the running text. A better scheme is to save and restore both the current *and* previous values as shown for size in the following:

```
.de fo
.nr s1 \\n(.s    \"current size
.ps
.nr s2 \\n(.s    \"previous size
.  ---   \"rest of footer
..
.de hd
.  ---   \"header stuff
.ps \\n(s2   \"restore previous size
.ps \\n(s1   \"restore current size
..
```

Page numbers may be printed in the bottom margin by a separate macro triggered during the footer's page ejection:

```
.de bn  \"bottom number
.tl ''- % -''    \"centered page number
..
.wh -0.5i-1v  bn  \"tl base 0.5i up
```

Paragraphs and Headings

The housekeeping associated with starting a new paragraph should be collected in a paragraph macro that, for example, does the desired preparagraph spacing, forces the correct font, size, baseline spacing, and indent, checks that enough space remains for *more than one* line, and requests a temporary indent.

```
.de pg     \"paragraph
.br        \"break
.ft R      \"force  font,
.ps 10     \"size,
.vs 12p    \"spacing,
.in 0      \"and  indent
.sp 0.4    \"prespace
.ne 1+\\n(.Vu    \"want  more  than  1 line
.ti 0.2i           \"temp  indent
..
```

The first break in pg will force out any previous partial lines, and must occur before the vs. The forcing of font, etc., is partly a defense against prior error and partly to permit things like section heading macros to set parameters only once. The prespacing parameter is suitable for *troff*; a larger space, at least as big as the output device vertical resolution, would be more suitable in *nroff*. The choice of remaining space to test for in the ne is the smallest amount greater than one line (the .V is the available vertical resolution).

A macro to automatically number section headings might look like:

```
.de sc   \"section
.  ---   \"force  font,  etc.
.sp 0.4  \"prespace
.ne 2.4+\\n(.Vu   \"want  2.4+  lines
.fi
\\n+S.
..
.nr S 0 1         \"init  S
```

The usage is .sc, followed by the section heading text, followed by .pg. The ne test value includes one line of heading, 0.4 line in the following pg, and one line of the paragraph text. A word consisting of the next section number and a period is produced to begin the heading line. The format of the number may be set by af (§8).

Another common form is the labeled, indented paragraph, where the label protrudes left into the indent space.

```
.de lp   \"labeled  paragraph
.pg
.in 0.5i          \"paragraph  indent
.ta 0.2i 0.5i     \"label,  paragraph
.ti 0
\t\\$1\t\c         \"flow  into  paragraph
..
```

The intended usage is .lp *label*; *label* will begin at 0.2 inch, and cannot exceed a length of 0.3 inch without intruding into the paragraph. The label could be right adjusted against 0.4 inch by setting the tabs instead with .ta 0.4iR 0.5i. The last line of lp ends with \c so that it will become a part of the first line of the text that follows.

Multiple Column Output

The production of multiple column pages requires the footer macro to decide whether it was invoked by other than the last column, so that it will begin a new column rather than produce the bottom margin. The header can initialize a column register that the footer will increment and test. The following is arranged for two columns, but is easily modified for more.

```
.de  hd   \"header
.  ---
.nr  cl  0  1        \"init   column   count
.mk       \"mark   top  of  text
..

.de  fo   \"footer
.ie  \\n+(cl<2   \{\
.po  +3.4i          \"next   column;   3.1+0.3
.rt       \"back   to  mark
.ns  \}   \"no-space   mode
.el  \{\
.po  \\nMu          \"restore   left   margin
.  ---
'bp  \}
..
.ll  3.1i           \"column   width
.nr  M  \\n(.o      \"save   left   margin
```

Typically a portion of the top of the first page contains full width text; the request for the narrower line length, as well as another .mk would be made where the two column output was to begin.

Footnotes

The footnote mechanism to be described is used by embedding the footnotes in the input text at the point of reference, demarcated by an initial .fn and a terminal .ef :

```
.fn
Footnote text and control lines...
.ef
```

In the following, footnotes are processed in a separate environment and diverted for later printing in the space immediately prior to the bottom margin. There is provision for the case where the last collected footnote doesn t completely fit in the available space.

```
.de  hd   \"header
.  ---
.nr  x  0  1        \"init   footnote   count
.nr  y  0-\\nb      \"current   footer   place
.ch  fo  -\\nbu     \"reset   footer   trap
.if  \\n(dn   .fz   \"leftover   footnote
..

.de  fo   \"footer
.nr  dn  0   \"zero   last   diversion   size
.if  \\nx  \{\
.ev  1    \"expand   footnotes   in  ev1
.nf       \"retain   vertical   size
.FN       \"footnotes
.rm  FN   \"delete   it

.if  "\\n(.z"fy"    .di   \"end  overflow   di
.nr  x  0  \"disable   fx
.ev   \}  \"pop  environment
.  ---
'bp
..
```

```
.de fx  \"process  footnote  overflow
.if \\nx  .di fy \"divert  overflow
..

.de fn  \"start  footnote
.da FN  \"divert  (append)  footnote
.ev 1   \"in environment  1
.if \\n+x=1  .fs   \"if 1st, separator
.fi     \"fill  mode
..

.de ef  \"end  footnote
.br     \"finish  output
.nr z \\n(.v     \"save  spacing
.ev     \"pop  ev
.di     \"end  diversion
.nr y -\\n(dn    \"new  footer  position,
.if \\nx=1  .nr y -(\\n(.v-\\nz)   \
        \"uncertainty   correction
.ch fo \\nyu     \"y is negative
.if (\\n(nl+1v)>(\\n(.p+\\ny)     \
.ch fo \\n(nlu+1v         \"didn't  fit
..

.de fs  \"separator
\l'1i'  \"1 inch rule
.br
..

.de fz  \"get  leftover  footnote
.fn
.nf     \"retain  vertical  size
.fy     \"where  fx put it
.ef
..

.nr b 1.0i   \"bottom  margin  size
.wh 0 hd     \"header  trap
.wh 12i fo   \"footer  trap->temp  pos
.wh -\\nbu  fx   \"fx at footer  position
.ch fo -\\nbu    \"conceal  fx with  fo
```

The header hd initializes footnote count register x, and sets both the current footer trap position register y and the footer trap itself to a nominal position specified in register b. In addition, if the register dn indicates a leftover footnote, fz is invoked to reprocess it. The footnote start macro fn begins a diversion (append) in environment 1, and increments the count x; if the count is one, the footnote separator fs is interpolated. The separator is kept in a separate macro to permit user redefinition.

The footnote end macro ef restores the previous environment and ends the diversion after saving the spacing size in register z. y is then decremented by the size of the footnote, available in dn; then on the first footnote, y is further decremented by the difference in vertical base-line spacings of the two environments, to prevent the late triggering of the footer trap from causing the last line of the combined footnotes to overflow. The footer trap is then set to the lower (on the page) of y or the current page position (nl) plus one line, to allow for printing the reference line.

If indicated by x, the footer fo rereads the footnotes from FN in nofill mode in environment 1, and deletes FN. If the footnotes were too large to fit, the macro fx will be trap-invoked to redivert the overflow into fy, and the register dn will later indicate to the header whether fy is empty.

Both fo and fx are planted in the nominal footer trap position in an order that causes fx to be concealed unless the fo trap is moved. The footer then terminates the overflow diversion, if necessary, and zeros x to disable fx, because the uncertainty correction together with a not-too-late triggering of the footer can result in the footnote rereading finishing before reaching the fx trap.

A good exercise for the student is to combine the multiple-column and footnote mechanisms.

The Last Page

After the last input file has ended, *nroff* and *troff* invoke the *end macro* (§7), if any, and when it finishes, eject the remainder of the page. During the eject, any traps encountered are processed normally. At the end of this last page, processing terminates unless a partial line, word, or partial word remains. If it is desired that another page be started, the end-macro

```
.de  en   \"end-macro
\c
'bp
..
.em  en
```

will deposit a null partial word, and produce another last page.

Special Character Names

The following table lists names for a set of characters, most of which have traditionally been provided by *troff* using the special or symbol font. Many of these sequences are old ways to get what are now Unicode characters; Lucida Sans, for example, has glyphs corresponding to many of these but does not have the special sequences. Therefore the *troff* sequence \(*F gives the character Φ from the Times font instead of the character ‡ from the current font, in this case Lucida Sans. Not all sequences print on any particular device, including this one; Peter faces appear in their place.

´	\'	μ	\(*m	C	\(|=
`	\`	ν	\(*n	~	\(ap
	\(em	ξ	\(*c	`	\(!=
	\(en	ο	\(*o	´	\(->
–	\(hy	π	\(*p	ı	\(<-
	\-	ρ	\(*r	`	\(ua
	\(bu	σ	\(*s	ˆ	\(da
¡	\(sq	ς	\(ts	×	\(mu
_	\(ru	τ	\(*t	÷	\(di
	\(14	υ	\(*u	±	\(+-
	\(12	φ	\(*f	*	\(cu
	\(34	χ	\(*x	)	\(ca
	\(fi	ψ	\(*q		\(sb
	\(fl	ω	\(*w		\(sp
	\(ff	Α	\(*A		\(ib
	\(Fi	Β	\(*B		\(ip
	\(Fl	Γ	\(*G		\(if
°	\(de	Δ	\(*D		\(pd
	\(dg	Ε	\(*E		\(gr
′	\(fm	Ζ	\(*Z	¬	\(no
	\(ct	Η	\(*Y	+	\(is
®	\(rg	Θ	\(*H		\(pt
©	\(co	Ι	\(*I		\(es
+	\(pl	Κ	\(*K	∈	\(mo
−	\(mi	Λ	\(*L	|	\(br
=	\(eq	Μ	\(*M	!	\(dd
	\(**	Ν	\(*N	☞	\(rh
	\(sc	Ξ	\(*C	☜	\(lh
´	\(aa	Ο	\(*O		\(bs
`	\(ga	Π	\(*P	|	\(or
_	\(ul	Ρ	\(*R	Ë	\(ci
/	\(sl	Σ	\(*S	⎡	\(lt
α	\(*a	Τ	\(*T	⎣	\(lb
β	\(*b	Υ	\(*U	⎤	\(rt
γ	\(*g	Φ	\(*F	⎦	\(rb
δ	\(*d	Χ	\(*X	{	\(lk
ε	\(*e	Ψ	\(*Q	}	\(rk
ζ	\(*z	Ω	\(*W	|	\(bv
η	\(*y		\(sr	⌊	\(lf
θ	\(*h	‾	\(rn	⌋	\(rf
ι	\(*i	g	\(>=	⌈	\(lc
κ	\(*k	f	\(<=	⌉	\(rc
λ	\(*l	a	\(==		\(LH

Using SPIN

Gerard J. Holzmann
gerard@plan9.bell-labs.com

ABSTRACT

SPIN can be used for proving or disproving logical properties of concurrent systems. To render the proofs, a concurrent system is first modeled in a formal specification language called PROMELA. The language allows one to specify the behaviors of asynchronously executing processes that may interact through synchronous or asynchronous message passing, or through direct access to shared variables.

System models specified in this way can be verified for both safety and liveness properties. The specification of general properties in linear time temporal logic is also supported.

The first part of this manual discusses the basic features of the specification language PROMELA. The second part describes the verifier SPIN.

1. The Language PROMELA

PROMELA is short for Protocol Meta Language [Ho91]. PROMELA is a *modeling* language, not a programming language. A formal model differs in two essential ways from an implementation. First, a model is meant to be an abstraction of a design that contains only those aspects of the design that are directly relevant to the properties one is interested in proving. Second, a formal model must contain things that are typically not part of an implementation, such as worst-case assumptions about the behavior of the environment that may interact with the system being studied, and a formal statement of relevant correctness properties. It is possible to mechanically extract abstract models from implementation level code, as discussed, for instance in [HS99].

Verification with SPIN is often performed in a series of steps, with the construction of increasingly detailed models. Each model can be verified under different types of assumptions about the environment and for different types of correctness properties. If a property is not valid for the given assumptions about system behavior, the verifier can produce a counter-example that demonstrates how the property may be violated. If a property is valid, it may be possible to simplify the model based on that fact, and prove still other properties.

Section 1.1 covers the basic building blocks of the language. Section 1.2 introduces the control flow structures. Section 1.3 explains how correctness properties are specified. Section 1.4 concludes the first part with a discussion of special predefined variables and functions that can be used to express some correctness properties.

Up to date manual pages for SPIN can always be found online at: http://cm.bell-labs.com/cm/cs/what/spin/Man/

1.1. Basics

A PROMELA model can contain three different types of objects:

Processes (section 1.1.1),
Variables (section 1.1.2),
Message channels (section 1.1.3).

All processes are global objects. For obvious reasons, a PROMELA model must contain at least one process to be meaningful. Since SPIN is specifically meant to prove properties of concurrent systems, a model typically contains more than one process.

Message channels and variables, the two basic types of data objects, can be declared with either a global scope or a local scope. A data object with global scope can be referred to by all processes. A data object with a local scope can be referred to by just a single process: the process that declares and instantiates the object. As usual, all objects must be declared in the specification before they are referenced.

1.1.1. Processes

Here is a simple process that does nothing except print a line of text:

```
init {
        printf("it works\n")
}
```

There are a few things to note. Init is a predefined keyword from the language. It can be used to declare and instantiate a single initial process in the model. (It is comparable to the main procedure of a C program.) The init process does not take arguments, but it can start up (instantiate) other processes that do. Printf is one of a few built-in procedures in the language. It behaves the same as the C version. Note, finally, that no semicolon follows the single printf statement in the above example. In PROMELA, semicolons are used as statement separators, not statement terminators. (The SPIN parser, however, is lenient on this issue.)

Any process can start new processes by using another built-in procedure called run. For example,

```
proctype you_run(byte x)
{
        printf("my x is: %d\n", x)
}

init {
        run you_run(1);
        run you_run(2)
}
```

The word proctype is again a keyword that introduces the declaration of a new type of process. In this case, we have named that type you_run and declared that all instantiations of processes of this type will take one argument: a data object of type byte, that can be referred to within this process by the name x. Instances of a proctype can be created with the predefined procedure run, as shown in the example. When the run statement completes, a copy of the process has been started, and all its arguments have been initialized with the arguments provided. The process may, but need not, have performed any statement executions at this point. It is now part of the concurrent system, and its execution can be interleaved arbitrarily with those of the other, already executing processes. (More about the semantics of execution follows shortly.)

In many cases, we are only interested in creating a single instance of each process type that is declared, and the processes require no arguments. We can define this by prefixing the keyword proctype from the process declaration with another keyword: active . Instances of all active proctypes are created when the system itself is initialized. We could, for instance, have avoided the use of init by declaring the corresponding process in the last example as follows:

```
active proctype main() {
        run you_run(1);
        run you_run(2)
}
```

Note that there are no parameters to instantiate in this case. Had they been declared, they would default to a zero value, just like all other data objects that are not explicitly instantiated.

Multiple copies of a process type can also be created in this way. For example:

```
active [4] proctype try_me() {
        printf("hi, i am process %d\n", _pid)
}
```

creates four processes. A predefined variable _pid is assigned to each running process, and holds its unique process instantiation number. In some cases, this number is needed when a reference has to be made to a specific process.

Summarizing: process behavior is declared in proctype definitions, and it is instantiated with either run statements or with the prefix active . Within a proctype declaration, statements are separated (not terminated) by semicolons. As we shall see in examples that follow, instead of the semicolon, one can also use the alternative separator -> (arrow), wherever that may help to clarify the structure of a PROMELA model.

Semantics of Execution

In PROMELA there is no difference between a condition or expression and a statement. Fundamental to the semantics of the language is the notion of the *executability* of statements. Statements are either executable or blocked. Executability is the basic means of enforcing synchronization between the processes in a distributed system. A process can wait for an event to happen by waiting for a statement to become executable. For instance, instead of writing a busy wait loop:

```
while (a != b)  /* not valid Promela syntax */
        skip;   /* wait for a==b */
...
```

we achieve the same effect in PROMELA with the statement

```
(a == b);
...
```

Often we indicate that the continuation of an execution is conditional on the truth of some expression by using the alternate statement separator:

```
(a == b) -> ...
```

Assignments and printf statements are always executable in PROMELA . A condition, however, can only be executed (passed) when it holds. If the condition does not hold, execution blocks until it does. There are similar rules for determining the executability of all other primitive and compound statements in the language. The semantics of each statement is defined in terms of rules for executability and effect. The rules for executability set a precondition on the state of the system in which a statement can be executed. The effect defines how a statement will alter a system state when executed.

PROMELA assumes that all individual statements are executed atomically: that is, they model the smallest meaningful entities of execution in the system being studied. This means that PROMELA defines the standard asynchronous interleaving model of execution, where a supposed scheduler is free at each point in the execution to select any one of the processes to proceed by executing a single primitive statement. Synchronization constraints can be used to influence the interleaving patterns. It is the purpose of a concurrent system's design to constrain those patterns in such a way that no correctness requirements can be violated, and all service requirements are met. It is the purpose of the verifier either to find counter-examples to a designer's claim that this goal has been met, or to demonstrate that the claim is indeed valid.

1.1.2. Variables

The table summarizes the five basic data types used in PROMELA. Bit and bool are synonyms for a single bit of information. The first three types can store only unsigned quantities. The last two can hold either positive or negative values. The precise value ranges of variables of types short and int is implementation dependent, and corresponds to those of the same types in C programs that are compiled for the same hardware. The values given in the table are most common.

Type	Range
bit	0..1
bool	0..1
byte	0..255
short	-2^{15} .. $2^{15}-1$
int	-2^{31} .. $2^{31}-1$

The following example program declares a array of two elements of type bool and a scalar variable turn of the same type. Note that the example relies on the fact that _pid is either 0 or 1 here.

```
/*
 * Peterson's algorithm for enforcing
 * mutual exclusion between two processes
 * competing for access to a critical section
 */
bool turn, want[2];

active [2] proctype user()
{
again:
        want[_pid] = 1; turn = _pid;

        /* wait until this condition holds: */
        (want[1 - _pid] == 0 || turn == 1 - _pid);

        /* enter */
critical:       skip;
        /* leave */

        want[_pid] = 0;
        goto again
}
```

In the above case, all variables are initialized to zero. The general syntax for declaring and instantiating a variable, respectively for scalar and array variables, is:

```
type name = expression;
type name[constant] = expression
```

In the latter case, all elements of the array are initialized to the value of the expression.

A missing initializer field defaults to the value zero. As usual, multiple variables of the same type can be grouped behind a single type name, as in:

```
byte  a, b[3], c = 4
```

In this example, the variable c is initialized to the value 4; variable a and the elements of array b are all initialized to zero.

Variables can also be declared as structures. For example:

```
typedef Field {
        short f = 3;
        byte  g
};

typedef Msg {
        byte a[3];
        int fld1;
        Field fld2;
        chan p[3];
        bit b
};

Msg foo;
```

introduces two user-defined data types, the first named Field and the second named Msg. A single variable named foo of type Msg is declared. All fields of foo that are not explicitly initialized (in the example, all fields except foo.fld2.f) are initialized to zero. References to the elements of a structure are written as:

```
foo.a[2] = foo.fld2.f + 12
```

A variable of a user-defined type can be passed as a single argument to a new process in run statements. For instance,

```
proctype me(Msg z) {
        z.a[2] = 12
}
init {
        Msg foo;
        run me(foo)
}
```

Note that even though PROMELA supports only one-dimensional arrays, a two-dimensional array can be created indirectly with user-defined structures, for instance as follows:

```
typedef Array {
        byte el[4]
};

Array a[4];
```

This creates a data structure of 16 elements that can be referenced, for instance, as a[i].el[j].

As in C, the indices of an array of N elements range from 0 to N-1.

Expressions

Expressions must be side-effect free in PROMELA. Specifically, this means that an expression cannot contain assignments, or send and receive operations (see section 1.1.3).

```
c = c + 1; c = c - 1
```

and

```
c++; c--
```

are assignments in PROMELA, with the same effects. But, unlike in C,

```
b = c++
```

is not a valid assignment, because the right-hand side operand is not a valid expression in PROMELA (it is not side-effect free).

It is also possible to write a side-effect free conditional expression, with the following syntax:

```
(expr1 -> expr2 : expr3)
```

The parentheses around the conditional expression are required to avoid misinterpretation of the arrow. The example expression has the value of expr2 when expr1 evaluates to a non-zero value, and the value of expr3 otherwise.

In assignments like

```
variable = expression
```

the values of all operands used inside the expression are first cast to signed integers before the operands are applied. After the evaluation of the expression completes, the value produced is cast to the type of the target variable before the assignment takes place.

1.1.3. Message Channels

Message channels are used to model the transfer of data between processes. They are declared either locally or globally, for instance as follows:

```
chan qname = [16] of { short, byte }
```

The keyword chan introduces a channel declaration. In this case, the channel is named qname, and it is declared to be capable of storing up to 16 messages. Each message stored in the channel is declared here to consist of two fields: one of type short and one of type byte. The fields of a message can be any one of the basic types bit, bool, byte, short, int, and chan, or any user-defined type. Message fields cannot be declared as arrays.

A message field of type chan can be used to pass a channel identifier through a channel from one process to another.

The statement

```
qname!expr1,expr2
```

sends the values of expressions expr1 and expr2 to the channel that we just created. It appends the message field created from the values of the two expressions (and cast to the appropriate types of the message fields declared for qname) to the tail of the message buffer of 16 slots that belongs to channel qname. By default the send statement is only executable if the target channel is non-full. (This default semantics can be changed in the verifier into one where the send statement is always executable, but the message will be lost when an attempt is made to append it to a full channel.)

The statement

```
qname?var1,var2
```

retrieves a message from the head of the same buffer, and stores the two expressions in variables var1 and var2.

The receive statement is executable only if the source channel is non-empty.

If more parameters are sent per message than were declared for the message channel, the redundant parameters are lost. If fewer parameters are sent than declared, the value of the remaining parameters is undefined. Similarly, if the receive operation tries to retrieve more parameters than available, the value of the extra parameters is undefined; if it receives fewer than the number of parameters sent, the extra information is lost.

An alternative, and equivalent, notation for the send and receive operations is to structure the message fields with parentheses, as follows:

```
qname!expr1(expr2,expr3)
qname?var1(var2,var3)
```

In the above case, we assume that qname was declared to hold messages consisting of three fields.

Some or all of the arguments of the receive operation can be given as constants instead of as variables:

```
qname?cons1,var2,cons2
```

In this case, an extra condition on the executability of the receive operation is that the value of all message fields specified as constants match the value of the corresponding fields in the message that is to be received.

Here is an example that uses some of the mechanisms introduced so far.

```
proctype A(chan q1)
{       chan q2;
        q1?q2;
        q2!123
}

proctype B(chan qforb)
{       int x;
        qforb?x;
        printf("x = %d\n", x)
}

init {
        chan qname = [1] of { chan };
        chan qforb = [1] of { int };
        run A(qname);
        run B(qforb);
        qname!qforb
}
```

The value printed by the process of type B will be 123.

A predefined function len(qname) returns the number of messages currently stored in channel qname. Two shorthands for the most common uses of this function are empty(qname) and full(qname), with the obvious connotations.

Since all expressions must be side-effect free, it is not valid to say:

```
(qname?var == 0)
```

or

```
(a > b && qname!123)
```

We could rewrite the second example (using an atomic sequence, as explained further in section 1.2.1):

```
atomic { (a > b && !full(qname)) -> qname!123 }
```

The meaning of the first example is ambiguous. It could mean that we want the condition to be true if the receive operation is unexecutable. In that case, we can rewrite it without side-effects as:

```
empty(qname)
```

It could also mean that we want the condition to be true when the channel does contain a message with value zero. We can specify that as follows:

```
atomic { qname?[0] -> qname?var }
```

The first statement of this atomic sequence is an expression without side-effects that evaluates to a non-zero value only if the receive operation

```
qname?0
```

would have been executable at that point (i.e., channel qname contains at least one message and the oldest message stored consists of one message field equal to zero). Any receive statement can be turned into a side-effect free expression by placing square brackets around the list of all message parameters. The channel contents remain undisturbed by the evaluation of such expressions.

Note carefully, however, that in non-atomic sequences of two statements such as

```
!full(qname) -> qname!msgtype
```

and

```
qname?[msgtype] -> qname?msgtype
```

the second statement is not necessarily executable after the first one has been executed. There may be race conditions when access to the channels is shared between several processes. Another process can send a message to the channel just after this process determined that it was not full, or another process can steal away the message just after our process determined its presence.

Two other types of send and receive statements are used less frequently: sorted send and random receive. A sorted send operation is written with two, instead of one, exclamation marks, as follows:

```
qname!!msg
```

A sorted send operation will insert a message into the channels buffer in numerical order, instead of in FIFO order. The channel contents are scanned from the first message towards the last, and the message is inserted immediately before the first message that follows it in numerical order. To determine the numerical order, all message fields are taken into account.

The logical counterpart of the sorted send operation is the random receive. It is written with two, instead of one, question marks:

```
qname??msg
```

A random receive operation is executable if it is executable for *any* message that is currently buffered in a message channel (instead of only for the first message in the channel). Normal send and receive operations can freely be combined with sorted send and random receive operations.

Rendezvous Communication

So far we have talked about asynchronous communication between processes via message channels, declared in statements such as

```
chan qname = [N] of { byte }
```

where N is a positive constant that defines the buffer size. A logical extension is to allow for the declaration

```
chan port = [0] of { byte }
```

to define a rendezvous port. The channel size is zero, that is, the channel port can pass, but cannot store, messages. Message interactions via such rendezvous ports are by definition synchronous. Consider the following example:

```
#define msgtype 33

chan name = [0] of { byte, byte };

active proctype A()
{       name!msgtype(124);
        name!msgtype(121)
}

active proctype B()
{       byte state;
        name?msgtype(state)
}
```

Channel name is a global rendezvous port. The two processes will synchronously execute their first statement: a handshake on message msgtype and a transfer of the value 124 to local variable state. The second statement in process A will be unexecutable, because there is no matching receive operation in process B.

If the channel name is defined with a non-zero buffer capacity, the behavior is different. If the buffer size is at least 2, the process of type A can complete its execution, before its peer even starts. If the buffer size is 1, the sequence of events is as follows. The process of type A can complete its first send action, but it blocks on the second, because the channel is now filled to capacity. The process of type B can then retrieve the first message and complete. At this point A becomes executable again and completes, leaving its last message as a residual in the channel.

Rendezvous communication is binary: only two processes, a sender and a receiver, can be synchronized in a rendezvous handshake.

As the example shows, symbolic constants can be defined with preprocessor macros using #define. The source text of a PROMELA model is translated by the standard C preprocessor. The disadvantage of defining symbolic names in this way is, however, that the PROMELA parser will only see the expanded text, and cannot refer to the symbolic names themselves. To prevent that, PROMELA also supports another way to define symbolic names, which are preserved in error reports. For instance, by including the declaration

```
mtype = { ack, msg, error, data };
```

at the top of a PROMELA model, the names provided between the curly braces are equivalent to integers of type byte, but known by their symbolic names to the SPIN parser and the verifiers it generates. The constant values assigned start at 1, and count up. There can be only one mtype declaration per model.

1.2. Control Flow

So far, we have seen only some of the basic statements of PROMELA, and the way in which they can be combined to model process behaviors. The five types of statements we have mentioned are: printf, assignment, condition, send, and receive.

The pseudo-statement skip is syntactically and semantically equivalent to the condition (1) (i.e., to true), and is in fact quietly replaced with this expression by the lexical analyzer of SPIN.

There are also five types of compound statements.

- Atomic sequences (section 1.2.1),
- Deterministic steps (section 1.2.2),
- Selections (section 1.2.3),
- Repetitions (section 1.2.4),
- Escape sequences (section 1.2.5).

1.2.1. Atomic Sequences

The simplest compound statement is the atomic sequence:

```
atomic {        /* swap the values of a and b */
        tmp = b;
        b = a;
        a = tmp
}
```

In the example, the values of two variables a and b are swapped in a sequence of statement executions that is defined to be uninterruptable. That is, in the interleaving of process executions, no other process can execute statements from the moment that the first statement of this sequence begins to execute until the last one has completed.

It is often useful to use atomic sequences to start a series of processes in such a way that none of them can start executing statements until all of them have been initialized:

```
init {
        atomic {
                run A(1,2);
                run B(2,3);
                run C(3,1)
        }
}
```

Atomic sequences may be non-deterministic. If any statement inside an atomic sequence is found to be unexecutable, however, the atomic chain is broken, and another process can take over control. When the blocking statement becomes executable later, control can non-deterministically return to the process, and the atomic execution of the sequence resumes as if it had not been interrupted.

1.2.2. Deterministic Steps

Another way to define an indivisible sequence of actions is to use the d_step statement. In the above case, for instance, we could also have written:

```
d_step {        /* swap the values of a and b */
        tmp = b;
        b = a;
        a = tmp
}
```

The difference between d_step sequence and an atomic sequence are:

- A d_step sequence must be completely deterministic. (If non-determinism is nonetheless encountered, it is always resolved in a fixed and deterministic way: i.e., the first true guard in selection or repetition structures is always selected.)
- No goto jumps into or out of d_step sequence are permitted.
- The execution of a d_step sequence cannot be interrupted when a blocking statement is encountered. It is an error if any statement other than the first one in

a d_step sequence is found to be unexecutable.

A d_step sequence is executed as one single statement. In a way, it is a mechanism for adding new types of statements to the language.

None of the items listed above apply to atomic sequences. This means that the keyword d_step can always be replaced with the keyword atomic , but the reverse is not true. (The main, perhaps the only, reason for using d_step sequences is to improve the efficiency of verifications.)

1.2.3. Selection Structures

A more interesting construct is the selection structure. Using the relative values of two variables a and b to choose between two options, for instance, we can write:

```
if
:: (a != b) -> option1
:: (a == b) -> option2
fi
```

The selection structure above contains two execution sequences, each preceded by a double colon. Only one sequence from the list will be executed. A sequence can be selected only if its first statement is executable. The first statement is therefore called a *guard* .

In the above example the guards are mutually exclusive, but they need not be. If more than one guard is executable, one of the corresponding sequences is selected nondeterministically. If all guards are unexecutable the process will block until at least one of them can be selected. There is no restriction on the type of statements that can be used as a guard: it may include sends or receives, assignments, printf , skip , etc. The rules of executability determine in each case what the semantics of the complete selection structure will be. The following example, for instance, uses receive statements as guards in a selection.

```
mtype = { a, b };

chan ch = [1] of { byte };

active proctype A()
{       ch!a
}

active proctype B()
{       ch!b
}

active proctype C()
{       if
        :: ch?a
        :: ch?b
        fi
}
```

The example defines three processes and one channel. The first option in the selection structure of the process of type C is executable if the channel contains a message named a, where a is a symbolic constant defined in the mtype declaration at the start of the program. The second option is executable if it contains a message b, where, similarly, b, is a symbolic constant. Which message will be available depends on the unknown relative speeds of the processes.

A process of the following type will either increment or decrement the value of variable count once.

```
byte count;

active proctype counter()
{
        if
        :: count++
        :: count--
        fi
}
```

Assignments are always executable, so the choice made here is truly a non-deterministic one that is independent of the initial value of the variable (zero in this case).

1.2.4. Repetition Structures

We can modify the above program as follows, to obtain a cyclic program that randomly changes the value of the variable up or down, by replacing the selection structure with a repetition.

```
byte count;

active proctype counter()
{
        do
        :: count++
        :: count--
        :: (count == 0) -> break
        od
}
```

Only one option can be selected for execution at a time. After the option completes, the execution of the structure is repeated. The normal way to terminate the repetition structure is with a break statement. In the example, the loop can be broken only when the count reaches zero. Note, however, that it need not terminate since the other two options remain executable. To force termination we could modify the program as follows.

```
active proctype counter()
{
        do
        :: (count != 0) ->
                if
                :: count++
                :: count--
                fi
        :: (count == 0) -> break
        od
}
```

A special type of statement that is useful in selection and repetition structures is the else statement. An else statement becomes executable only if no other statement within the same process, at the same control-flow point, is executable. We could try to use it in two places in the above example:

```
active proctype counter()
{
        do
        :: (count != 0) ->
                if
                :: count++
                :: count--
                :: else
                fi
        :: else -> break
        od
}
```

The first else, inside the nested selection structure, can never become executable though, and is therefore redundant (both alternative guards of the selection are assignments, which are always executable). The second usage of the else, however, becomes executable exactly when !(count != 0) or (count == 0), and is therefore equivalent to the latter to break from the loop.

There is also an alternative way to exit the do-loop, without using a break statement: the infamous goto. This is illustrated in the following implementation of Euclid's algorithm for finding the greatest common divisor of two non-zero, positive numbers:

```
proctype Euclid(int x, y)
{
        do
        :: (x > y) -> x = x - y
        :: (x < y) -> y = y - x
        :: (x == y) -> goto done
        od;
done:
        skip
}

init { run Euclid(36, 12) }
```

The goto in this example jumps to a label named done. Since a label can only appear before a statement, we have added the dummy statement skip. Like a skip, a goto statement is always executable and has no other effect than to change the control-flow point of the process that executes it.

As a final example, consider the following implementation of a Dijkstra semaphore, which is implemented with the help of a synchronous channel.

```
#define p       0
#define v       1

chan sema = [0] of { bit };

active proctype Dijkstra()
{       byte count = 1;

        do
        :: (count == 1) ->
                sema!p; count = 0
        :: (count == 0) ->
                sema?v; count = 1
        od
}
```

```
active [3] proctype user()
{       do
        :: sema?p;
           /* critical section */
           sema!v;
           /* non-critical section */
        od
}
```

The semaphore guarantees that only one of the three user processes can enter its critical section at a time. It does not necessarily prevent the monopolization of the access to the critical section by one of the processes.

PROMELA does not have a mechanism for defining functions or procedures. Where necessary, though, these may be modeled with the help of additional processes. The return value of a function, for instance, can be passed back to the calling process via global variables or messages. The following program illustrates this by recursively calculating the factorial of a number

```
proctype fact(int n; chan p)
{       chan child = [1] of { int };
        int result;

        if
        :: (n <= 1) -> p!1
        :: (n >= 2) ->
                run fact(n-1, child);
                child?result;
                p!n*result
        fi
}

init
{       chan child = [1] of { int };
        int result;

        run fact(7, child);
        child?result;
        printf("result: %d\n", result)
}
```

Each process creates a private channel and uses it to communicate with its direct descendant. There are no input statements in PROMELA. The reason is that models must always be complete to allow for logical verifications, and input statements would leave at least the source of some information unspecified. A way to read input would presuppose a source of information that is not part of the model.

We have already discussed a few special types of statement: skip, break, and else. Another statement in this class is the timeout. The timeout is comparable to a system level else statement: it becomes executable if and only if no other statement in any of the processes is executable. Timeout is a modeling feature that provides for an escape from a potential deadlock state. The timeout takes no parameters, because the types of properties we would like to prove for PROMELA models must be proven independent of all absolute and relative timing considerations. In particular, the relative speeds of processes can never be known with certainty in an asynchronous system.

1.2.5. Escape Sequences

The last type of compound structure to be discussed is the unless statement. It is used as follows:

```
{ P } unless { E }
```

where the letters P and E represent arbitrary PROMELA fragments. Execution of the unless statement begins with the execution of statements from P. Before each statement execution in P the executability of the first statement of E is checked, using the normal PROMELA semantics of executability. Execution of statements from P proceeds only while the first statement of E remains unexecutable. The first time that this guard of the escape sequence is found to be executable, control changes to it, and execution continues as defined for E. Individual statement executions remain indivisible, so control can only change from inside P to the start of E in between individual statement executions. If the guard of the escape sequence does not become executable during the execution of P, then it is skipped entirely when P terminates.

An example of the use of escape sequences is:

```
A;
do
:: b1 -> B1
:: b2 -> B2
...
od
unless { c -> C };
D
```

As shown in the example, the curly braces around the main sequence (or the escape sequence) can be deleted if there can be no confusion about which statements belong to those sequences. In the example, condition c acts as a watchdog on the repetition construct from the main sequence. Note that this is not necessarily equivalent to the construct

```
A;
do
:: b1 -> B1
:: b2 -> B2
...
:: c -> break
od;
C; D
```

if B1 or B2 are non-empty. In the first version of the example, execution of the iteration can be interrupted at *any* point inside each option sequence. In the second version, execution can only be interrupted at the start of the option sequences.

1.3. Correctness Properties

There are three ways to express correctness properties in PROMELA, using:

Assertions (section 1.3.1),
Special labels (section 1.3.2),
Never claims (section 1.3.3).

1.3.1. Assertions

Statements of the form

```
assert(expression)
```

are always executable. If the expression evaluates to a non-zero value (i.e., the corresponding condition holds), the statement has no effect when executed. The correctness property expressed, though, is that it is impossible for the expression to evaluate to zero (i.e., for the condition to be false). A failing assertion will cause execution to be aborted.

1.3.2. Special Labels

Labels in a PROMELA specification ordinarily serve as targets for unconditional goto jumps, as usual. There are, however, also three types of labels that have a special meaning to the verifier. We discuss them in the next three subsections.

1.3.2.1. End-State Labels

When a PROMELA model is checked for reachable deadlock states by the verifier, it must be able to distinguish valid *end state* s from invalid ones. By default, the only valid end states are those in which every PROMELA process that was instantiated has reached the end of its code. Not all PROMELA processes, however, are meant to reach the end of their code. Some may very well linger in a known wait state, or they may sit patiently in a loop ready to spring into action when new input arrives.

To make it clear to the verifier that these alternate end states are also valid, we can define special end-state labels. We can do so, for instance, in the process type Dijkstra , from an earlier example:

```
proctype Dijkstra()
{       byte count = 1;

end:    do
        :: (count == 1) ->
                sema!p; count = 0
        :: (count == 0) ->
                sema?v; count = 1
        od
}
```

The label end defines that it is not an error if, at the end of an execution sequence, a process of this type has not reached its closing curly brace, but waits at the label. Of course, such a state could still be part of a deadlock state, but if so, it is not caused by this particular process.

There may be more than one end-state label per PROMELA model. If so, all labels that occur within the same process body must be unique. The rule is that every label name with the prefix end is taken to be an end-state label.

1.3.2.2. Progress-State Labels

In the same spirit, PROMELA also allows for the definition of progress labels. Passing a progress label during an execution is interpreted as a good thing: the process is not just idling while waiting for things to happen elsewhere, but is making effective progress in its execution. The implicit correctness property expressed here is that any infinite execution cycle allowed by the model that does not pass through at least one of these progress labels is a potential starvation loop. In the Dijkstra example, for instance, we can label the successful passing of a semaphore test as progress and ask a verifier to make sure that there is no cycle elsewhere in the system.

```
proctype Dijkstra()
{       byte count = 1;

end:    do
        :: (count == 1) ->
progress:       sema!p; count = 0
        :: (count == 0) ->
                sema?v; count = 1
        od
}
```

If more than one state carries a progress label, variations with a common prefix are again valid.

1.3.2.3. Accept-State Labels

The last type of label, the accept-state label, is used primarily in combination with `never` claims. Briefly, by labeling a state with any label starting with the prefix `accept` we can ask the verifier to find all cycles that *do* pass through at least one of those labels. The implicit correctness claim is that this cannot happen. The primary place where accept labels are used is inside `never` claims. We discuss `never` claims next.

1.3.3. Never Claims

Up to this point we have talked about the specification of correctness criteria with assertions and with three special types of labels. Powerful types of correctness criteria can already be expressed with these tools, yet so far our only option is to add them to individual `proctype` declarations. We can, for instance, express the claim every system state in which property P is true eventually leads to a system state in which property Q is true, with an extra monitor process, such as:

```
active proctype monitor()
{
progress:
        do
        :: P -> Q
        od
}
```

If we require that property P must *remain* true while we are waiting Q to become true, we can try to change this to:

```
active proctype monitor()
{
progress:
        do
        :: P -> assert(P || Q)
        od
}
```

but this does not quite do the job. Note that we cannot make any assumptions about the relative execution speeds of processes in a PROMELA model. This means that if in the remainder of the system the property P becomes true, we can move to the state just before the `assert`, and wait there for an unknown amount of time (anything between a zero delay and an infinite delay is possible here, since no other synchronizations apply). If Q becomes true, we may pass the assertion, but we need not do so. Even if P becomes false only *after* Q has become true, we may still fail the assertion, as long as there exists some later state where neither P nor Q is true. This is clearly unsatisfactory, and we need another mechanism to express these important types of liveness properties.

The Connection with Temporal Logic

A general way to express system properties of the type we have just discussed is to use linear time temporal logic (LTL) formulae. Every PROMELA expression is automatically also a valid LTL formula. An LTL formula can also contain the unary temporal operators ¡ (pronounced always), Ê (pronounced eventually), and two binary temporal operators U (pronounced weak until) and ***U*** (pronounced strong until).

Where the value of a PROMELA expression without temporal operators can be defined uniquely for individual system states, without further context, the truth value of an LTL formula is defined for sequences of states: specifically, it is defined for the first state of a given infinite sequence of system states (a trace). Given, for instance, the sequence of system states:

```
s0;s1;s2;...
```

the LTL formula pUq, with p and q standard PROMELA expressions, is true for s0 either if q is true in s0, or if p is true in s0 and pUq holds for the remainder of the sequence after s0.

Informally, pUq says that p is required to hold at least until q becomes true. If, instead, we would write p***U***q, then we also require that there exists at least one state in the sequence where q does indeed become true.

The temporal operators ¡ and Ê can be defined in terms of the strong until operator ***U***, as follows.

```
¡ p = !Ê !p = !(true U !p)
```

Informally, ¡ p says that property p must hold in all states of a trace, and Ê p says that p holds in at least one state of the trace.

To express our original example requirement: every system state in which property P is true eventually leads to a system state in which property Q is true, we can write the LTL formula:

```
¡ (P -> Ê Q)
```

where the logical implication symbol -> defined in the usual way as

```
P => Q means !P || Q
```

Mapping LTL Formulae onto Never Claims

PROMELA does not include syntax for specifying LTL formula directly, but it relies on the fact that every such formula can be translated into a special type of automaton, known as a Büchi automaton. In the syntax of PROMELA this automaton is called a never claim. If you don t care too much about the details of never claims, you can skip the remainder of this section and simple remember that SPIN can convert any LTL formula automatically into the proper never claim syntax with the command:

```
spin -f "...formula..."
```

Here are the details. The syntax of a never claim is:

```
never {
        ...
}
```

where the dots can contain any PROMELA fragment, including arbitrary repetition, selection, unless constructs, jumps, etc.

There is an important difference in semantics between a proctype declaration and a never claim. Every statement inside a never claim is interpreted as a proposition, i.e., a condition. A never claim should therefore only contain expressions and never statements that can have side-effects (assignments, sends or receives, run-statements, etc.)

Never claims are used to express behaviors that are considered undesirable or illegal. We say that a never claim is matched if the undesirable behavior can be realized, contrary to the claim, and thus the correctness requirement violated. The claims are evaluated over system executions, that is, the propositions that are listed in the claim are evaluated over the traces from the remainder of the system. The claim, therefore, should not alter that behavior: it merely monitors it. Every time that the system reaches a new state, by asynchronously executing statements from the model, the claim will evaluate the appropriate propositions to determine if a counter-example can be constructed to the implicit LTL formula that is specified.

Since LTL formulae are only defined for infinite executions, the behavior of a never claim can only be matched by an infinite system execution. This by itself would restrict us to the use of progress labels and accept labels as the only means we have discussed so far for expressing properties of infinite behaviors. To conform to standard omega automata theory, the behaviors of never claims are expressed exclusively with accept labels (never with progress labels). To match a claim, therefore, an infinite sequence of true propositions must exist, at least one of which is labeled with an accept label (inside the never claim).

Since PROMELA models can also express terminating system behaviors, we have to define the semantics of the never claims also for those behaviors. To facilitate this, it is defined that a never claim can also be matched when it reaches its closing curly brace (i.e., when it appears to terminate). This semantics is based on what is usually referred to as a stuttering semantics. With stuttering semantics, any terminating execution can be extended into an equivalent infinite execution (for the purposes of evaluating LTL properties) by repeating (stuttering) the final state infinitely often. As a syntactical convenience, the final state of a never claim is defined to be accepting, i.e., it could be replaced with the explicit repetition construct:

```
accept:  do :: skip od
```

Every process behavior, similarly, is (for the purposes of evaluating the never claims) thought to be extended with a dummy self-loop on all final states:

```
         do :: skip od
```

(Note the accept labels only occur in the never claim, not in the system.)

The Semantics of a Never Claim

Never claims are probably the hardest part of the language to understand, so it is worth spending a few extra words on them. On an initial reading, feel free to skip the remainder of this section.

The difference between a never claim and the remainder of a PROMELA system can be explained as follows. A PROMELA model defines an asynchronous interleaving product of the behaviors of individual processes. Given an arbitrary system state, its successor states are conceptually obtained in two steps. In a first step, all the executable statements in the individual processes are identified. In a second step, each one of these statements is executed, each one producing one potential successor for the current state. The complete system behavior is thus defined recursively and represents all possible interleavings of the individual process behaviors. It is this asynchronous product machine that we call the global system behavior.

The addition of a never claim defines a *synchronous* product of the global system behavior with the behavior expressed in the claim. This synchronous product can be thought of as the construction of a new global state machine, in which every state is defined as a pair (s, n) with s a state from the global system (the asynchronous product of processes), and n a state from the claim. Every transition in the new global machine is similarly defined by a pair of transitions, with the first element a statement from the system, and the second a proposition from the claim. In other words, every transition in this final synchronous product is defined as a joint transition of the system and the claim. Of course, that transition can only occur if the proposition from the second half of the transition pair evaluates to true in the current state of the system (the first half of the state pair).

Examples

To manually translate an LTL formula into a never claim (e.g. foregoing the builtin translation that SPIN offers), we must carefully consider whether the formula expresses a positive or a negative property. A positive property expresses a good behavior that we would like our system to have. A negative property expresses a bad behavior that we claim the system does not have. A never claim can express only negative claims, not positive ones. Fortunately, the two are exchangeable: if we want to express that a good behavior is unavoidable, we can formalize all ways in which the good behavior could be violated, and express that in the never claim.

Suppose that the LTL formula Ê¡ p, with p a PROMELA expression, expresses a negative claim (i.e. it is considered a correctness violation if there exists any execution sequence in which p can eventually remain true infinitely long). This can be written in a never claim as:

```
never { /* <>[]p */
        do
        :: skip /* after an arbitrarily long prefix */
        :: p -> break   /* p becomes true */
        od;
accept: do
        :: p    /* and remains true forever after */
        od
}
```

Note that in this case the claim does not terminate, and also does not necessarily match all system behaviors. It is sufficient if it precisely captures all violations of our correctness requirement, and no more.

If the LTL formula expressed a positive property, we first have to invert it to the corresponding negative property Ê !p and translate that into a never claim. The requirement now says that it is a violation if p does not hold infinitely long.

```
never { /* <>!p*/
        do
        :: skip
        :: !p -> break
        od
}
```

We have used the implicit match of a claim upon reaching the closing terminating brace. Since the first violation of the property suffices to disprove it, we could also have written:

```
never  { /*  <>!p*/
         do
         :: p
         :: !p -> break
         od
}
```

or, if we abandon the connection with LTL for a moment, even more tersely as:

```
never  { do :: assert(p)   od }
```

Suppose we wish to express that it is a violation of our correctness requirements if there exists any execution in the system where ¡ (p -> Ê q) is violated (i.e., the negation of this formula is satisfied). The following never claim expresses that property:

```
never  {
         do
         :: skip
         :: p && !q -> break
         od;
accept:
         do
         :: !q
         od
}
```

Note that using (!p || q) instead of skip in the first repetition construct would imply a check for just the first occurrence of proposition p becoming true in the execution sequence, while q is false. The above formalization checks for all occurrences, anywhere in a trace.

Finally, consider a formalization of the LTL property ¡ (p -> (q U r)). The corresponding claim is:

```
never  {
         do
         :: skip           /* to match  any  occurrence   */
         :: p &&  q && !r -> break
         :: p && !q && !r -> goto  error
         od;
         do
         ::  q && !r
         :: !q && !r -> break
         od;
error:   skip
}
```

Note again the use of skip instead of (!p || r) to avoid matching just the first occurrence of (p && !r) in a trace.

1.4. Predefined Variables and Functions

The following predefined variables and functions can be especially useful in never claims.

The predefined variables are: _pid and _last.

_pid is a predefined local variable in each process that holds the unique instantiation number for that process. It is always a non-negative number.

_last is a predefined global variable that always holds the instantiation number of the process that performed the last step in the current execution sequence. Its value is not part of the system state unless it is explicitly used in a specification.

```
never {
        /* it is not possible for the process with pid=1
         * to execute precisely every other step forever
         */
accept:
        do
        :: _last != 1 -> _last == 1
        od
}
```

The initial value of _last is zero.

Three predefined functions are specifically intended to be used in never claims, and may not be used elsewhere in a model: pc_value(pid), enabled(pid), procname[pid]@label.

The function pc_value(pid) returns the current control state of the process with instantiation number pid, or zero if no such process exists.

Example:

```
never {
        /* Whimsical use: claim that it is impossible
         * for process 1 to remain in the same control
         * state as process 2, or one with smaller value.
         */
accept: do
        :: pc_value(1) <= pc_value(2)
        od
}
```

The function enabled(pid) tells whether the process with instantiation number pid has an executable statement that it can execute next.

Example:

```
never {
        /* it is not possible for the process with pid=1
         * to remain enabled without ever executing
         */
accept:
        do
        :: _last != 1 && enabled(1)
        od
}
```

The last function procname[pid]@label tells whether the process with instantiation number pid is currently in the state labeled with label in proctype procname. It is an error if the process referred to is not an instantiation of that proctype.

2. Verifications with SPIN

The easiest way to use SPIN is probably on a Windows terminal with the Tcl/Tk implementation of XSPIN. All functionality of SPIN, however, is accessible from any plain ASCII terminal, and there is something to be said for directly interacting with the tool itself.

The description in this paper gives a short walk-through of a common mode of operation in using the verifier. A more tutorial style description of the verification process can be found in [Ho93]. More detail on the verification of large systems with the help of SPINs supertrace (bitstate) verification algorithm can be found in [Ho95].

Random and interactive simulations (section 2.1),
Generating a verifier (section 2.2),
Compilation for different types of searches (section 2.3),

Performing the verification (section 2.4),
Inspecting error traces produced by the verifier (section 2.5),
Exploiting partial order reductions (section 2.6).

2.1. Random and Interactive Simulations

Given a model in PROMELA, say stored in a file called spec, the easiest mode of operation is to perform a random simulation. For instance,

```
spin -p spec
```

tells SPIN to perform a random simulation, while printing the process moves selected for execution at each step (by default nothing is printed, other than explicit printf statements that appear in the model itself). A range of options exists to make the traces more verbose, e.g., by adding printouts of local variables (add option -l), global variables (add option -g), send statements (add option -s), or receive statements (add option -r). Use option -nN (with N any number) to fix the seed on SPIN's internal random number generator, and thus make the simulation runs reproducible. By default the current time is used to seed the random number generator. For instance:

```
spin -p -l -g -r -s -n1 spec
```

If you don't like the system randomly resolving non-deterministic choices for you, you can select an interactive simulation:

```
spin -i -p spec
```

In this case you will be offered a menu with choices each time the execution could proceed in more than one way.

Simulations, of course, are intended primarily for the debugging of a model. They cannot prove anything about it. Assertions will be evaluated during simulation runs, and any violations that result will be reported, but none of the other correctness requirements can be checked in this way.

2.2. Generating the Verifier

A model-specific verifier is generated as follows:

```
spin -a spec
```

This generates a C program in a number of files (with names starting with pan).

2.3. Compiling the Verifier

At this point it is good to know the physical limitations of the computer system that you will run the verification on. If you know how much physical (not virtual) memory your system has, you can take advantage of that. Initially, you can simply compile the verifier for a straight exhaustive verification run (constituting the strongest type of proof if it can be completed). Compile as follows.

```
pcc -o pan pan.c                # standard exhaustive search
```

If you know a memory bound that you want to restrict the run to (e.g., to avoid paging), find the nearest power of 2 (e.g., 23 for the bound 2^{23} bytes) and compile as follows.

```
pcc '-DMEMCNT=23' -o pan pan.c
```

or equivalently in terms of MegaBytes:

```
pcc '-DMEMLIM=8' -o pan pan.c
```

If the verifier runs out of memory before completing its task, you can decide to increase the bound or to switch to a frugal supertrace verification. In the latter case, compile as follows.

```
pcc -DBITSTATE -o pan pan.c
```

2.4. Performing the Verification

There are three specific decisions to make to perform verifications optimally: estimating the size of the reachable state space (section 2.4.1), estimating the maximum length of a unique execution sequence (2.4.2), and selecting the type of correctness property (2.4.3). No great harm is done if the estimates from the first two steps are off. The feedback from the verifier usually provides enough clues to determine quickly what the optimal settings for peak performance should be.

2.4.1. Reachable States

For a standard exhaustive run, you can override the default choice for the size for the hash table (2^{18} slots) with option -w. For instance,

```
pan -w23
```

selects 2^{23} slots. The hash table size should optimally be roughly equal to the number of reachable states you expect (within say a factor of two or three). Too large a number merely wastes memory, too low a number wastes CPU time, but neither can affect the correctness of the result.

For a supertrace run, the hash table *is* the memory arena, and you can override the default of 2^{22} bits with any other number. Set it to the maximum size of physical memory you can grab without making the system page, again within a factor of say two or three. Use, for instance -w23 if you expect 8 million reachable states and have access to at least 8 million (2^{23}) bits of memory (i.e., 2^{20} or 1 Megabyte of RAM).

2.4.2. Search Depth

By default the analyzers have a search depth restriction of 10,000 steps. If this isnt enough, the search will truncate at 9,999 steps (watch for it in the printout). Define a different search depth with the -m flag.

```
pan -m100000
```

If you exceed also this limit, it is probably good to take some time to consider if the model you have specified is indeed finite. Check, for instance, if no unbounded number of processes is created. If satisfied that the model is finite, increase the search depth at least as far as is required to avoid truncation completely.

If you find a particularly nasty error that takes a large number of steps to hit, you may also set lower search depths to find the shortest variant of an error sequence.

```
pan -m40
```

Go up or down by powers of two until you find the place where the error first appears or disappears and then home in on the first depth where the error becomes apparent, and use the error trail of that verification run for guided simulation.

Note that if a run with a given search depth fails to find an error, this does not necessarily mean that no violation of a correctness requirement is possible within that number of steps. The verifier performs its search for errors by using a standard depth-first graph search. If the search is truncated at N steps, and a state at level N-1 happens to be reachable also within fewer steps from the initial state, the second time it is reached it will not be explored again, and thus neither will its successors. Those successors may contain errors states that are reachable within N steps from the initial state. Normally, the verification should be run in such a way that no execution paths can be truncated, but to force the complete exploration of also truncated searches one can override the defaults with a compile-time flag -DREACH . When the verifier is compiled with that additional directive, the depth at which each state is visited is remembered, and a state is now considered unvisited if it is revisited via a shorter path later in the search. (This option cannot be used with a supertrace search.)

2.4.3. Liveness or Safety Verification

For the last, and perhaps the most critical, runtime decision, it must be decided if the system is to be checked for safety violations or for liveness violations.

```
pan -l   # search for non-progress cycles
pan -a   # search for acceptance cycles
```

(In the first case, though, you must compile pan.c with -DNP as an additional directive. If you forget, the executable will remind you.) If you don't use either of the above two options, the default types of correctness properties are checked (assertion violations, completeness, race conditions, etc.) Note that the use of a never claim that contains accept labels requires the use of the -a flag for complete verification.

Adding option -f restricts the search for liveness properties further under a standard *weak fairness* constraint:

```
pan -f -l      # search for weakly fair non-progress cycles
pan -f -a      # search for weakly fair acceptance cycles
```

With this constraint, each process is required to appear infinitely often in the infinite trace that constitutes the violation of a liveness property (e.g., a non-progress cycle or an acceptance cycle), unless it is permanently blocked (i.e., has no executable statements after a certain point in the trace is reached). Adding the fairness constraint increases the time complexity of the verification by a factor that is linear in the number of active processes.

By default, the verifier will report on unreachable code in the model only when a verification run is successfully completed. This default behavior can be turned off with the runtime option -n , as in:

```
pan -n -f -a
```

(The order in which the options such as these are listed is always irrelevant.) A brief explanation of these and other runtime options can be determined by typing:

```
pan --
```

2.5. Inspecting Error Traces

If the verification run reports an error, any error, SPIN dumps an error trail into a file named spec.trail , where spec is the name of your original PROMELA file. To inspect the trail, and determine the cause of the error, you must use the guided simulation option. For instance:

```
spin -t -c spec
```

gives you a summary of message exchanges in the trail, or

```
spin -t -p spec
```

gives a printout of every single step executed. Add as many extra or different options as you need to pin down the error:

```
spin -t -r -s -l -g spec
```

Make sure the file spec didn't change since you generated the analyzer from it.

If you find non-progress cycles, add or delete progress labels and repeat the verification until you are content that you have found what you were looking for.

If you are not interested in the first error reported, use pan option -c to report on specific others:

```
pan -c3
```

ignores the first two errors and reports on the third one that is discovered. If you just want to count all errors and not see them, use

```
pan -c0
```

State Assignments

Internally, the verifiers produced by SPIN deal with a formalization of a PROMELA model in terms of extended finite state machines. SPIN therefore assigns state numbers to all statements in the model. The state numbers are listed in all the relevant output to make it completely unambiguous (source line references unfortunately do not have that property). To confirm the precise state assignments, there is a runtime option to the analyzer generated:

```
pan -d    # print state machines
```

which will print out a table with all state assignments for each proctype in the model.

2.6. Exploiting Partial Order Reductions

The search algorithm used by SPIN is optimized according to the rules of a partial order theory explained in [HoPe94]. The effect of the reduction, however, can be increased considerably if the verifier has extra information about the access of processes to global message channels. For this purpose, there are two keywords in the language that allow one to assert that specific channels are used exclusively by specific processes. For example, the assertions

```
xr q1;
xs q2;
```

claim that the process that executes them is the *only* process that will receive messages from channel q1, and the *only* process that will send messages to channel q2.

If an exclusive usage assertion turns out to be invalid, the verifier will be able to detect this, and report it as a violation of an implicit correctness requirement.

Every read or write access to a message channel can introduce new dependencies that may diminish the maximum effect of the partial order reduction strategies. If, for instance, a process uses the len function to check the number of messages stored in a channel, this counts as a read access, which can in some cases invalidate an exclusive access pattern that might otherwise exist. There are two special functions that can be used to poll the size of a channel in a safe way that is compatible with the reduction strategy.

The expression nfull(qname) returns true if channel qname is not full, and nempty(qname) returns true if channel qname contains at least one message. Note that the parser will not recognize the free form expressions !full(qname) and !empty(qname) as equally safe, and it will forbid constructions such as !nfull(qname) or !nempty(qname). More detail on this aspect of the reduction algorithms can be found in [HoPe94].

Keywords

For reference, the following table contains all the keywords, predefined functions, predefined variables, and special label-prefixes of the language PROMELA, and refers to the section of this paper in which they were discussed.

_last (1.4)	_pid (1.1.1)	accept (1.3.2)	active (1.1.1)
assert (1.3.1)	atomic (1.2.1)	bit (1.1.2)	bool (1.1.2)
break (1.2.4)	byte (1.1.2)	chan (1.1.3)	d_step (1.2.2)
do (1.2.4)	else (1.2.4)	empty (1.1.3)	enabled (1.4)
end (1.3.2)	fi (1.2.3)	full (1.1.3)	goto (1.2.2)
hidden (not discussed)	if (1.2.3)	init (1.1.1)	int (1.1.2)
len (1.1.3)	mtype (1.1.3)	nempty (2.6)	never (1.3.3)
nfull (2.6)	od (1.2.4)	of (1.1.3)	pc_value (1.4)
printf (1.1.1)	proctype (1.1.1)	progress (1.3.2)	run (1.1.1)
short (1.1.2)	skip (1.2)	timeout (1.2.4)	typedef (1.1.2)
unless (1.2.5)	xr (2.6)	xs (2.6)	

References

[Ho91] G.J. Holzmann, *Design and Validation of Computer Protocols,* Prentice Hall, 1991.

[Ho93] G.J. Holzmann, Tutorial: Design and Validation of Protocols, *Computer Networks and ISDN Systems,* 1993, Vol. 25, No. 9, pp. 981-1017.

[HoPe94] G.J. Holzmann and D.A. Peled, An improvement in formal verification, *Proc. 7th Int. Conf. on Formal Description Techniques,* FORTE94, Berne, Switzerland. October 1994.

[Ho95] G.J. Holzmann, An Analysis of Bitstate Hashing, technical report 2/95, available from author.

[HS99] G.J. Holzmann, Software model checking: extracting verification models from source code, *Proc. Formal Methods in Software Engineering and Distributed Systems,* PSTV/FORTE99, Beijng, China, Oct. 1999, Kluwer, pp. 481-497.

The Various Ports

This document collects comments about the various architectures supported by Plan 9. The system tries to hide most of the differences between machines, so the machines as seen by a Plan 9 user look different from how they are perceived through commercial software. Also, because we are a small group, we couldn't do everything: exploit every optimization, support every model, drive every device. This document records what we *have* done. The first section discusses the compiler/assembler/loader suite for each machine. The second talks about the operating system implemented on each of the various machines.

The MIPS compiler

This compiler generates code for the R2000, R3000, and R4000 machines configured to be big-endians. The compiler generates no R4000-specific instructions although the assembler and loader support the new user-mode instructions. There are options to generate code for little-endian machines. Considering its speed, the Plan 9 compiler generates good code, but the commercial MIPS compiler with all the stops pulled out consistently beats it by 20% or so, sometimes more. Since ours compiles about 10 times faster and we spend most of our time compiling anyway, we are content with the tradeoff.

The compiler is solid: we've used it for several big projects and, of course, all our applications run under it. The behavior of floating-point programs is much like on the 68040: the operating system emulates where necessary to get past non-trapping underflow and overflow, but does not handle gradual underflow or denormalized numbers or not-a-numbers.

The SPARC compiler

The SPARC compiler is also solid and fast, although we haven't used it for a few years, due to a lack of current hardware. We have seen it do much better than GCC with all the optimizations, but on average it is probably about the same.

We used to run some old SPARC machines with no multiply or divide instructions, so the compiler does not produce them by default. Instead it calls internal subroutines. A loader flag, -M, causes the instructions to be emitted. The operating system has trap code to emulate them if necessary, but the traps are slower than emulating them in user mode. In any modern lab, in which SPARCS have the instructions, it would be worth enabling the -M flag by default.

The floating point story is the same as on the MIPS.

The Intel i386 compiler

This is really an *x*86 compiler, for *x*>2. It works only if the machine is in 32-bit protected mode. It is solid and generates tolerable code; it is our main compiler these days.

Floating point is well-behaved, but the compiler assumes i387-compatible hardware to execute the instructions. With 387 hardware, the system does the full IEEE 754 job, just like the MC68881. By default, the libraries don't use the 387 built-ins for transcendentals. If you want them, build the code in /sys/src/libc/386/387.

The AMD64 compiler

The AMD64 compiler has been used to build 64-bit variants of Plan 9. It seems to be reasonably solid.

The PowerPC compiler

The PowerPC compiler supports the 32-bit PowerPC architecture only; it does not support either the 64-bit extensions or the POWER compatibility instructions. It has been used for production operating system work on the 603, 603e, 604e, 821, 823, and 860, and experimental work on the 405, 440 and 450. On the 8xx floating-point instructions must be emulated. Instruction scheduling is not implemented; otherwise the code generated is similar to that for the other load-store architectures. The compiler makes little or no use of unusual PowerPC features such as the counter register, several condition code registers, and multiply-accumulate instructions, but they are sometimes used by assembly language routines in the libraries.

The PowerPC64 compiler

The PowerPC64 compiler supports the 64-bit PowerPC architecture only. It has been lightly used on IBM's Blue Gene machines.

The ARM compiler

The ARM compiler is fairly solid; it has been used for some production operating system work including Inferno and the Plan 9 kernel for the iPAQ, which uses a StrongArm SA1, and the Sheevaplug, Guruplug, Dreamplug, Gumstix Overo, Compulab Trimslice and others. The compiler supports the ARMv4 and later 32-bit architectures; it does not support the Thumb instruction sets. It has been used on ARM7500FE, ARM926 and Cortex-A8 and -A9 processors and the Strongarm SA1 core machines. The compiler generates instructions for ARM 7500 FPA floating-point coprocessor 1 by default, but 5l -f instead generates VFP instructions for coprocessors 10 and 11.

The IBM PC operating system

The PC version of Plan 9 can boot via PXE or directly from a disk created by the format command; see *prep* (8). Plan 9 runs in 32-bit mode which requires a 386 or later model x86 processor and has an interrupt-driven I/O system, so it does not use the BIOS (except for a small portion of the boot program and floppy boot block). This helps performance but limits the set of I/O devices that it can support without special code.

Plan 9 supports the ISA, EISA, and PCI buses as well as PCMCIA and PC card devices. It is infeasible to list all the supported machines, because the PC-clone marketplace is too volatile and there is no guarantee that the machine you buy today will contain the same components as the one you bought yesterday. (For our lab, we buy components and assemble the machines ourselves in an attempt to lessen this effect.) IDE/ATA, SATA and SCSI disks are supported. CD-ROMs are supported two ways, either on the SCSI bus, or as ATA(PI) devices. The SCSI adapter must be a member of the Mylex Multimaster (old Buslogic BT-*) series or the Symbios 53C8XX series.

Supported Ethernet cards include the AMD79C790, 3COM Etherlink III and 3C589 series, Lucent Wavelan and compatibles, NE2000, WD8003, WD8013, Realtek 8139, SMC Elite and Elite Ultra, Linksys Combo Ethernet Card and EtherFast 10/100, and a variety of controllers based on the Intel 8255[789] and Digital (now Intel) 21114x chips. We support Gigabit Ethernet via Realtek 8110S/8169S, and Intel 8254[013467], 8256[36], and 8257[1-79] controllers. We support 10-Gigabit Ethernet via Intel's 8259[89], and Myricom's 10g-pcie-8a. We mostly use Intel and Realtek gigabit controllers, so those drivers may be more robust.

There must be an explicit Plan 9 driver for peripherals; it cannot use DOS or Windows drivers. Plan 9 cannot exploit special hardware-related features that fall outside of the IBM PC model, such as power management, unless architecture-dependent code is added to the kernel. For more details see *plan9.ini* (8).

Over the years, Plan 9 has run on a number of VGA cards. Recent changes to the graphics system have not been tested on most of the older cards; some effort may be needed to get them working again. In our lab, most of our machines use the ATI or Nvidia chips, so such devices are probably the most reliable. The system requires a hardware cursor. For more details see *vgadb* (6) and *vga* (8). The wiki (http://plan9.bell-labs.com/wiki/plan9) contains the definitive list of cards that are known to work; see the "supported PC hardware" page.

For audio, Plan 9 supports the Sound Blaster 16 and compatibles. (Note that audio doesn't work under Plan 9 with 8-bit Sound Blasters.) There is also user-level support for USB audio devices; see *usb* (4).

Finally, it's important to have a three-button mouse with Plan 9. The system currently works only with mice on the PS/2 port or USB. Serial mouse support should return before long.

Once you have Plan 9 installed (see the wiki's installation document), use PXE or a boot disk to load the system. See *booting* (8), *9boot* (8), and *prep* (8) for more information.

The Routerboard 450G operating system

This is a CPU kernel that runs on the Mikrotik Routerboard RB450G, which contains a MIPS 24K CPU (the Atheros 7161), which implements the MIPS32R2 architecture. It has 256MB of RAM and a serial port. The CPU lacks the 64-bit instructions of previous MIPS systems (e.g., SGI Challenge and Carrera). There is no hardware floating-point, so we emulate the instructions. Only the first of the five Gigabit Ethernet ports is currently supported; the other four are connected via an internal switch. To avoid a bug in the CPU (erratum 48), we run the caches write-through, rather than write-back, and compiled /mips with a *vl* modified to emit enough NOPs to avoid three consecutive store instructions (see /sys/src/cmd/vl/noop.c to enable this).

The PowerPC operating system

We have a version of the system that runs on the PowerPC on a home-grown machine called Viaduct. The Viaduct minibrick is a small (12x9x3 cm) low-cost embedded computer consisting of a 50Mhz MPC850, 16MB sdram, 2MB flash, and two 10Mb Ethernet ports. It is designed for home/SOHO networking applications such as VPN, firewalls, NAT, etc.

The kernel has also been ported to the Motorola MTX embedded motherboard; that port is included in the distribution. The port only works with a 604e processor (the 603e is substantially different) and at present only a single CPU is permitted.

We have ports to the Xilinx Virtex 4 and 5 FPGAs which use PowerPC 405 and 440 processors, respectively.

The Marvell Kirkwood operating system

This is an ARM kernel for the ARM926EJ-S processor and it emulates ARM 7500 floating-point and CAS (compare-and-swap) instructions. It is known to run on the Sheevaplug, Guruplug, Dreamplug and Openrd-client boards. It is derived from a port of native Inferno to the Sheevaplug by Salva Peiró and Mechiel Lukkien. There are many features of the Kirkwood system-on-a-chip that it does not exploit. There are currently drivers for up to two Gigabit Ethernet interfaces, USB and the console serial port; we hope to add crypto acceleration, and a video driver for the Openrd-client.

The Marvell PXA168 operating system

This is an ARM kernel for the ARM-v5-architecture processor in the Marvell PXA168 system-on-a-chip and it emulates ARM 7500 floating-point and CAS (compare-and-swap) instructions. It is known to run on the Guruplug Display. There are many features of the system-on-a-chip that it does not exploit. There are currently drivers for a Fast Ethernet interface, and the console serial port; we hope to add crypto acceleration, and a video driver.

The TI OMAP35 operating system

This is an ARM kernel for the Cortex-A8 processor and it emulates ARM 7500 floating-point and CAS (compare-and-swap) instructions. It is known to run on the IGEPv2 board and the Gumstix Overo, and might eventually run on the Beagleboard, once USB is working. There are many features of the OMAP system-on-a-chip that it does not exploit. Initially, there are drivers for the SMSC 9221 100Mb/s Ethernet interface in the IGEPv2 and Overo, and the console serial port; we hope to add USB, flash memory and video drivers.

The Nvidia Tegra2 operating system

This is an ARM kernel for the dual Cortex-A9 processors in the Nvidia Tegra2 system-on-a-chip and it emulates ARM 7500 floating-point and CAS (compare-and-swap) instructions, but the hardware includes VFP3 floating-point. It runs on the Compulab Trimslice. There are many features of the system-on-a-chip that it does not exploit. Initially, there are drivers for the Ethernet interface and the console serial port; we hope to add USB, flash memory and video drivers.

The Broadcom 2835 operating system

This consists of terminal and CPU kernels for the ARM1176 processor in the Broadcom 2835 system-on-a-chip. The hardware includes VFP2 floating-point. It runs on the Raspberry Pi Models A and B. Since it relies upon USB Ethernet and the Plan 9 USB Ethernet driver doesn't implement multicast, this port can't speak IPv6.

This book was typeset by the authors; the input text was characters from the Unicode Standard encoded in UTF-8.

The fonts used were Lucida Sans, in a special version incorporating over 1700 characters from the Unicode Standard, along with Lucida Sans Italic, Lucida Sans DemiBold, and Lucida Typewriter, designed by Bigelow & Holmes, Atherton, California. The hinted Adobe Type 1 representation of the fonts was provided by Y&Y Inc., 45 Walden Street, Concord, MA, 01742, USA.