Plan 9: The Early Papers

Rob Pike
Dave Presotto
Ken Thompson
Howard Trickey
Tom Duff
Gerard Holzmann

{rob|presotto|ken|howard|td|gerard}@research.att.com
AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This report reprints half a dozen early but still current papers on Plan 9 from Bell Labs, a distributed computing system being developed at the Computing Science Research Center of AT&T Bell Laboratories.

The papers were all presented at conferences; the original citations were:

Rob Pike, Dave Presotto, Ken Thompson, and Howard Trickey, ‘‘Plan 9 from Bell Labs’’, Proc. of the Summer 1990 UKUUG Conf., London, July, 1990, pp. 1-9

Dave Presotto, Rob Pike, Ken Thompson, and Howard Trickey, ‘‘Plan 9, A Distributed System’’, Proc. of the Spring 1991 EurOpen Conf., Troms, May, 1991, pp. 43-50

Rob Pike, ‘‘8½, the Plan 9 Window System’’, Proc. of the Summer 1991 USENIX Conf., Nashville, June, 1991, pp. 257-265

Dave Presotto, ‘‘Multiprocessor Streams for Plan 9’’, Proc. of the Summer 1990 UKUUG Conf., London, July, 1990, pp. 11-19

Rob Pike, Dave Presotto, Ken Thompson, and Gerard Holzmann, ‘‘Process Sleep and Wakeup on a Shared-Memory Multiprocessor’’, Proc. of the Spring 1991 EurOpen Conf., Troms, May, 1991, pp. 161-166

Tom Duff, ‘‘Rc - A Shell for Plan 9 and UNIX Systems’’, Proc. of the Summer 1990 UKUUG Conf., London, July, 1990, pp. 21-33

Ken Thompson, ‘‘A New C Compiler’’, Proc. of the Summer 1990 UKUUG Conf., London, July, 1990, pp. 41-51

Plan 9 from Bell Labs

Rob Pike
Dave Presotto
Ken Thompson
Howard Trickey

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Plan 9 is a distributed computing environment. It is assembled from separate machines acting as CPU servers, file servers, and terminals. The pieces are connected by a single file-oriented protocol and local name space operations. By building the system from distinct, specialised components rather than from similar general-purpose components, Plan 9 achieves levels of efficiency, security, simplicity, and reliability seldom realised in other distributed systems. This paper discusses the building blocks, interconnections, and conventions of Plan 9.

Introduction.

Plan 9 is a general-purpose, multi-user, portable distributed system implemented on a variety of computers and networks. It lacks a number of features often found in other distributed systems, including

(i) A uniform distributed name space,
(ii) Process migration,
(iii) Lightweight processes,
(iv) Distributed file caching,
(v) Personalised workstations,
(vi) Support for X windows.

Unhappy with the trends in commercial systems, we began a few years ago to design a system that could adapt well to changes in computing hardware. In particular, we wanted to build a system that could profit from continuing improvements in personal machines with bitmap graphics, in medium- and high-speed networks, and in high-performance microprocessors. A common approach is to connect a group of small personal timesharing systems—workstations—by a medium-speed network, but this has a number of failings. Because each workstation has private data, each must be administered separately; maintenance is difficult to centralise. The machines are replaced every couple of years to take advantage of technological improvements, rendering the hardware obsolete often before it has been paid for. Most telling, a workstation is a largely self-contained system, not specialised to any particular task, too slow and I/O-bound for fast compilation, too expensive to be used just to run a window system. For our purposes, primarily software development, it seemed that an approach based on distributed specialisation rather than compromise could better address issues of cost-effectiveness, maintenance, performance, reliability, and security. We decided to build a completely new system, including compiler, operating system, networking software, command interpreter, window system, and (on the hardware side) terminal. This construction would also offer an occasion to rethink, revisit, and perhaps even replace most of the utilities we had accumulated over the years.

Plan 9 is divided along lines of service function. CPU servers concentrate computing power into large (not overloaded) multiprocessors; file servers provide repositories for storage; and terminals give each user of the system a dedicated computer with bitmap screen and mouse on which to run a window system. The sharing of computing and file storage services provides a sense of community for a group of programmers,

amortises costs, and centralises and hence simplifies management and administration.

The pieces communicate by a single protocol, built above a reliable data transport layer offered by an appropriate network, that defines each service as a rooted tree of files. Even for services not usually considered as files, the unified design permits some noteworthy and profitable simplification. Each process has a local file *name space* that contains attachments to all services the process is using and thereby to the files in those services. One of the most important jobs of a terminal is to support its user's customised view of the entire system as represented by the services visible in the name space.

To be used effectively, the system requires a CPU server, a file server, and a terminal; it is intended to provide service at the level of a departmental computer centre or larger. The CPU server and file server are large machines best housed in an air conditioned machine room with conditioned power. The system's strengths stem in part from economies of scale, and the scale we have in mind is large. One of our goals, perhaps unrealisable, is to unite the computing environment for all of AT&T Bell Laboratories (about 30,000 people) into a single Plan 9 system comprising thousands of CPU and file servers spread throughout, and clustered in, the company's various departments. That is clearly beyond the administrative capacity of workstations on Ethernets.

The following sections describe the basic components of Plan 9, explain the name space and how it is used, and offer some examples of unusual services that illustrate how the ideas of Plan 9 can be applied to a variety of problems.

CPU Servers

Several computers provide CPU service for Plan 9. The production CPU server is a Silicon Graphics Power Series machine with four 25MHz MIPS processors, 128 megabytes of memory, no disk, and a 20 megabyte-per-second back-to-back DMA connection to the file server. It also has Datakit and Ethernet controllers to connect to terminals and non-Plan 9 systems. [1, 2] The operating system provides a conventional view of processes, based on `fork` and `exec` system calls, [3] and of files, mostly determined by the remote file server. Once a connection to the CPU server is established, the user may begin typing commands to a command interpreter in a conventional-looking environment. [4, 5]

A multiprocessor CPU server has several advantages. The most important is its ability to absorb load. If the machine is not saturated (which can be economically feasible for a multiprocessor) there is usually a free processor ready to run a new process. This is similar to the notion of free disk blocks in which to store new files on a file system. The comparison extends farther: just as one might buy a new disk when a file system gets full, one may add processors to a multiprocessor when the system gets busy, without needing to replace or duplicate the entire system. Of course, one may also add new CPU servers and share the file servers.

The CPU server performs compilation, text processing, and other applications. It has no local storage; all the permanent files it accesses are provided by remote servers. Transient parts of the name space, such as the collected images of active processes [6] or services provided by user processes, may reside locally but these disappear when the CPU server is rebooted. Plan 9 CPU servers are as interchangeable for their task—computation— as are ordinary terminals for theirs.

File Servers

The Plan 9 file servers hold all permanent files. The current server is another Silicon Graphics computer with two processors, 64 megabytes of memory, 600 megabytes of magnetic disk, and a 300 gigabyte jukebox of write-once optical disk (WORM). (This machine is to be replaced by a MIPS 6280, a single processor with much greater I/O bandwidth.) It connects to Plan 9 CPU servers through 20 megabyte-per-second DMA links, and to terminals and other machines through conventional networks.

The file server presents to its clients a file system rather than, say, an array of disks or blocks or files. The files are named by slash-separated components that label branches of a tree, and may be addressed for I/O at the byte level. The location of a file in the server is invisible to the client. The true file system resides on the WORM, and is accessed through a two-level cache of magnetic disk and RAM. The contents of recently-used files reside in RAM and are sent to the CPU server rapidly by DMA over a high-speed link, which is much faster than regular disk although not as fast as local memory. The magnetic disk acts as a

cache for the WORM and simultaneously as a backup medium for the RAM. With the high-speed links, it is unnecessary for clients to cache data; instead the file server centralises the caching for all its clients, avoiding the problems of distributed caches.

The file server actually presents several file systems. One, the ‘‘main’’ system, is used as the file system for most clients. Other systems provide less generally-used data for private applications. One service is unusual: the backup system. Once a day, the file server freezes activity on the main file system and flushes the data in that system to the WORM. Normal file service continues unaffected, but changes to files are applied to a fresh hierarchy, fabricated on demand, using a copy-on-write scheme. [7] Thus, the file tree is split into two: a read-only version representing the system at the time of the dump, and an ordinary system that continues to provide normal service. The roots of these old file trees are available as directories in a file system that may be accessed exactly as any other (read-only) system. For example, the file `/usr/rob/doc/plan9.ms` as it existed on April 1, 1990, can be accessed through the backup file system by the name `/1990/0401/usr/rob/doc/plan9.ms`. This scheme permits recovery or comparison of lost files by traditional commands such as file copy and comparison routines rather than by special utilities in a backup subsystem. Moreover, the backup system is provided by the same file server and the same mechanism as the original files so permissions in the backup system are identical to those in the main system; one cannot use the backup data to subvert security.

Terminals

The standard terminal for Plan 9 is a Gnot (with silent ’G’), a locally-designed machine of which several hundred have been manufactured. The terminal’s hardware is reminiscent of a diskless workstation: 4 or 8 megabytes of memory, a 25MHz 68020 processor, a 1024×1024 pixel display with two bits per pixel, a keyboard, and a mouse. It has no external storage and no expansion bus; it is a terminal, not a workstation. A 2 megabit per second packet-switched distribution network connects the terminals to the CPU and file servers. Although the bandwidth is low for applications such as compilation, it is more than adequate for the terminal’s intended purpose: to provide a window system, that is, a multiplexed interface to the rest of Plan 9.

Unlike a workstation, the Gnot does not handle compilation; that is done by the CPU server. The terminal runs a version of the CPU server’s operating system, configured for a single, smaller processor with support for bitmap graphics, and uses that to run programs such as a window system and a text editor. Files are provided by the standard file server over the terminal’s network connection.

Just like old character terminals, all Gnots are equivalent, as they have no private storage either locally or on the file server. They are inexpensive enough that every member of our research centre can have two: one at work and one at home. A person working on a Gnot at home sees exactly the same system as at work, as all the files and computing resources remain at work where they can be shared and maintained effectively.

Networks

Plan 9 has a variety of networks that connect the components. CPU servers and file servers communicate over back-to-back DMA controllers. That is only practical for the scale of, say, a computer centre or departmental computing resource. More distant machines are connected by traditional networks such as Ethernet or Datakit. A terminal or CPU server may use a remote file server completely transparently except for performance considerations. As our Datakit network spans the country, Plan 9 systems could be assembled on a large scale, although this has not been tried in practice. (See Figure 1.)

To keep their cost down, Gnots employ an inexpensive network that uses standard telephone wire and a single-chip interface. (The throughput is respectable, about 120 kilobytes per second.)

To get even that bandwidth to home is of course problematic. Some of us have DS-1 lines at 1.54 megabits per second; others are experimenting with more modest communications equipment. Since the terminal only mediates communication—it instructs the CPU server to connect to the file server but does not participate in the resulting communication —the relatively low bandwidth to the terminal does not affect the overall performance of the system.

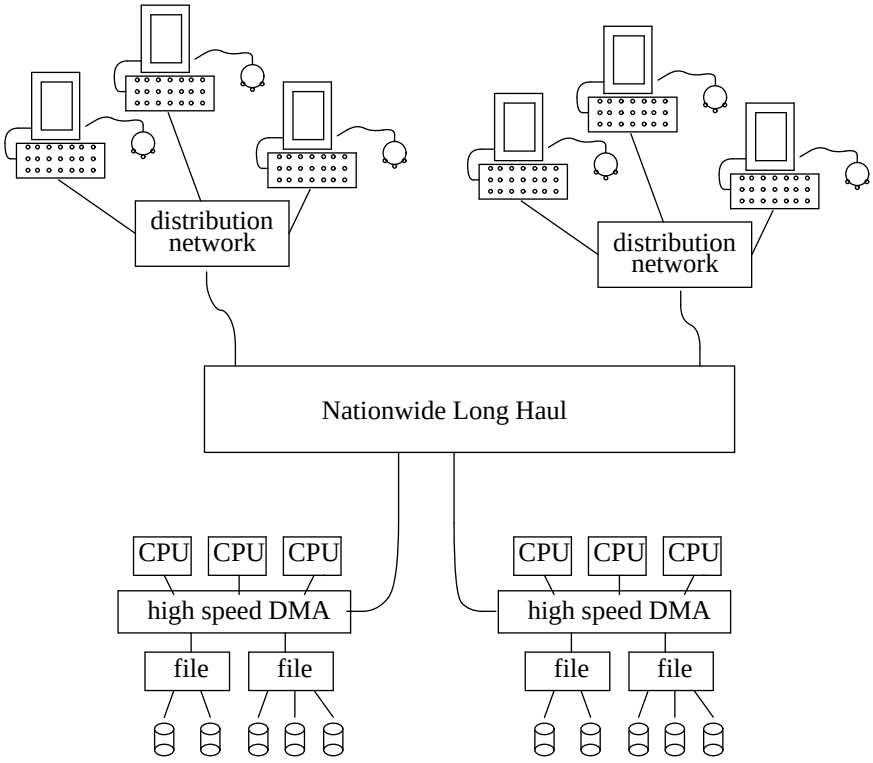

Figure 1 - Plan 9 Topology

Name Spaces

There are two kinds of name space in Plan 9: the global space of the names of the various servers on the network and the local space of files and servers visible to a process. Names of machines and services connected to Datakit are hierarchical, for example `nj/mh/astro/helix`, defining (roughly) the area, building, department, and machine in a department. [1] Because the network provides naming for its machines, global naming issues need not be handled directly by Plan 9. However one of Plan 9's fundamental operations is to attach network services to the local name space on a per-process basis. This fine-grained control of the local name space is used to address issues of customisability, transparency, and heterogeneity.

The protocol for communicating with Plan 9 services is file-oriented; all services must implement a file system. That is, each service, local or remote, is arranged into a set of file-like objects collected into a hierarchy called the name space of the server. For a file server, this is a trivial requirement. Other services must sometimes be more imaginative. For instance, a printing service might be implemented as a directory in which processes create files to be printed. Other examples are described in the following sections; for the moment, consider just a set of ordinary file servers distributed around the network.

When a program calls a Plan 9 service (using mechanisms inherent in the network and outside Plan 9 itself) the program is connected to the root of the name space of the service. Using the protocol, usually as mediated by the local operating system into a set of file-oriented system calls, the program accesses the service by opening, creating, removing, reading, and writing files in the name space.

From the set of services available on the network, a user of Plan 9 selects those desired: a file server where personal files reside, perhaps other file servers where data is kept, or a departmental file server where the software for a group project is being written. The name spaces of these various services are collected and joined to the user's own private name space by a fundamental Plan 9 operator, called *attach*, that joins a service's name space to a user's. The user's name space is formed by the union of the spaces of the services being used. The local name space is assembled by the local operating system for each user, typically

by the terminal. The name space is modifiable on a per-process level, although in practice the name space is assembled at log-in time and shared by all that user's processes.

To log in to the system, a user sits at a terminal and instructs it which file server to connect to. The terminal calls the server, authenticates the user (see below), and loads the operating system from the server. It then reads a file, called the *profile*, in the user's personal directory. The profile contains commands that define what services are to be used by default and where in the local name space they are to be attached. For example, the main file server to be used is attached to the root of the local name space, `/`, and the process file system is attached to the directory `/proc`. The profile then typically starts the window system.

Within each window in the window system runs a command interpreter that may be used to execute commands locally, using file names interpreted in the name space assembled by the profile. For computation-intensive applications such as compilation, the user runs a command `cpu` that selects (automatically or by name) a CPU server to run commands. After typing `cpu`, the user sees a regular prompt from the command interpreter. But that command interpreter is running on the CPU server *in the same name space—even the same current directory—as the* `cpu` *command itself.* The terminal exports a description of the name space to the CPU server, which then assembles an identical name space, so the customised view of the system assembled by the terminal is the same as that seen on the CPU server. (A description of the name space is used rather than the name space itself so the CPU server may use high-speed links when possible rather than requiring intervention by the terminal.) The `cpu` command affects only the performance of subsequent commands; it has nothing to do with the services available or how they are accessed.

Although there is a large catalogue of services available in Plan 9, including the service that finds services, a few suffice to illustrate the usage and possibilities of this design.

The Process File System

An example of a local service is the 'process file system', which permits examination and debugging of executing processes through a file-oriented interface. It is related to Killian's process file system [6] but its differences exemplify the way that Plan 9 services are constructed.

The root of the process file system is conventionally attached to the directory `/proc`. (Convention is important in Plan 9; although the name space may be assembled willy-nilly, many programs have conventional names built in that require the name space to have a certain form. It doesn't matter which server the program `/bin/rc` (the command interpreter) comes from but it must have that name to be accessible by the commands that call on it.) After attachment, the directory `/proc` itself contains one subdirectory for each local process in the system, with name equal to the numerical unique identifier of that process. (Processes running on the remote CPU server may also be made visible; this will be discussed below.) Each subdirectory contains a set of files that implement the view of that process. For example, `/proc/77/mem` contains an image of the virtual memory of process number 77. That file is closely related to the files in Killian's process file system, but unlike Killian's, Plan 9's `/proc` implements other functions through other files rather than through peculiar operations applied to a single file. Here is a list of the files provided for each process.

`mem` The virtual memory of the process image. Offsets in the file correspond to virtual addresses in the process.

`ctl` Control behaviour of the processes. Messages sent (by a `write` system call) to this file cause the process to stop, terminate, resume execution, etc.

`text` The file from which the program originated. This is typically used by a debugger to examine the symbol table of the target process, but is in all respects except name the original file; thus one may type '`/proc/77/text`' to the command interpreter to instantiate the program afresh.

`note` Any process with suitable permissions may write the `note` file of another process to send it an asynchronous message for interprocess communication. The system also uses this file to send (poisoned) messages when a process misbehaves, for example divides by zero.

`status` A fixed-format ASCII representation of the status of the process. It includes the name of the file the process was executed from, the CPU time it has consumed, its current state, etc.

The `status` file illustrates how heterogeneity and portability can be handled by a file server model for

system functions. The command `cat /proc/*/status` presents (readably but somewhat clumsily) the status of all processes in the system; in fact the process status command `ps` is just a reformatting of the ASCII text so gathered. The source for `ps` is a page long and is completely portable across machines. Even when `/proc` contains files for processes on several heterogeneous machines, the same implementation works.

Whether the functions provided by the `ctl` file should instead be accessed through further files— `stop`, `terminate`, etc.—is a matter of taste. We chose to fold all the true control operations into the `ctl` file and provide the more data-intensive functions through separate files.

It is worth noting that the services `/proc` provides, although varied, do not strain the notion of a process as a file. For example, it is not possible to terminate a process by attempting to remove its process file nor is it possible to start a new process by creating a process file. The files give an active view of the processes, but they do not literally represent them. This distinction is important when designing services as file systems.

The Window System

In Plan 9, user programs, as well as specialised stand-alone servers, may provide file service. The window system is an example of such a program; one of Plan 9's most unusual aspects is that the window system is implemented as a user-level file server.

The window system is a server that presents a file `/dev/cons`, similar to the `/dev/tty` or `CON:` of other systems, to the client processes running in its windows. Because it controls all I/O activities on that file, it can arrange that each window's group of processes sees a private `/dev/cons`. When a new window is made, the window system allocates a new `/dev/cons` file, puts it in a new name space (otherwise the same as its own) for the new client, and begins a client process in that window. That process connects the standard input and output channels to `/dev/cons` using the normal file opening system call and executes a command interpreter. When the command interpreter prints a prompt, it will therefore be written to `/dev/cons` and appear in the appropriate window.

It is instructive to compare this structure to other operating systems. Most operating systems provide a file like `/dev/cons` that is an alias for the terminal connected to a process. A process that opens the special file accesses the terminal it is running on without knowing the terminal's precise name. Since the alias is usually provided by special arrangement in the operating system, it can be difficult for a window system to guarantee that its client processes can access their window through this file. This issue is handled easily in Plan 9 by inverting the problem. A set of processes in a window shares a name space and in particular `/dev/cons`, so by multiplexing `/dev/cons` and forcing all textual input and output to go through that file the window system can simulate the expected properties of the file.

The window system serves several files, all conventionally attached to the directory of I/O devices, `/dev`. These include `cons`, the port for ASCII I/O; `mouse`, a file that reports the position of the mouse; and `bitblt`, which may be written messages to execute bitmap graphics primitives. Much as the different `cons` files keep separate clients' output in separate windows, the `mouse` and `bitblt` files are implemented by the window system in a way that keeps the various clients independent. For example, when a client process in a window writes a message (to the `bitblt` file) to clear the screen, the window system clears only that window. All graphics sent to partially or totally obscured windows is maintained as a bitmap layer, in memory private to the window system. [8] The clients are oblivious of one another.

Since the window system is implemented entirely at user level with file and name space operations, it can be run recursively: it may be a client of itself. The window system functions by opening the files `/dev/cons`, `/dev/bitblt`, etc., as provided by the operating system, and reproduces—multiplexes—their functionality among its clients. Therefore, if a fresh instantiation of the window system is run in a window, it will behave normally, multiplexing *its* `/dev/cons` and other files for *its* clients. This recursion can be used profitably to debug a new window system in a window or to multiplex the connection to a CPU server. [9] Since the window system has no bitmap graphics code—all its graphics operations are executed by writing standard messages to a file— the window system may be run on any machine that has `/dev/bitblt` in its name space, including the CPU server.

`CPU` Command

The `cpu` command connects from a terminal to a CPU server using a full-duplex network connection and runs a setup process there. The terminal and CPU processes exchange information about the user and name space, and then the terminal-resident process becomes a user-level file server that makes the terminal's private files visible from the CPU server. (At the time of writing, the CPU server builds the name space by re-executing the user's profile; a version being designed will export the name space using a special terminal-resident server that can be queried to recover the terminal's name space.) The CPU process makes a few adjustments to the name space, such as making the file `/dev/cons` on the CPU server *be the same file as on the terminal*, perhaps making both the local and remote process file system visible in `/proc`, and begins a command interpreter. The command interpreter then reads commands from, and prints results on, its file `/dev/cons`, which is connected through the terminal process to the appropriate window (for example) on the terminal. Graphics programs such as bitmap editors also may be executed on the CPU server since their definition is entirely based on I/O to files 'served' by the terminal for the CPU server. The connection to the CPU server and back again is utterly transparent.

This connection raises the issue of heterogeneity: the CPU server and the terminal may be, and in the current system are, different types of processors. There are two distinct problems: binary data and executable code. Binary data can be handled two ways: by making it not binary or by strictly defining the format of the data at the byte level. The former is exemplified by the `status` file in `/proc`, which enables programs to examine, transparently and portably, the status of remote processes. Another example is the file, provided by the terminal's operating system, `/dev/time`. This is a fixed-format ASCII representation of the number of seconds since the epoch that serves as a time base for `make` and other programs. [3] Processes on the CPU server get their time base from the terminal, thereby obviating problems of distributed clocks.

For files that are I/O intensive, such as `/dev/bitblt`, the overhead of an ASCII interface can be prohibitive. In Plan 9, such files therefore accept a binary format in which the byte order is predefined, and programs that access the files use portable libraries that make no assumptions about the order. Thus `/dev/bitblt` is usable from any machine, not just the terminal. This principle is used throughout Plan 9. For instance, the format of the compilers' object files and libraries is similarly defined, which means that object files are independent of the type of the CPU that compiled them.

Having different formats of executable binaries is a thornier problem, and Plan 9 solves it adequately if not gracefully. Directories of executable binaries are named appropriately: `/mips/bin`, `/68020/bin`, etc., and a program may ascertain, through a special server, what CPU type it is running on. A program, in particular the `cpu` command, may therefore attach the appropriate directory to the conventional name `/bin` so that when a program runs, say, `/bin/rc`, the appropriate file is found. Although this is a fairly clumsy solution, it works well in practice. The various object files and compilers use distinct formats and naming conventions, which makes cross-compilation painless, at least once automated by `make` or a similar program. [3]

Security

Plan 9 does not address security issues directly, but some of its aspects are relevant to the topic. Breaking the file server away from the CPU server enhances the possibilities for security. As the file server is a separate machine that can only be accessed over the network by the standard protocol, and therefore can only serve files, it cannot run programs. Many security issues are resolved by the simple observation that the CPU server and file server communicate using a rigorously controlled interface through which it is impossible to gain special privileges.

Of course, certain administrative functions must be performed on the file server, but these are available only through a special command interface accessible only on the console and hence subject to physical security. Moreover, that interface is for administration only. For example, it permits making backups and creating and removing files, but it does not permit reading files or changing their permissions. *The contents of a file with read permission for only its owner will not be divulged by the file server to any other user, even the administrator.*

Of course, this begs the question of how a user proves who he or she is. At the moment, we use a simple

authentication manager on the Datakit network itself, so that when a user logs in from a terminal, the network assures the authenticity of the maker of calls from the associated terminal. In order to remove the need for trust in our local network, we plan to replace the authentication manager by a Kerberos-like system. [10]

Discussion

A fairly complete version of Plan 9 was built in 1987 and 1988, and then its development was abandoned for a number of aesthetic and technical reasons. In May of 1989 work was begun on a completely new system, based on the SGI MIPS-based multiprocessors, using the first version as a bootstrap environment. By October, the CPU server could compile all its own software, using the first-draft file server. The SGI file server came on line in February 1990; the true operating system kernel at its core was taken from the CPU server's system, but the file server is otherwise a completely separate program (and computer). The CPU server's system was ported to the 68020 in 13 hours elapsed time on November 12-13, 1989. One portability bug was found; the fix affected two lines of code. At the time of writing (April 1990), work has just begun on the new window system; it should be running well before this paper appears (July 1990). (Until it is complete, we will continue to use the terminal software from the 1987-1988 implementation.)

All the authors use Plan 9 almost exclusively; only the lack of an electronic mail facility, which is being addressed, prevents us from moving over permanently. Plan 9 is up and running and comfortable to use, although it is certainly too early to pass final judgement.

The multiprocessor operating system for the MIPS-based CPU server has 454 lines of assembly language, more than half of which saves and restores registers on interrupts. The kernel proper contains 3647 lines of C plus 774 lines of header files, which includes all process control, virtual memory support, trap handling, and so on. There are 1020 lines of code to interface to the 29 system calls. Much of the functionality of the system is contained in the 'drivers' that implement built-in servers such as `/proc`; these and the network software add another 9511 lines of code. Most of this code is identical on the 68020 version; for instance, all the code to implement processes, including the process switcher and the `fork` and `exec` system calls, is identical in the two versions; the peculiar properties of each processor are encapsulated in two five-line assembler routines. (The code for the respective MMU's is quite different, although the page fault handler is substantially the same.) It is only fair to admit, however, that the compilers for the two machines are closely related, and the operating system may depend on properties of the compiler in unknown ways.

The system is efficient. On the four-processor machine connected to the MIPS file server, the 45 source files of the operating system compile in about ten seconds of real time and load in another ten. (The loader runs single-threaded.) Partly due to the register-saving convention of the compiler, the null system call takes only 7 microseconds on the MIPS, about half of which is attributed to relatively slow memory on the multiprocessor. A process fork takes 700 microseconds irrespective of the process's size.

Plan 9 does not implement lightweight processes explicitly. We are uneasy about deciding where on the continuum from fine-grained hardware-supported parallelism to the usual timesharing notion of a process we should provide support for user multiprocessing. Existing definitions of threads and lightweight processes seem arbitrary and raise more questions than they resolve. [11] We prefer to have a single kind of process and to permit multiple processes to share their address space. With the ability to share local memory and with efficient process creation and switching, both of which are in Plan 9, we can match the functionality of threads without taking a stand on how users should multiprocess.

Process migration is also deliberately absent from Plan 9. Although Plan 9 makes it easy to instantiate processes where they can most effectively run, it does nothing explicit to make this happen. The compiler, for instance, does not arrange that it run on the CPU server. We prefer to do coarse-grained allocation of computing resources simply by running each new command interpreter on a lightly-loaded CPU server. Reasonable management of computing resources renders process migration unnecessary.

Other aspects of the system lead to other efficiencies. A large single-threaded chess database problem runs about four times as fast on Plan 9 as on the same machine running commercial software because the remote cache on the file server is so large. In general, most file I/O is done by direct DMA from the file server's cache; the file server rarely needs to read from disk at all.

Much of Plan 9 is straightforward. The individual pieces that make it up are relatively ordinary; its unusual

aspects are in how the pieces are put together. As a case in point, the recent interest in using X terminals connected to timeshared hosts might seem to be similar in spirit to how Plan 9 terminals are used, but that is a mistaken impression. The Gnot, although similar in hardware power to a typical X terminal, serves a much higher-level function in the computing environment. It is a fully programmable computer running a virtual memory operating system that maintains its user's view of the entire Plan 9 system. It offloads from the CPU server all the bookkeeping and I/O intensive chores that a window system must perform. It is not really a workstation either; for example one would rarely bother to compile on the Gnot, although one would certainly run a text editor there. Like the other pieces of Plan 9, the Gnot's strength derives from careful specialisation in concert with other specialised components.

Acknowledgements

Many people helped build the system. We would like especially to thank Bart Locanthi, who built the Gnot and encouraged us to program it; Tom Duff, who wrote the command interpreter `rc`, Tom Killian and Ted Kowalski, who cheerfully endured early versions of the software; and Dennis Ritchie, who frequently provided us with much-needed wisdom.

References

1. A. G. Fraser, ''Datakit – A Modular Network for Synchronous and Asynchronous Traffic,'' *Proc. Int. Conf. on Commun.*, Boston, MA (June 1980).
2. R. M Metcalfe and D. R. Boggs, *The Ethernet Local Network: Three Reports*, XEROX Palo Alto Research Center (February 1980).
3. Brian W. Kernighan and Rob Pike, *The UNIX Programming Environment*, Prentice-Hall, Englewood Cliffs, NJ (1984).
4. T. Duff, ''Rc – A Shell for Plan 9 and UNIX,'' *UNIX Programmer's Manual, Tenth Edition*, Murray Hill, NJ, AT&T Bell Laboratories (1990).
5. D. M. Ritchie and K. Thompson, ''The UNIX Time-Sharing System,'' *Comm. Assoc. Comp. Mach.* **17**(7), pp. 365-375 (July 1974).
6. T. J. Killian, ''Processes as Files,'' *USENIX Summer Conference Proceedings*, Salt Lake City, UT, USA (June 1984).
7. S. Quinlan, ''A Cached WORM File System,'' *Software – Practice and Experience*, p. To appear.
8. R. Pike, ''Graphics in Overlapping Bitmap Layers,'' *Transactions on Graphics* **2**(2), pp. 135-160.
9. R. Pike, ''A Concurrent Window System,'' *Computing Systems* **2**(2), pp. 133-153.
10. S. P. Miller, B. C. Neumann, J. I. Schiller, and J. H. Saltzer, *Kerberos Authentication and Authorization System*, MIT (1987).
11. M. J. Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young, ''Mach: A New Kernel Foundation for UNIX Development,'' *USENIX Conference Proceedings*, Atlanta, GA (July, 1986).

Plan 9, A Distributed System

Dave Presotto
Rob Pike
Ken Thompson
Howard Trickey

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Plan 9 is a computing environment physically distributed across many machines. The distribution itself is transparent to most programs giving both users and administrators wide latitude in configuring the topology of the environment. Two properties make this possible: a per process group name space and uniform access to all resources by representing them as files.

1. Introduction

Plan 9 is a general-purpose, multi-user, portable distributed system implemented on a variety of computers and networks. Because commands, libraries, and system calls are similar to those of the Unix operating system, it is possible to port many Unix programs to Plan 9 with little or no changes. A casual user would find little difference between the two systems.

What distinguishes Plan 9 is its organization. The goals of this organization were to reduce administration and to promote resource sharing. Our programming style was minimalism. We believe that a small number of well-chosen abstractions can, with much less code, provide most of the function of a larger system. This is the approach that made the Unix operating system so much smaller than its contemporaries such as Multics. In building Plan 9, we generalized proven ideas from the Unix operating system rather than add new untried concepts.

Plan 9 is divided along lines of service function. Diskless CPU servers concentrate computing power into large multiprocessors; file servers provide repositories for storage; and terminals give each user of the system a dedicated computer with bitmap screen and mouse on which to run a window system. The sharing of computing and file storage services provides a sense of community for a group of programmers, amortizes costs, and centralizes and hence simplifies management and administration.

Since both CPU servers and terminals use the same kernel, users may choose whether to run programs locally on their terminals or remotely on CPU servers. Plan 9 provides this flexibility without constraining the choice. Therefore, both users and administrators can configure their environment to be as distributed or centralized as they wish. At work, users tend to use their terminals more like workstations running all interactive programs locally and reserving the CPU servers for data or compute intensive jobs such as compiling and computing chess end games. At home, connected via a dedicated 9600 baud line to work, users choose what they run locally and remotely to reduce communication cost. Some applications, such as the editor [Pik87], are split into multiple programs to make this choice even more flexible.

Figure 1 in any Plan 9 paper shows how we have configured our environment. Multiprocessor CPU and file servers are clustered in a few computer rooms and connected via 7 megabyte/sec point-to-point links [Pre88]. This permits the CPU servers to be used as high performance compute engines without becoming starved for data. Terminals are connected to the servers via lower speed, lower cost distribution networks such as the 10 megabit Ethernet [Met80] and 2 megabit Incon [Kal, Res]. By emphasizing the shared service clusters we can quickly and cheaply incorporate new technologies as they arise. At the same

time, users wishing more autonomy can incorporate as much computing power as they wish in their own offices without losing the advantage of transparently sharing other resources.

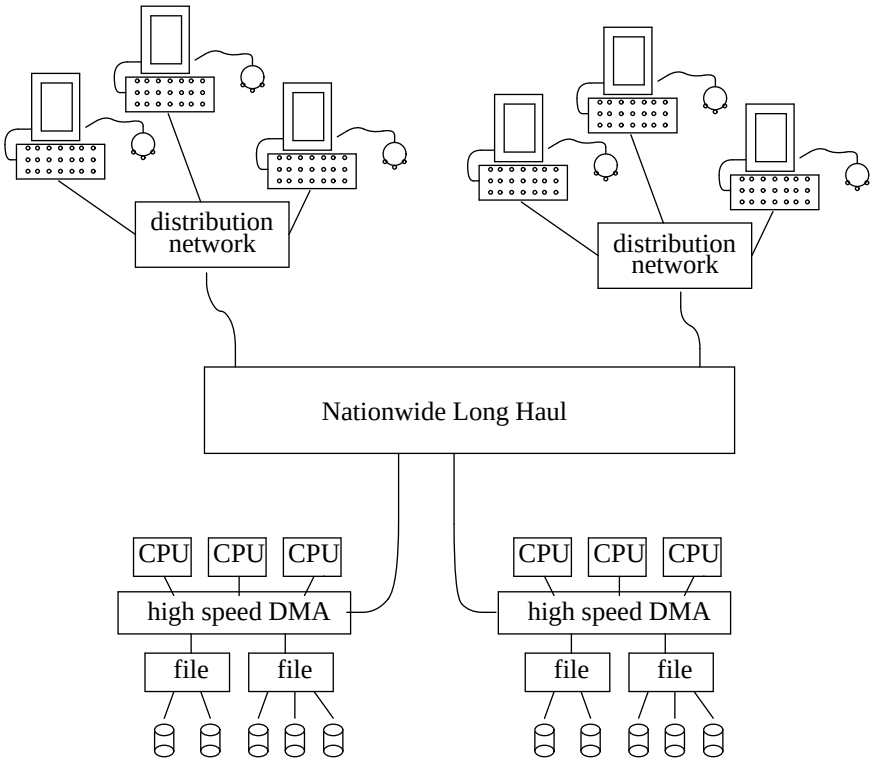

Figure 1 - Plan 9 Topology

The rest of this paper describes the features of Plan 9 that make possible such a flexible topology. For more information on hardware and use of the system, see our previous paper [Pik90] . For details of the file server, see [Qui] .

2. Minimalism

All resources that a process can access, aside from program memory, reside in one name space and are accessed uniformly. Simply stated, all resources are implemented to look like file systems and, henceforth, we shall call them file systems. A file system is a strict tree with no links. File systems can be the traditional type representing persistent storage on a disk as implemented by the shared file servers. They can also represent physical devices such as terminals or complex abstractions such as processes. The file systems can be implemented by kernel resident drivers, by user level processes, or by remote servers.

A file system representing a physical device normally contains one or two files. For example, an RS232 line is represented as a directory containing a `data` and a `ctl` file. The `data` file is the stream of bytes transmitted/received on the line. The `ctl` file is a control channel used to change device parameters such as baud rate.†

Some file systems represent software concepts. Environment variables (as in Unix) are implemented as files in a kernel resident file system. Even processes themselves are represented as directories with separate files representing different aspects of the process such as memory, text file, and control. Many things that require a system call in other operating systems are represented by I/O operations on files in Plan 9;

† We neither need nor have an `ioctl` system call.

reading the id of a process, the user id associated with a process, the time, etc.

A kernel data structure, called a *channel*, is used as a pointer to a file. A user level file descriptor is just a handle for a kernel channel. All I/O system calls eventually translate into nine primitive operations on channels. They are:

attach – point a channel to the root of a file system. The file system is told which user is attaching.

clone – make a copy of a channel. The new channel points to the same file as the old one.

walk – do a one level directory lookup on the channel and point it to the new file (or directory).

stat – get the attributes of the file pointed to.

wstat – change the attributes of the file pointed to.

open – check permissions prior to I/O on the channel.

read – read from the opened file.

write – write to the opened file.

close – close the opened file.

Each kernel resident file system is implemented by a *device driver* containing a procedure for each primitive operation. The device drivers are accessed indirectly via a kernel array, `devtab`, which contains 9 pointers per driver, one to each primitive procedure. Each channel contains an offset into `devtab` indicating the driver to be used in accessing the file it points to.

Accessing file systems not resident in the kernel is via a special device driver, the *mount driver*. All channels pointing to this driver contain a pointer to a communication channel. The mount driver turns operations on such channels into request messages written to the communication channel. The mount driver is written as a multiplexor allowing multiple outstanding messages. Because the messages on the communication channel are transmitted using `read's` and `write's`, any type of channel can be used: a pipe to a process, a network connection, even an RS232 line. The `mount` system call, described below, is used to create a new mount device channel and supply a communication channel for it.

All Plan 9 components are connected using this file system protocol. The code used to encapsulate the primitives into request and reply messages is 580 lines long. The mount driver is 899 lines long. Compared to the equivalent NFS code implementing vnodes and XDR this is tiny.

Of the 18000 lines of code that make up Plan 9, about 5000 lines perform memory management, process management, hardware interface, and system calls. The rest are for the 17 different file systems implementing devices, networks, process control, etc. Since most of the file systems are completely self contained, the complexity of the kernel code is even lower than its 18000 lines would imply. A working, albeit not very useful, kernel can be configured containing only the file systems implementing pipes, a local root, and a console. This totals 5899 lines of commented C code (counted using `wc *.[ch]`). As a comparison, Mach's micro-kernel without device drivers has 25530 lines of C code (calculated, we're told, by counting semi-colons). By the same metric our minimal kernel is only 4622 lines long, less than 1/5 the size. In fact, our kernel with every file system included is still less than half the size of their micro-kernel.

One might note the similarities between `devtab` and parts of the Unix operating system; the block device switch, character device switch, file system switch and vnodes. One advantage of Plan 9 is that we have recognized that these are all essentially the same mechanism and have implemented them as such.

3. Virtual Name Space

When a user boots a terminal or connects to a cpu server, a new process group is created for her processes. This process group starts with an initial name space that provides at minimum a root (`/`), some binaries for the processor the process is running on (`/bin/*`), and some local devices (`/dev/*`). The processes in the group can then either add to or rearrange their name space using two systems calls, `mount` and `bind`. The mount call is used to attach a new (not kernel resident) file system to a point in the name space. Its syntax is

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

where *fd* is a file descriptor for a communication stream such as a pipe or a network connection and *old* is

the name of an existing file in the current name space where the file system will be attached. The attachment creates a new mount device channel whose communication channel is that referred to by *fd.* Subsequent accesses to *old* and any files below it in the hierarchy become request messages written to the communication stream.

The bind call is used to attach a kernel resident file system to the name space and also to rearrange pieces of the name space. Its syntax is

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

where *new* is a name in the current name space† and *old* is the same as in mount.

How the attachment works depends on the *flags* specified in the call. One possibility is that the old file is replaced by the new one. However, when both files are directories, Plan 9 allows another possibility. The result can be the union of the two directories. The effect is that of putting one directory behind the other. In the case of name conflicts for files contained in the directories, the one in front wins. *Flags* specifies whether the new directory replaces, goes in front of, or goes behind the old one. This concept is essentially the same as the search paths used in the Unix libraries and the various shells. In fact, Plan 9 has no search paths and uses these *union directories* in their place. When a command is executed, Plan 9 uses the directory `/bin` the same way Unix uses an execution path.

The ability to specify the complete name space for a process that contains all resources the process can access forms the basis for a true virtual machine. Any aspect of a process' world can be rearranged. Remote objects can be substituted for local ones. Processes can implement part or all of the name space of other processes. This capability is the basis for a number of important services, three of which we present here.

3.1. The Cpu Command

We consider the shared CPU servers as accelerators for our terminals, someplace where commands can run while maintaining the same environment. It is important that as little as possible change when running on the CPU server. The virtual name space provides us with a means to make the CPU servers actually feel this way to our users. A command, `cpu,` calls a CPU server across a network. A daemon process on the server answers the call, creates a new process group for the caller, sets up a name space, and starts a shell process in the new process group. The name space set up is an analogue of the name space of the calling process on the terminal. In particular, local resources on the terminal, such as the screen and the mouse, become visible to the server processes at the same place in the name space as on the terminal. The standard input, standard output, standard error, and current directory of the cpu command become those of the remote shell. The directories mounted on `/bin` are changed to be those that contain executables for the CPU server's processor type (the terminal may be a 68020 while a CPU server could be a MIPS). In general, a user typing the cpu command just notices that things such as compilations speed up while graphics operations slow down.

After the initial handshake to pass information describing the caller's environment, the cpu command becomes a file server answering file system requests from the network connection. The server daemon mounts the network connection to the terminal in a standard place, `/mnt/term,` and then binds the resources it decides to keep into the same places in the new name space. For example, it binds `/mnt/term/dev/mouse` onto `/dev/mouse, /mnt/term/dev/bitblt` onto `/dev/bitblt,` etc. Subsequent accesses to those files are converted by the mount driver in the CPU server into file system messages sent to the terminal.

---

† Local kernel resources are referred to by a syntactic escape (hack) in the name space. Any name starting with a ''#'' refers to a local resource. The first character following the ''#'' specifies the type of resource and the remaining characters are a parameter specifying the instance of the resource. Thus, to bind the local console to a standard place in the name space, one would use `bind("#c", "/dev", FRONT).`

3.2. The Window System

The user interface is made up of three files:

`/dev/bitblt` – writes represent bitblt operations to the screen

`/dev/mouse` – reads return mouse events, i.e., button clicks and movement

`/dev/cons` – reads return keyboard input, writes put characters to the screen.

Between them, these devices represent all I/O to the user. The window system, 8.5 [Pik91], offers processes a multiplexed view to these devices. When a window is opened, the window system starts a new process group for a command (usually a shell) that will run in that window. In that process group's name space, the window system mounts a pipe to itself in front of `/dev.` Subsequent references by the new process group to any of these devices are sent as file system messages to the window server. 8.5 interprets those requests as accesses of the window instead of the whole screen. Similarly, 8.5 multiplexes the mouse and the keyboard so that mouse and keyboard input is available to processes only when their window is selected.

The result is that any program written to use the kernel resident user interface will also work inside a window. Because this is also true of the window system itself, new versions of the window system can be run and debugged in windows of the current window system.

3.3. Network Gateways

One, sometimes insurmountable, problem is accessing a network to which a system is not physically attached. For example, a system may be connected to our Datakit [Fra80] network but not to the DoD Internet. Many gateways exist that try to solve this problem by performing protocol to protocol translation. Unfortunately, few transport protocols have completely equivalent concepts. In order to perform the best translation, it is be necessary to know the semantics requested by the program. For example, TP4 has message delimiters but TCP does not. A protocol translator going from TCP to TP4 would not know which bytes correspond to a single write by the sender.

In Plan 9, every network interface is a file system. A gateway is a file server that serves its own network interfaces to other machines. A process that wants to get at a remote network connects to the gateway and mounts the gateway's interface to the remote network into its name space. Whenever the process accesses the interface, the mount driver will send the request to the gateway. Thus, the gateway sees exactly what the process does.

4. File Caching

In building our environment, we've been reluctant to add local disk file systems to any of our terminals or CPU servers. There are essentially two reasons for this choice. The first is administration. Anyone with a local disk must administer it. Any disk that has unique long term state requires both knowledge and time to administer. In fact, the Bell Labs computer center at Murray Hill is doing a lucrative business maintaining other peoples' disked Sun workstations because the owners have neither the time nor the experience necessary to do it themselves.

The second reason is sharing. Although most workstations can export access to their local file systems, when left up to individual users, this rarely happens. Terminals become personified and users become tied to a particular room to do their work.

Plan 9 survives without local disk file systems thanks partially to hardware and partially to caching. The CPU servers do so because their links to the file servers transfer at a substantial percentage of memory speed. The file servers maintain large main memory caches for their disk file systems. These servers are configured with 128 megabytes or more of main memory to ensure that there is plenty of room for cache. Getting a file from a file server is generally faster than it would be to get it from a local disk.

Office terminals are connected to the file servers by shared 1 or 10 megabit/sec links. Home terminals use 9600 or 19200 baud links. In both cases, the link is much slower than access to a local disk would be. To avoid the obvious performance hit, we use caching. To keep the caches coherent, we use file identifiers supplied by the file server. The identifiers are unique 64 bit quantities. 32 bits identify the file, the other 32 bits identify the version of the file. The version number is incremented each time the file is

modified. Each time a file is opened the file server returns the identifier with the reply. Therefore, it is possible to guarantee coherency at each opening of a file.

Office terminals only cache pages of executable files. Whenever a program terminates, its unmodified text and data pages are not immediately freed. Instead they are retained until the space is required by other programs. When a program is rerun its executable file is reopened and the current version number returned. If the version number has not changed and pages remain from the last run, they are reused. If the version number has changed, any remaining pages of the stale version are discarded. Since most data intensive work is done on the CPU servers, this simple cache saves most of the traffic between office terminals and the file servers. Other caching could be helpful but would require much more complexity.

This cache might also have sufficed for home terminals if it were persistent, but it is not. Therefore, we have added disks to our home terminals to be used as write through caches of the file server files. As a write through cache, it contains no state that isn't duplicated on the file servers. Therefore, it needs little maintenance compared to a local file system. If the code discovers a disk problem, it reformats the disk discarding the current contents. If the user should suspect that the cache is contaminated, she can request that it be reformatted at the next boot. The system slows down until subsequent use refills the cache but no information is lost. The user need not consciously update the disk because the cache uses file identifiers to maintain coherency with the file servers. Each time a file is opened, the cache discards any stale data it might have for that file. The user doesn't have to copy what she needs to the disk because it is done as a consequence of her using the data.

The disk based cache is implemented by a process that resides between the kernel and the file server connection. For every read request, the process satisfies as much as it can with data cached on the disk. It gets the rest from the file server. Any new data that passes through it is saved on the disk. When the cache fills up the least recently used file is discarded. The amount of data cached for any one file is limited to 1.75 megabytes to prevent one file from displacing all others.

Because the disk based cache only checks for coherency when a file is opened, it provides slightly different semantics than that seen on office terminals which do not cache data files. This looser coherency constraint forces programs that communicate via files to ensure an open between each transaction. Thus far we have not had to change any programs because of it.

5. Conclusion

We have presented a distributed system that is simple in structure and flexible in its use. Both the flexibility and simplicity are the result of two properties, a per process group name space and a single resource interface. Coupled with some minimal caching we provide a simple system that is as usable at home as at work.

6. Acknowledgements

Many people helped build the system. We would like especially to thank Bart Locanthi, who built our terminal, the Gnot, and encouraged us to program it; Tom Duff, who wrote the command interpreter `rc`; Tom Killian, who built and programmed the Gnot's SCSI interface; Ted Kowalski, who cheerfully endured early versions of the software; and Dennis Ritchie, who frequently provided us with much-needed wisdom.

References

Fra80. A. G. Fraser, ''Datakit–A Modular Network for Synchronous and Asynchronous Traffic,'' in *Proc. Int. Conf. on Commun.*, Boston, MA (June 1980).

Kal. C. R. Kalmanek, ''INCON: Network Maintenance and Privacy,'' Internal Memorandum 220106-0450, AT&T Bell Laboratories.

Met80. R. Metcalfe, D. Boggs, C. Crane, E. Taft, J. Shoch, and J. Hupp, ''The Ethernet Local Network: Three Reports,'' CSL-80-2, XEROX Palo Alto Research Centers (February, 1980).

Pik91. Pike, R., ''8.5, The Plan 9 Window System,'' *1991 USENIX Summer Conference Proceedings* (1991).

Pik87. Rob Pike, ‘‘The Text Editor sam,’’ *Software - Practice and Experience* **17**(11), pp. 813-845 (November 1987).

Pik90. R. Pike, D. Presotto, K. Thompson, and H. Trickey, ‘‘Plan 9 from Bell Labs,’’ in *UKUUG Proceedings of the Summer 1990 Conference*, London, England (July, 1990).

Pre88. D. Presotto, ‘‘Plan 9 from Bell Labs - The Network,’’ in *EUUG Proceedings of the Spring 1988 Conference*, London, England (April, 1988).

Qui. S. Quinlan, ‘‘A Cached WORM File System,’’ *Software – Practice and Experience*, p. To appear.

Res. R. C. Restrick, ‘‘INCON Wire Interface Integrated Circuit Design,’’ Internal Memorandum 52413-860314-01TM, AT&T Bell Laboratories.

8½, the Plan 9 Window System

Rob Pike

AT&T Bell Laboratories
Murray Hill, New Jersey 07974
rob@research.att.com

ABSTRACT

The Plan 9 window system, 8½, is a modest-sized program of novel design. It provides ASCII I/O and bitmap graphic services to both local and remote client programs by offering a multiplexed file service to those clients. It serves traditional UNIX files like `/dev/tty` as well as more unusual ones that provide access to the mouse and the raw screen. Bitmap graphics operations are provided by serving a file called `/dev/bitblt` that interprets client messages to perform raster operations. The file service that 8½ offers its clients is identical to that it uses for its own implementation, so it is fundamentally no more than a multiplexer. This architecture has some rewarding symmetries and can be implemented compactly; indeed 8½ is considerably *smaller* than most of its clients.

Introduction

In 1989 I constructed a toy window system from only a few hundred lines of source code using a custom language and an unusual architecture involving concurrent processes [Pike89]. Although that system was rudimentary at best, it demonstrated that window systems are not inherently complicated. Last year, for the new Plan 9 distributed system [Pike90], I applied some of the lessons from that toy project to write, in C, a production-quality window system called 8½. 8½ provides, on black-and-white or grey-scale but not color displays, the services required of a modern window system, including programmability and support for remote graphics. The entire system, including the default program that runs in the window — the equivalent of `xterm` [Far89] with 'cutting and pasting' between windows — is well under 60 kilobytes of text on a Motorola 68020 processor, about half the size of the operating system kernel that supports it and a tenth the size of the X server [Sche86] *without* `xterm`.

What makes 8½ so compact? Much of the saving comes from overall simplicity: 8½ has little graphical fanciness, a concise programming interface, and a simple, fixed user interface. 8½ also makes some decisions by fiat — three-button mouse, overlapping windows, built-in terminal program and window manager, etc. — rather than trying to appeal to all tastes. Although compact, 8½ is not ascetic. It provides the fundamentals and enough extras to make them comfortable to use. The most important contributor to its small size, though, is its overall design as a file server. This structure may be applicable to window systems on traditional UNIX-like operating systems.

The small size of 8½ does not reflect reduced functionality: 8½ provides service roughly equivalent to X windows, other than the lack of support for color terminals. (The provision of that support would not dramatically increase the code size.) 8½'s clients may of course be as complex as they choose, although the tendency to mimic 8½'s design and the clean programming interface means they are not nearly as bloated as X applications.

User's Model

8½ turns the single screen, mouse, and keyboard of the terminal (in Plan 9 terminology) or workstation (in commercial terminology) into an array of independent virtual terminals that may be ASCII terminals supporting a shell and the usual suite of tools or graphical applications using the full power of the bitmap screen and mouse. The entire programming interface is provided through reading and writing files in `/dev`.

Primarily for reasons of history and familiarity, the general model and appearance of 8½ are similar to those of `mux` [Pike88]. The right button has a short menu for controlling window creation, destruction, and placement. When a window is created, it runs the default shell, `rc` [Duff90], with standard input and output directed to the window and accessible through the file `/dev/cons` ('console'), analogous to the `/dev/tty` of UNIX.® The name change represents a break with the past: Plan 9 does not provide a Teletype-style model of terminals. 8½ provides the only way most users ever access Plan 9.

Graphical applications, like ordinary programs, may be run by typing their names to the shell running in a window. This runs the application in the same window; to run the application in a new window one may use an external program, `window`, described below. For graphical applications, the virtual terminal model is extended somewhat to allow programs to perform graphical operations, access the mouse, and perform related functions by reading and writing files with suggestive names such as `/dev/mouse` and `/dev/window` multiplexed per-window much like `/dev/cons`. The implementation and semantics of these files, described below, is central to the structure of 8½.

The default program that runs in a window is familiar to users of Blit terminals [Pike83]. It is very similar to that of `mux` [Pike88], providing mouse-based editing of input and output text, the ability to scroll back to see earlier output, and so on. It also has a new feature, toggled by typing ESC, that enables the user to control when typed characters may be read by the shell or application, instead of (for example) after each newline. This feature makes the window program directly useful for many text-editing tasks such as composing mail messages before sending them.

Plan 9 and 8½

Plan 9 is a distributed system that provides support for UNIX-like applications in an environment built from distinct CPU servers, file servers, and terminals connected by a variety of networks [Pike90]. The terminals are comparable to modest workstations that, once connected to a file server over a medium-bandwidth network such as Ethernet, are self-sufficient computers running a full operating system. Unlike workstations, however, their role is just to provide an affordable multiplexed user interface to the rest of the system: they run the window system and support simple interactive tasks such as text editing. Thus they lie somewhere between workstations and X terminals in design, cost, performance, and function. (The terminals can be used for general computing, but in practice Plan 9 users do their computing on the CPU servers.) The Plan 9 terminal software, including 8½, was developed on a 68020-based machine called a Gnot and has been ported to the NeXTstation, the MIPS Magnum 3000, and the Sun SPARCstation SLC: all small workstations that we use as terminals.

Heavy computations such as compilation, text processing, or scientific calculation are done on the CPU servers, which are connected to the file servers by high-bandwidth networks, typically point-to-point DMA links. For interactive work, these computations can access the terminal that instantiated them. The terminal and CPU server being used by a particular user are connected to the same file server, although over different networks; Plan 9 provides a view of the file server that is independent of location in the network.

The components of Plan 9 are connected by a common protocol based on the sharing of files. All resources in the network are implemented as file servers; programs that wish to access them connect to them over the network and communicate using ordinary file operations. An unusual aspect of Plan 9 is that the *name space* of a process, the set of files that can be accessed by name (for example by an `open` system call) is not global to all processes on a machine; distinct processes may have distinct name spaces. The system provides methods by which processes may change their name spaces, such as the ability to *mount* a service upon an existing directory, making the files of the service visible in the directory. (This is a different operation from its UNIX namesake.) Multiple services may be mounted upon the same

directory, allowing the files from multiple services to be accessed in the same directory. Options to the `mount` system call control the order of searching for files in such a *union directory*.

The most obvious example of a network resource is a file server, where permanent files reside. There are a number of unusual services, however, whose design in a different environment would likely not be file-based. Many are described elsewhere [Pike90]; some examples are the representation of processes for debugging, much like Killian's process files for the 8th edition [Kill84], and the implementation of the name/value pairs of the UNIX `exec` environment as files. User processes may also implement a file service and make it available to clients in the network, much like the 'mounted streams' in the 9th Edition [Pres90]. A typical example is a program that interprets an externally-defined file system such as that on a CD-ROM or a standard UNIX system and makes the contents available to Plan 9 programs. This design is used by all distributed applications in Plan 9, including 8½.

8½ serves a set of files in the conventional directory `/dev` with names like `cons`, `mouse`, and `screen`. Clients of 8½ communicate with the window system by reading and writing these files. For example, a client program, such as a shell, can print text by writing its standard output, which is automatically connected to `/dev/cons`, or it may open and write that file explicitly. Unlike files served by a traditional file server, however, the instance of `/dev/cons` served in each window by 8½ is a distinct file; the per-process name spaces of Plan 9 allow 8½ to provide a unique `/dev/cons` to each client. This mechanism is best illustrated by the creation of a new 8½ client.

When 8½ starts, it creates a full-duplex pipe to be the communication medium for the messages that implement the file service it will provide. One end will be shared by all the clients; the other end is held by 8½ to accept requests for I/O. When a user makes a new window using the mouse, 8½ allocates the window data structures and forks a child process. The child's name space, initially shared with the parent, is then duplicated so that changes the child makes to its name space will not affect the parent. The child then attaches its end of the communication pipe, `cfd`, to the directory `/dev` by doing a `mount` system call:

```
mount(cfd, "/dev", MBEFORE, buf)
```

This call attaches the service associated with the file descriptor `cfd` — the client end of the pipe — to the beginning of `/dev` so that the files in the new service take priority over existing files in the directory. This makes the new files `cons`, `mouse`, and so on, available in `/dev` in a way that hides any files with the same names already in place. The argument `buf` is a character string (null in this case), described below.

The client process then closes file descriptors 0, 1, and 2 and opens `/dev/cons` repeatedly to connect the standard input, output, and error files to the window's `/dev/cons`. It then does an `exec` system call to begin executing the shell in the window. This entire sequence, complete with error handling, is 28 lines of C.

The view of these events from 8½'s end of the pipe is a sequence of file protocol messages from the new client generated by the intervening operating system in response to the `mount` and `open` system calls executed by the client. The message generated by the `mount` informs 8½ that a new client has attached to the file service it provides; 8½'s response is a unique identifier kept by the operating system and passed in all messages generated by I/O on the files derived from that `mount`. This identifier is used by 8½ to distinguish the various clients so each sees a unique `/dev/cons`; most servers do not need to make this distinction.

A process unrelated to 8½ may create windows by a variant of this mechanism. When 8½ begins, it uses a Plan 9 service to 'post' the client end of the communication pipe in a public place. A process may open that pipe and `mount` it to attach to the window system, much in the way an X client may connect to a UNIX domain socket to the server bound to the file system. The final argument to `mount` is passed through uninterpreted by the operating system. It provides a way for the client and server to exchange information at the time of the `mount`. 8½ interprets it as the dimensions of the window to be created for the new client. (In the case above, the window has been created by the time the mount occurs, and `buf` carries no information.) When the `mount` returns, the process can open the files of the new window and begin I/O to use it.

Because 8½'s interface is based on files, standard system utilities can be used to control its services. For example, its method of creating windows externally is packaged in a 12-line shell script, called `window`, the core of which is just a `mount` operation that prefixes 8½'s directory to `/dev` and runs a command passed on the argument line:

```
mount -b $'8.5serv' /dev
$* < /dev/cons > /dev/cons >[2] /dev/cons &
```

The `window` program is typically employed by users to create their initial working environment when they boot the system, although it has more general possibilities.

Other basic features of the system fall out naturally from the file-based model. When the user deletes a window, 8½ sends the equivalent of a UNIX signal to the process group — the clients — in the window, removes the window from the screen, and poisons the incoming connections to the files that drive it. If a client ignores the signal and continues to write to the window, it will get I/O errors. If, on the other hand, all the processes in a window exit spontaneously, they will automatically close all connections to the window. 8½ counts references to the window's files; when none are left, it shuts down the window and removes it from the screen. As a different example, when the user hits the DEL key to generate an interrupt, 8½ writes a message to a special file, provided by Plan 9's process control interface, that interrupts all the processes in the window. In all these examples, the implementation works seamlessly across a network.

There are two valuable side effects of implementing a window system by multiplexing `/dev/cons` and other such files. First, the problem of giving a meaningful interpretation to the file `/dev/cons` (`/dev/tty`) in each window is solved automatically. To provide `/dev/cons` is the fundamental job of the window system, rather than just an awkward burden; other systems must often make special and otherwise irrelevant arrangements for `/dev/tty` to behave as expected in a window. Second, any program that can access the server, including a process on a remote machine, can access the files using standard read and write system calls to communicate with the window system, and standard open and close calls to connect to it. Again, no special arrangements need to be made for remote processes to use all the graphics facilities of 8½.

Graphical input

Of course 8½ offers more than ASCII I/O to its clients. The state of the mouse may be discovered by reading the file `/dev/mouse`, which returns a ten-byte message encoding the state of the buttons and the position of the cursor. If the mouse has not moved since the last read of `/dev/mouse`, or if the window associated with the instance of `/dev/mouse` is not the 'input focus', the read blocks.

The format of the message is:

```
'm'
1 byte of button state
4 bytes of x, low byte first
4 bytes of y, low byte first
```

As in all shared data structures in Plan 9, the order of every byte in the message is defined so all clients can execute the same code to unpack the message into a local data structure.

For keyboard input, clients can read `/dev/cons` or, if they need character-at-a-time input, `/dev/rcons` ('raw console'). There is no explicit event mechanism to help clients that need to read from multiple sources. Instead, a small (336 line) external support library can be used. It attaches a process to the various blocking input sources — mouse, keyboard, and perhaps a third user-provided file descriptor — and funnels their input into a single pipe from which may be read (three types of) events in the traditional style. This package is a compromise. As discussed in a previous paper [Pike89] I prefer to free applications from event-based programming. Unfortunately, though, I see no easy way to achieve this in single-threaded C programs, and am unwilling to require all programmers to master concurrent programming. It should be noted, though, that even this compromise results in a small and easily understood interface. An example program that uses it is given near the end of the paper.

Graphical output

The file `/dev/screen` may be read by any client to recover the contents of the entire screen, such as for printing (see Figure 1). Similarly, `/dev/window` holds the contents of the current window. These are read-only files.

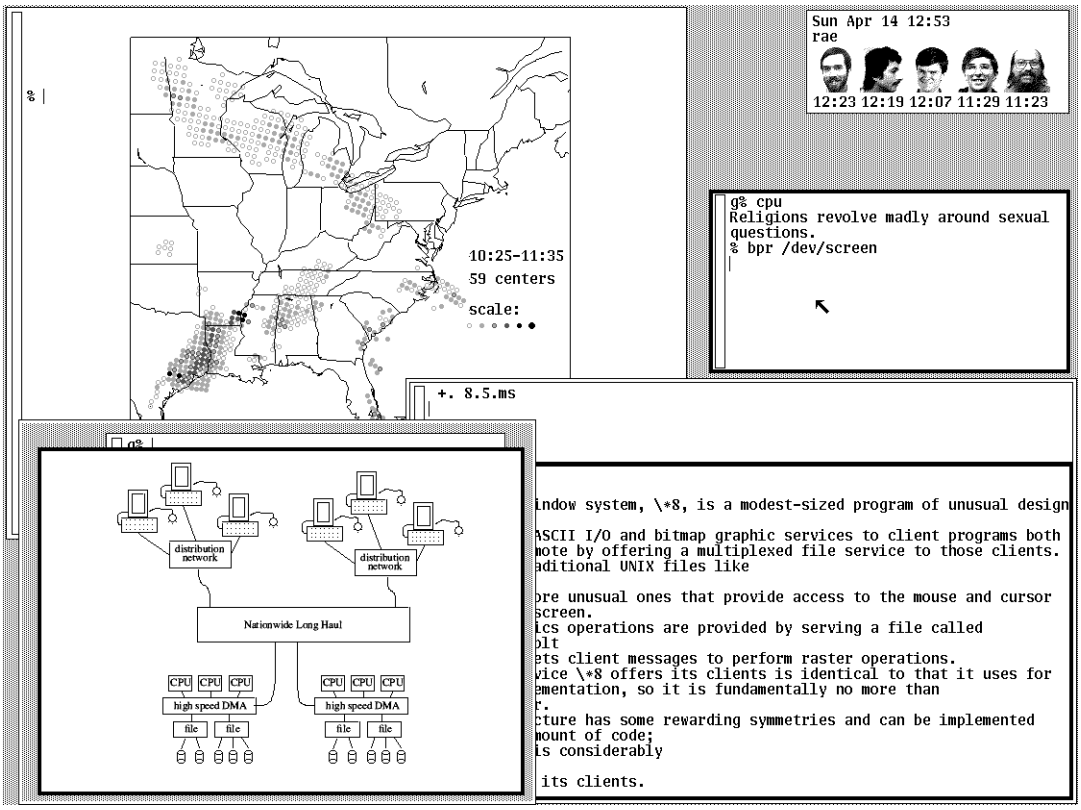

Figure 1. A representative 8½ screen, running on a NeXTstation under Plan 9 (with no NeXT software). In the upper right, a program announces the arrival of mail. In the upper left is a map displayed by the CPU server that shows the radar reflectance of precipitation in the eastern United States. In the lower right there is a screen editor, `sam` [Pike87], and in the lower left an 8½ running recursively and, inside that instantiation, a previewer for `troff` output. Underneath the faces is a small window running the command that prints the screen by passing `/dev/screen` to the bitmap printing utility.

To perform graphics operations in their windows, client programs access `/dev/bitblt`. It implements a protocol that encodes bitmap graphics operations. Most of the messages in the protocol (there are 14 messages in all) are transmissions (via a write) from the client to the window system to perform a graphical operation such as a `bitblt` [PLR85] or character-drawing operation; a few include return information (recovered via a read) to the client. As with `/dev/mouse`, the `/dev/bitblt` protocol is in a defined byte order. Here, for example, is the layout of the `bitblt` message:

`'b'`
2 bytes of destination id
2x4 bytes of destination point
2 bytes of source id
4x4 bytes of source rectangle
2 bytes of boolean function code

The message is trivially constructed from the `bitblt` subroutine in the library, defined as

```
void bitblt(Bitmap *dst, Point dp, Bitmap *src, Rectangle sr, Fcode c).
```

The 'id' fields in the message belie another property of 8½: the clients do not store the actual data for any of their bitmaps locally. Instead, the protocol provides a message to allocate a bitmap, to be stored in the server, and returns to the client an integer identifier, much like a UNIX file descriptor, to be used in operations on that bitmap. Bitmap number 0 is conventionally the client's window, analogous to standard input for file I/O. In fact, no bitmap graphics operations are executed in the client at all; they are all performed on its behalf by the server. Again, using the standard remote file operations in Plan 9, this permits remote machines having no graphics capability, such as the CPU server, to run graphics applications. Analogous features of the original Andrew window system [Gos86] and of X [Sche86] require more complex mechanisms.

Nor does 8½ itself operate directly on bitmaps. Instead, it calls another server to do its graphics operations for it, using an identical protocol. The operating system for the Plan 9 terminals contains an internal server that implements that protocol, exactly as does 8½, but for a single client. That server stores the actual bytes for the bitmaps and implements the fundamental bitmap graphics operations. Thus the environment in which 8½ runs has exactly the structure it provides for its clients; 8½ reproduces the environment for its clients, multiplexing the interface to keep the clients separate.

This idea of multiplexing by simulation is applicable to more than window systems, of course, and has some side effects. Since 8½ simulates its own environment for its clients, it may run in one of its own windows (see Figure 1). A useful and common application of this technique is to connect a window to a remote machine, such as a CPU server, and run the window system there so that each subwindow is automatically on the remote machine. It is also a handy way to debug a new version of the window system or to create an environment with, for example, a different default font.

Implementation

To provide graphics to its clients, 8½ mostly just multiplexes and passes through to its own server the clients' requests, occasionally rearranging the messages to maintain the fiction that the clients have unique screens (windows). To manage the overlapping windows it uses the layers model, which is handled by a separate library [Pike83a]. Thus it has little work to do and is a fairly simple program; it is dominated by a couple of modest-sized switch statements to interpret the bitmap and file server protocols. The built-in window program and its associated menus and text-management support are responsible for most of the code.

The operating system's server is also compact; the version for the 68020 processor, excluding the implementation of a half dozen bitmap graphics operations, is 1153 lines of C; the graphics operations are another 1118 lines.

8½ is structured as a set of communicating coroutines, much as discussed in a 1989 paper [Pike89]. One coroutine manages the mouse, another the keyboard, and another is instantiated to manage the state of each window and associated client. When no coroutine wishes to run, 8½ reads the next file I/O request from its clients, which arrive serially on the full-duplex communication pipe. Thus 8½ is entirely synchronous.

The program source is small and compiles in about 10 seconds in our Plan 9 environment. There are ten source files and one `makefile` totaling 3860 lines. This includes the source for the window management process, the cut-and-paste terminal program, the window/file server itself, and a small coroutine library (`proc.c`). It does not include the layer library (another 610 lines) or the library to handle the cutting and pasting of text displayed in a window (898 lines), or the general graphics support library that manages all the non-drawing aspects of graphics — arithmetic on points and rectangles, memory management, error handling, clipping, — plus events and non-primitive drawing operations such as circles and ellipses (a final 1797 lines). Not all the pieces of these libraries are used by 8½ itself; a large part of the graphics library in particular is used only by clients. Thus it is somewhat unfair to 8½ just to sum these numbers, including the 2271 lines of support in the kernel, and arrive at a total implementation size of 9436 lines of source to implement all of 8½ from the lowest levels to the highest. But that number gives a fair measure of the complexity of the overall system.

The implementation is also efficient. 8½'s performance is competitive to X windows'. Compared using Dunwoody's and Linton's `gbench` benchmarks on the 68020, distributed with the ''X Test Suite'', circles and arcs are drawn about half as fast in 8½ as in X11 release 4 compiled with `gcc` for equivalent hardware, probably because they are currently implemented in a user library by calls to the `point` primitive. Line drawing speed is about equal between the two systems. Text is drawn about 20% faster in 8½ and the `bitblt` test is about four times faster. These numbers vary enough to caution against drawing sweeping conclusions, but they suggest that 8½'s architecture does not penalize its performance. Finally, 8½ boots in under a second and creates a new window apparently instantaneously.

An example

Here is a complete program that runs under 8½. It prints the string `"hello world"` wherever the left mouse button is depressed, and exits when the right mouse button is depressed. It also prints the string in the center of its window, and maintains that string when the window is resized.

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

void
ereshaped(Rectangle r)
{
        Point p;

        screen.r = r;
        bitblt(&screen, screen.r.min, &screen, r, Zero); /* clear window */
        p.x = screen.r.min.x + Dx(screen.r)/2;
        p.y = screen.r.min.y + Dy(screen.r)/2;
        p = sub(p, div(strsize(font, "hello world"), 2));
        string(&screen, p, font, "hello world", S);
}

main(void)
{
        Mouse m;

        binit(0, 0);    /* initialize graphics library */
        einit(Emouse);  /* initialize event library */
        ereshaped(screen.r);
        for(;;){
                m = emouse();
                if(m.buttons & RIGHTB)
                        break;
                if(m.buttons & LEFTB){
                        string(&screen, m.xy, font, "hello world", S);
                        /* wait for release of button */
                        do; while(emouse().buttons & LEFTB);
                }
        }
}
```

The complete loaded binary is a little over 18K bytes on a 68020. This program should be compared to the similar ones in the excellent paper by Rosenthal [Rose88]. (The current program does more: it also employs the mouse.) The clumsiest part is `ereshaped`, a function with a known name that is called from the event library whenever the window is reshaped or moved, as is discovered inelegantly but adequately by a special case of a mouse message. (Simple so-called expose events are not events at all in 8½; the layer library takes care of them transparently.) The lesson of this program, in deference to Rosenthal, is that if the window system is cleanly designed a toolkit should be unnecessary for simple tasks.

Status

8½ is in regular daily use by almost all the 60 people in our research center. Some of those people use it to access Plan 9 itself; others use it as a front end to remote UNIX systems, much as one would use an X terminal. It has been fairly stable for almost a year.

Some things about 8½ may change. It would be nice if its capabilities were more easily accessible from the shell. A companion to this paper [Pike91] proposes one way to do this, but that does not include any graphics functionality. Perhaps an ASCII version of the `/dev/bitblt` file is a way to proceed; that would allow, for example, `awk` programs to draw graphs directly. Finally, 8½ offers no 'iconization' of its clients, and probably never will. But it would be helpful for it to provide some quick mechanism for managing a cluttered screen, and some ideas are being considered. By the time this paper appears, 8½ will probably address this issue. The code is unlikely to grow substantially as a result, however, and clients will be unaffected by the changes.

Can this style of window system be built on other operating systems? A major part of the design of 8½ depends on its structure as a file server. In principle this could be done for any system that supports user processes that serve files, such as any system running NFS or AFS [Sun89, Kaza87]. One requirement, however, is 8½'s need to respond to its clients' requests out of order: if one client reads `/dev/cons` in a window with no characters to be read, other clients should be able to perform I/O in their windows, or even the same window. Another constraint is that the 8½ files are like devices, and must not be cached by the client. NFS cannot honor these requirements; AFS may be able to. Of course, other interprocess communication mechanisms such as sockets could be used as a basis for a window system. One may even argue that X's model fits into this overall scheme. It may prove easy and worthwhile to write a small 8½-like system for commercial UNIX systems to demonstrate that its merits can be won in systems other than Plan 9.

Conclusion

In conclusion, 8½ uses an unusual architecture in concert with the file-oriented interprocess communication of Plan 9 to provide network-based interactive graphics to client programs. It demonstrates that even production-quality window systems are not inherently large or complicated and may be simple to use and to program.

Acknowledgements

Helpful comments on early drafts of this paper were made by Doug Blewett, Stu Feldman, Chris Fraser, Brian Kernighan, Dennis Ritchie, and Phil Winterbottom. Many of the ideas leading to 8½ were tried out in earlier, sometimes less successful, programs. I would like to thank those users who suffered through some of my previous 7½ window systems.

References

[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

[Far89] Far too many people, XTERM(1), Massachusetts Institute of Technology, 1989

[Gos86] James Gosling and David Rosenthal, ''A window manager for bitmapped displays and UNIX'', in Methodology of Window Management, edited by F.R.A. Hopgood et al., Springer, 1986

[Kaza87] Mike Kazar, ''Synchronization and Caching issues in the Andrew File System'', Tech. Rept. CMU-ITC-058, Information Technology Center, Carnegie Mellon University, June, 1987

[Kill84] Tom Killian, ''Processes as Files'', USENIX Summer Conf. Proc., Salt Lake City June, 1984

[Pike83] Rob Pike, ''The Blit: A Multiplexed Graphics Terminal'', Bell Labs Tech. J., V63, #8, part 2, pp. 1607-1631

[Pike83a] Rob Pike, ''Graphics in Overlapping Bitmap Layers'', Trans. on Graph., Vol 2, #2, 135-160, reprinted in Proc. SIGGRAPH '83, pp. 331-356

[Pike87] Rob Pike, ''The Text Editor `sam`'', Softw. - Prac. and Exp., Nov 1987, Vol 17 #11, pp. 813-845

[Pike88] Rob Pike, ''Window Systems Should Be Transparent'', Comp. Sys., Summer 1988, Vol 1 #3, pp. 279-296

[Pike89] Rob Pike, ‘‘A Concurrent Window System’’, Comp. Sys., Spring 1989, Vol 2 #2, pp. 133-153
[Pike90] Rob Pike, Dave Presotto, Ken Thompson, and Howard Trickey, ‘‘Plan 9 from Bell Labs’’, Proc. of the Summer 1990 UKUUG Conf., London, July, 1990, pp. 1-9
[Pike91] Rob Pike, ‘‘A Minimalist Global User Interface’’, USENIX Summer Conf. Proc., Nashville, June, 1991, this volume
[PLR85] Rob Pike, Bart Locanthi and John Reiser, ‘‘Hardware/Software Tradeoffs for Bitmap Graphics on the Blit’’, Softw. - Prac. and Exp., Feb 1985, Vol 15 #2, pp. 131-152
[Pres90] David L. Presotto and Dennis M. Ritchie, ‘‘Interprocess Communication in the Ninth Edition Unix System’’, Softw. - Prac. and Exp., June 1990, Vol 20 #S1, pp. S1/3-S1/17
[Rose88] David Rosenthal, ‘‘A Simple X11 Client Program -or- How hard can it really be to write ‘‘Hello, World’’?’’, USENIX Winter Conf. Proc., Dallas, Jan, 1988, pp. 229-242
[Sche86] Robert W. Scheifler and Jim Gettys, ‘‘The X Window System’’, ACM Trans. on Graph., Vol 5 #2, pp. 79-109
[Sun89] Sun Microsystems, NFS: Network file system protocol specification, RFC 1094, Network Information Center, SRI International, March, 1989.

Multiprocessor Streams for Plan 9

David Leo Presotto

AT&T Bell Laboratories
Murray Hill, New Jersey 07974
research!presotto
presotto@research.att.com

ABSTRACT

This paper describes an implementation of Streams for the Plan 9 kernel, a multi-threaded, multiprocessor kernel with a system call interface reminiscent of UNIX. Rather than port Dennis Ritchie's Streams to Plan 9, we changed the abstraction to fit more naturally into the new environment. The result is a mechanism that has similar performance and is internally easier to program.

1. Introduction

Plan 9 is a new computing environment being built and used by the Computing Science Research Center at AT&T Bell Laboratories. Plan 9 consists of terminals, CPU servers and file servers connected by various networks. These components run specialized operating systems based on a common multi-threaded kernel. The kernel runs on both uniprocessors and shared memory multiprocessors.

Plan 9 communicates via a number of different networks. Therefore we decided to base all our network code on a single structure. This allowed us to solve at once a number of problems such as flow control, memory allocation, and user interface. Given our past experience with it, we chose Dennis Ritchie's Stream I/O System [Rit84] to provide the structure for Plan 9. This coroutine-based design, introduced in the Eighth Edition, provides a clean, flexible mechanism for handling asynchronous I/O. Although Plan 9's kernel is unrelated to that of the Eighth Edition [McI85], the concept of Streams remained directly applicable. We have, however, made two major alterations.

Plan 9 runs on multiprocessor systems so we wanted to exploit their concurrency. In the Plan 9 kernel, the basic unit of concurrency is the process. We therefore converted Ritchie's coroutine-based design to a process-based one. As we shall see later, this change has both advantages and disadvantages.

Associated with the change to a process-based structure, we also had to reduce the number of threads. If we had made the most obvious change to convert each of Ritchie's coroutines into a process, we would have incurred very high CPU penalties. No matter how cheap we make our kernel processes they would never be as cheap as coroutines. Instead, we chose a structure that performs in one process what Streams does in many coroutines.

The result is a structure very similar to Streams but, we believe, easier to program. The interfaces, flow control, and memory allocation are the same. However, the freedom to allow processing modules to block and to use any resources available to a user process has made many pieces much easier to program. A process is a familiar programming construct.

In the rest of this paper we will refer to Ritchie's Streams simply as Streams and to Plan 9 Streams as Plan 9.

2. Data Structures

Plan 9, aside from minor changes, uses the same data structures as Streams. Our description here is very brief and is intended to highlight the differences. We refer the reader to Dennis's excellent BLTJ paper [Rit84] for a more comprehensive treatment.

The appendix contains the C definitions of our data structures.

2.1. Stream

A `Stream` is a full duplex channel connecting a device or pseudo-device to a user process. User processes insert and remove data at one end of the stream. Kernel processes acting on behalf of a device insert data at the other.

A stream is made up of a linear list of *processing modules*. Each module has both an upstream (toward the user) and a downstream (toward the device) *put procedure*. Data is inserted into a stream by calling the put procedure of the module at either end of the stream. Each module calls the succeeding one to send data up or down the stream.

2.2. Queue

An instance of a processing module is described by a pair of `Queues`, one for each direction. Each queue contains:

- a pointer to the put procedure
- a pointer to the next queue
- a linked list of queued data blocks
- the number of bytes queued
- the number of blocks queued
- a spin lock to control access to the data structure

Unlike a Stream queue ours has no *service procedure*. Also, since on a multiprocessor setting priority levels is not a valid synchronization mechanism, we require a spin lock from the operating system to control access to the queue.

2.3. Block

The objects passed through the stream are described by a data structure called a `Block`. Our blocks are identical to Streams and contain a *base pointer*, a *limit pointer*, a *read pointer*, and a *write pointer*. Each pointer refers to memory mapped into kernel space. The base and limit are never changed and are used to describe the data allocated to the block. The read and write pointers point to the start and end of usable data within the block.

There are two block types, *data* and *control.* Data blocks are used to pass information from process to process. Control blocks are used to control the action of the modules. They both have the same format. Data blocks are often queued in a processing module until some condition is met for passing them along or freeing them. Control blocks are never queued but are passed from module to module until one accepts and frees them.

Streams also have data and control blocks. However, their control blocks come in multiple flavors, all queuable; some with the same priority as data and some higher. Higher priority blocks are moved to the front of the queue. At a result, the routines used to manipulate these control blocks tend to be complex.

When a module's *put* is called it is passed a pointer to a block. If one desires to pass many blocks atomically, the blocks may be chained together and a pointer to the first is passed to the procedure. This is similar to the way *mbufs* are passed in the BSD kernel [Lef89]. Streams need no such concept since no two threads run simultaneously in a Streams procedure. UNIX System V STREAMS [Bac86] have a much more complicated construct to pass a multi-block message along with a single put. The System V construct is used both for atomicity (they originally had no other block delimiters) and for performance.

3. Algorithms

3.1. Memory Allocation

Stream memory is allocated at system start time. A list is kept for each of several fixed block sizes. A process that requests a size receives the smallest block that can hold the request. Synchronizing access of the free lists is performed using a spin lock per free list.

Since all stream code runs in the context of processes, whenever an allocation cannot be immediately processed the caller blocks until a block of the right size is freed. The result is that momentary surges in used blocks do not panic the kernel as they sometimes did in the Eight Edition.

The number of lists, the specific block sizes and the number allocated of each size depends on the kernel. Terminals tend to use more small blocks, the servers more large ones. The allocated blocks reflect this.

3.2. User Interface

A stream is represented at user level as a directory containing at least two files, `ctl` and `data`. The first process to open either file creates the stream automatically. The last process to close destroys the stream. Writing to the `data` file causes a switch to kernel mode. The process then copies the data into kernel data blocks and calls the put procedure of the first downstream processing module for each block. The last block of a write is flagged with a delimiter in the event that the downstream module cares about write boundaries. In most cases the first put procedure calls the second, the second calls the third, and so on until the data is output. Thus, data may often be sent without taking a context switch. A write lock at the top of the stream assures that no two processes can simultaneously insert data into the top of the same stream, which insures that all writes are atomic.

Our system has no `ioctl` system call. The syntax and semantics of `ioctl` in UNIX are so uncontrolled that we left it out. Writing to the `ctl` file takes the place of `ioctl`. Writing to the control file is the same as writing to a data file except that the blocks created are of type control. A processing module parses each control block put to it. The commands in the control blocks are simple ASCII strings. Therefore, there is no problem with byte ordering when one system is controlling streams in a name space implemented on another processor. The time to parse the control blocks is not important since the control operation is a rare one, usually used only when starting operation on a stream.

The stream system intercepts the control blocks that control configuration of the stream. These control blocks are:

push *name* to add an instance of the processing module *name* to the top of the stream.

pop to remove the top module of the stream.

hangup to send a control message containing the string ‘‘hangup’’ up the stream from the device end.

Other control blocks are read by each module they pass through.

Reading from the `data` file returns data queued at the top of the stream. The read terminates either when the read count is reached or when the end of a delimited block is read. There is a per stream read lock that ensures that only one process can read from the stream at a time. This ensures that the bytes read were contiguous bytes from the stream. Reading the `ctl` file is described in the section on multiplexing.

3.3. Device Input

When input exists at a device, the driver’s interrupt routine wakes up a kernel process to carry the data upstream. A kernel process is an ordinary process with no user level segments allocated to it and is scheduled just like any other process. Message-based devices like Ethernet [Met80] may have many processes ready to carry the next message upstream so that many messages can be processed simultaneously.

The kernel process carries the message upstream through protocol modules. Eventually, the message is delivered to the most upstream queue. The kernel process leaves it there and wakes any user process blocked in a read on that stream. Thus, the only difference between input and output is that the user process must perform the copy of the data from kernel blocks to user space. This can be a benefit in our

multiprocessors in which each processor has a separate cache. If the kernel process were to copy the data into the user process it would most likely do it on the wrong processor and hence into the wrong cache. This would force the user process to fault it into its own cache, increasing the load on the shared memory bus.

3.4. Multiplexing

Most protocols need to multiplex several conversations onto a single physical device. This is added to our scheme using pseudo-devices, one for each multiplexed conversation. This is very similar to the way Eighth Edition handles TCP/IP. A group of pseudo-devices are coupled with a multiplexing processing module that is pushed onto a physical device stream. The device end modules on the pseudo-devices add the necessary header onto downstream messages and then put them to the module downstream of the multiplexor. The multiplexing module looks at each message moving up its stream and puts it to the correct pseudo-device stream after stripping whatever header it used to do the demultiplexing.

The user interface to a multiplexed protocol is a directory. The directory contains a `clone` file and a stream directory for each conversation. The stream directories are numbered *1* to *n* (see Figure 1). Opening the clone file is a macro for finding a free stream directory and opening its `ctl` file. Reading the control file returns the ASCII number of the conversation chosen. This allows the user process to find the corresponding `data` file.

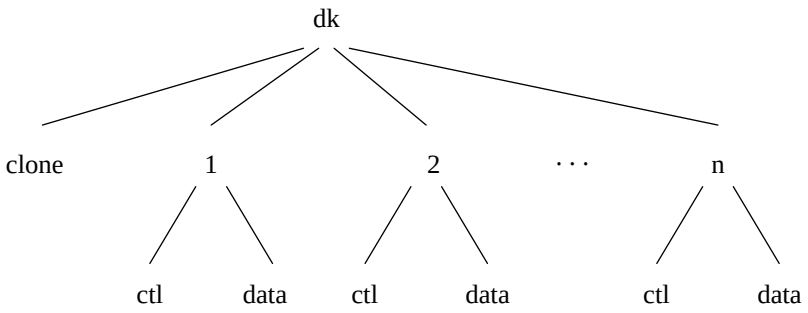

Figure 1

3.5. Pipes

Pipes, as in Eighth Edition, are just two streams joined at the device end. The `pipe` system call returns file descriptors for the data files of the two streams. The control files are inaccessible.

3.6. Helper Processes

Transport protocols need to retransmit lost data. However, to achieve true pipelining, the user process will want to queue data at the protocol module and return. Another process has to retransmit the data when needed since all put procedures must be called in the context of a process. For this purpose, processing modules can create kernel processes to perform such actions when needed. The processes are awakened on need by the processing module's put procedure or whenever a timer expires.

3.7. Flow Control

In any system that queues data one needs a mechanism to keep a queue with a slow reader from absorbing all of memory. We use a flow control mechanism similar to Streams. Each queue keeps a count of bytes and a count of blocks queued there. Whenever either exceeds a predetermined limit, the high water flag is set for that queue. Each caller of a processing module checks the high water flag for the next queue before calling its put procedure. If the next queue is past its high water mark, the would-be caller goes to sleep, leaving a pointer to itself in its queue (the rendezvous structure in `Queue`). When a process empties a queue past half its high water mark it wakes up any process waiting at the previous queue.

Modules implementing transport protocols with window schemes implement flow control a little

differently. Rather than go to sleep, they always pass data upstream. However, when the upstream queue is full, the transport module stops sending acknowledgements back to the remote system. Hence, the remote side will eventually stop sending. To open the window again, a helper process sleeps in the queue instead of the device process. When the next queue empties, the helper is awakened and it does whatever is needed to open the remote system's transmit window.

4. Performance

A different flavor of Streams cannot be evaluated without comparing it against earlier ones. Our results give a general idea of where the advantages and disadvantages of Plan 9 Streams lie. However, the systems compared are in many ways incomparable. The compilers, operating systems, and Streams code all have a considerable effect on the results.

For Plan 9 numbers we use 4 different configurations. Three are SGI Power Series machines with 1, 2, and 4 processors. We compare these against another 4 processor SGI Power Series running System V Release 3 and against a single processor MIPS M2000 system also running SVR3. The M2000 has the same CPU running at the same speed as the SGI machines. However, it has a considerably faster memory system. Outside of tight loops, this has a major impact on processing speed.

The other Plan 9 system is the Gnot terminal [Pik90]. This is a system developed in our center and now manufactured for us by AT&T. It uses a 25 MHz Motorola 68020. We compare it against a DEC MicroVAX 3. The machines on average are about the same speed. The Gnot CPU is about 4/3 the speed of the microVAX with a memory system that is about 2/3 the speed of the microVAX.

All tests measure both throughput and latency. The throughput is tested using the following program:

```
int i;
char buf[64*1024];
int p[2];

makeconnection(p);
switch(fork()){
case 0:
        close(p[1]);
        while(read(p[0], buf, sizeof buf) > 0)
                ;
        break;
default:
        close(p[0]);
        for(i = 0; i<ITER; i++){
                if(write(p[1], buf, sizeof buf) != sizeof buf){
                        perror("write");
                        exit(1);
                }
        }
        break;
}
```

The block size is chosen to be large to minimize the difference in system call speeds. The latency is tested using:

```
int p[2];
int i;
char c;

makeconnection(p);
switch(fork()){
case 0:
        close(p[1]);
        while(read(p[0], &c, 1) == 1)
                write(p[0], &c, 1);
        break;
default:
        close(p[0]);
        for(i = 0; i<ITER; i++){
                if(write(p[1], &c, 1) != 1){
                        perror("write");
                        exit(1);
                }
                if(read(p[1], &c, 1) != 1){
                        perror("read");
                        exit(1);
                }
        }
        break;
}
```

In both cases, we perform each operation for a large number of iterations to get an average time.

The first test (Table 1) compares Plan 9 pipes against SVR3 normal pipes, SVR3 stream pipes, a BSD sockets implementation running under SVR3, and Tenth Edition Streams. Since only two processes are involved in all configurations and no processing is being performed by processing modules, we are comparing the speed of the basic plumbing in the systems.

Table 1 - Pipes

system	throughput MBytes/sec	latency ms
SGI/1 CPU Plan 9	6.0	.29
SGI/2 CPUs Plan 9	8.4	.21
SGI/4 CPUs Plan 9	8.4	.28
SGI/4 CPUs sVr3 old pipes	4.5	.51
M2000 sVr3 stream	8.0	.51
M2000 sVr3 sockets	8.0	.36
68020 Gnot Plan 9	1.79	1.67
uVAX 3 10th Edition spipe	1.04	1.69

From Table 1 we can see that for the large machines, Plan 9 has lower latency. This was the expected result since the straight call structure requires many fewer instructions than traditional Streams which must

schedule each service procedure in addition to calling its put procedure. The dip of .08 ms when going from 1 processor to 2 is the result of concurrency. The second process is starting up before the first has returned from queuing its block.

The unexpected result is the rise from .21 ms to .28 when going from 2 processors to 4. We believe that this is contention over the process queue. We hope to verify this assumption before this paper is presented.

The low single processor SGI throughput for Plan 9 compared to the M2000 reflects the slower memory on the SGI box. When we use multiple processors, we take advantage of the concurrency and our throughput passes all the others.

Plan 9 on the Gnot compared to the Tenth Edition on the MicroVAX is less impressive. We still have a definite advantage in throughput. However, given the ratio of machine speeds, we should be much better in latency. Profiling the kernel showed that the disappointing latency time was due entirely to the MMU. The MMU on the Gnot only retains one process state. Whenever we switch context we do a lot of faulting to refill the MMU. The reason the throughput doesn't suffer from this problem is that the pipelining causes a lot of data to be moved per context switch.

To compare the performance of kernel driver processors to performing puts at interrupt level, we used our most prevalent network, Datakit [Che80]. It is both a local and wide area network spanning all of AT&T. The MicroVAX, SGI, and M2000 each have 8 megabit/sec links to Datakit. However, due to constraints in the Datakit the highest throughput is 2.6 megabits. The Gnots have a slower 2 megabit link [Pre88]. Table 2 presents performance of various systems through the Datakit. In all cases the remote system is a Plan 9 SGI processor. Once again, Plan 9 throughput matches or exceeds the throughput of the other systems. However, latency is worse. This is the price paid for using kernel processes to send device data upstream rather than doing it in the interrupt routine. The degradation is especially evident in the Gnot since it is the worst at process switching.

Table 2 - URP/Datakit

system	throughput KBytes/sec	latency ms
SGI/2 CPUs Plan 9	235	1.4
SGI/4 CPUs Plan 9	235	1.4
M2000 sVr3	235	1.2
68020 Gnot Plan 9	100	5.8
uVAX 3 10th Edition	85	3.2

Finally, we present some Ethernet performance results. We don't compare these against other systems since the protocol we use, Nonet, currently runs only on Plan 9. It was designed as a low weight transport protocol. It should be noted that the throughput figures are higher than any we've seen published to date for an Ethernet. This is as much a function the protocol as it is of Plan 9.

Table 3 - Nonet/Ethernet		
system	throughput KBytes/sec	latency ms
SGI/2 CPUs Plan 9	950	1.4
SGI/4 CPUs Plan 9	950	1.4

Related Work

We must mention Larry Peterson's *x*-Kernel work [Pet89]. The *x*-Kernel is very similar to Plan 9 Streams. It is used primarily to study the decomposition of network protocols. The process structure, multiplexing, and data structures are virtually identical to Plan 9 Steams.

The greatest differences between our systems are:

1) The *x*-Kernel uses very light weight kernel processes. They are just a PC and a stack. Our kernel processes carry all the baggage of user processes.

2) Rather than queue data at the user process and wait for the user process to read it, *x*-Kernel kernel processes call a put procedure in the user's address space which moves the data directly into user memory.

3) The message in the *x*-Kernel is in the form of a tree of blocks. A pointer to the top of the tree is passed through the processing modules. We use a linear structure.

Published performance of the *x*-Kernel is similar to ours with lower latency times. We hope to borrow some of the ideas of the *x*-Kernel to improve our own performance.

Conclusions

We have presented another variation of streams. The main advantage to Plan 9 Streams is making use of concurrency in multiprocessors. We have a very subjective belief that the process based model is easier to program than the coroutine based one.

The performance results show that the Plan 9 model has high throughput. However, the contexts switches caused by the kernel processes increase latency. Further research and tuning is required to reduce these costs.

5. References

Bac86. M. Bach, *The Design of the UNIX Operating System*, Prentice-Hall (1986).

Che80. G. L. Chesson and A. G. Fraser, ''Datakit Network Architecture,'' in *IEEE Compcon '80* (January, 1980).

Lef89. S. Leffler, M. K. McKusick, M. Karels, and J. Quarterman, *4.3BSD UNIX Operating System*, Addison Wesley (1989).

McI85. M. D. McIlroy, *Unix Programmer's Manual, Eighth Edition*, Bell Laboratories, Murray Hill, NJ, USA (February, 1985).

Met80. R. Metcalfe, D. Boggs, C. Crane, E. Taft, J. Shoch, and J. Hupp, ''The Ethernet Local Network: Three Reports,'' CSL-80-2, XEROX Palo Alto Research Centers (February, 1980).

Pet89. L. Peterson, ''RPC in the X-Kernel: Evaluating New Design Techniques,'' in *Proceedings Twelfth Symposium on Operating Systems Principles*, Litchfield Park, AZ (December, 1989).

Pik90. R. Pike, D. Presotto, K. Thompson, and H. Trickey, ''Plan 9 from Bell Labs,'' in *UKUUG Proceedings of the Summer 1990 Conference*, London, England (July, 1990).

Pre88. D. Presotto, ''Plan 9 from Bell Labs - The Network,'' in *EUUG Proceedings of the Spring 1988*

Conference, London, England (April, 1988).

Rit84. D. M. Ritchie, ‘‘A Stream Input-Output System,’’ *AT&T Bell Laboratories Technical Journal* **63**(8) (October, 1984).

Appendix

```
/*
 *  operations available to a queue
 */
typedef struct Qinfo    Qinfo;
struct Qinfo
{
        void (*iput)(Queue*, Block*);   /* input routine */
        void (*oput)(Queue*, Block*);   /* output routine */
        void (*open)(Queue*, Stream*);
        void (*close)(Queue*);
        char *name;
};

/*
 *  We reference kernel memory via descriptors kept in host memory
 */
typedef struct Block    Block;
struct Block
{
        Block   *next;
        uchar   *rptr;          /* first not consumed byte */
        uchar   *wptr;          /* first empty byte */
        uchar   *lim;           /* 1 past the end of the buffer */
        uchar   *base;          /* start of the buffer */
        uchar   flags;
        uchar   type;
};

/* flag bits */
#define S_DELIM 0x80    /* this block is the end of a higher level message */
#define S_CLASS 0x07

/* type values */
#define M_DATA 0
#define M_CTL 1

/*
 *  a list of blocks
 */
typedef struct Blist    Blist;
struct Blist {
        Lock;
        Block   *first;         /* first data block */
        Block   *last;          /* last data block */
        long    len;            /* length of list in bytes */
};
```

```
/*
 *  a queue of blocks
 */
typedef struct Queue    Queue;
struct Queue {
        Blist;
        int     nb;             /* number of blocks in queue */
        int     flag;
        Qinfo   *info;          /* line discipline definition */
        Queue   *other;         /* opposite direction, same line discipline */
        Queue   *next;          /* next queue in the stream */
        void    (*put)(Queue*, Block*);
        Rendez  r;              /* flow control rendezvous point */
        void    *ptr;           /* private info for the queue */
};
#define QHUNGUP 0x1     /* flag bit meaning the stream has been hung up */
#define QINUSE  0x2
#define QHIWAT  0x4     /* queue has gone past the high water mark */

/*
 *  a stream head
 */
struct Stream {
        Lock;                   /* structure lock */
        int     inuse;          /* use count */
        int     hread;          /* number of reads after hangup */
        int     type;           /* correlation with Chan */
        int     dev;            /* ... */
        int     id;             /* ... */
        QLock   rdlock;         /* read lock */
        QLock   wrlock;         /* write lock */
        Queue   *procq;         /* write queue at process end */
        Queue   *devq;          /* read queue at device end */
};
#define RD(q)           ((q)->other < (q) ? (q->other) : q)
#define WR(q)           ((q)->other > (q) ? (q->other) : q)
#define PUTNEXT(q,b)    (*(q)->next->put)((q)->next, b)
#define BLEN(b)         ((b)->wptr - (b)->rptr)
#define QFULL(q)        ((q)->flag & QHIWAT)
#define FLOWCTL(q)      { if(QFULL(q)) flowctl(q); }
```

Process Sleep and Wakeup on a Shared-memory Multiprocessor

Rob Pike
Dave Presotto
Ken Thompson
Gerard Holzmann

{rob|presotto|ken|gerard}@research.att.com
AT&T Bell Laboratories
Murray Hill, New Jersey 07974

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 months 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.

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)
{
        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; it 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, and Howard Trickey, ‘‘Plan 9 from Bell Labs,’’ pp. 1-9 in *Proceedings of the Summer 1990 UKUUG Conference*,, London (July, 1990).

Sal66. Jerome H. Saltzer, *Traffic Control in a Multiplexed Computer System*, MIT, Cambridge, Mass. (1966).

Rc — A Shell for Plan 9 and UNIX Systems

Tom Duff

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Rc is a command interpreter for Plan 9. It also runs on a variety of traditional systems, including SunOS and the Tenth Edition. It provides similar facilities to Bourne's */bin/sh*, with some small additions and mostly less idiosyncratic syntax. This paper introduces *rc*'s highlights with numerous examples, and discusses its design and why it varies from Bourne's.

1. Introduction

Plan 9 needs a command-programming language. As porting the Bourne shell to an incompatible new environment seemed a daunting task, I chose to write a new command interpreter, called *rc* because it runs commands. Although tinkering with perfection is a dangerous business, I could hardly resist trying to 'improve' on Bourne's design. Thus *rc* is similar in spirit but different in detail from Bourne's shell.

The bulk of this paper describes *rc*'s principal features with many small examples and a few larger ones. We close with a discussion of the principles guiding *rc*'s design and why it differs from Bourne's design. The descriptive sections include little discussion of the rationale for particular features, as individual details are hard to justify in isolation. The impatient reader may wish to skip to the discussion at the end before skimming the expository parts of the paper.

2. Simple commands

For the simplest uses *rc* has syntax familiar to Bourne-shell users. Thus all of the following behave as expected:

```
date
con alice
who >user.names
who >>user.names
wc <file
echo [a-f]*.c
who | wc
who; date
cc *.c &
cyntax *.c && cc -g -o cmd *.c
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. Variables

Rc provides variables whose values are lists of arguments. Variables may be given values by typing, for example:

```
path=(. /bin /usr/bin)
user=td
tty=/dev/tty8
```

The parentheses indicate that the value assigned to `path` is a list of three strings. The variables `user` and `tty` 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 /usr/bin
```

Variables may be subscripted by numbers or lists of numbers, like this:

```
echo $path(2)
echo $path(3 2 1)
```

These are equivalent to

```
echo /bin
echo /usr/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 the number of entries in `$path`.

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 `()`.

5. 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.

6. 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

```
cc -o $i $i^.c
```

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.

7. 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 `$'` follows a quoted or unquoted word, or an unquoted word follows a quoted word with no intervening blanks or tabs, a ^ is inserted between the two. If an unquoted word immediately following a `$` contains a character other than an alphanumeric, underscore or `*`, a ^ is inserted before the first such character.

8. 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 catenate the ten oldest files in the current directory in temporal order.

9. 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`.) On systems without `/dev/fd` or something similar (SunOS for example) this feature does not work.

10. 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 commands' exit statuses 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*.

11. 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!'&
```

this would lock up the terminal for an hour, then print the message in the background!

12. 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 `in` *list* is not given, *$** is used.

13. Conditional execution — `if`

Rc also provides a general if-statement. For example:

```
if(cyntax *.c) cc -g -o cmd *.c
```

runs the C compiler whenever `cyntax` finds no problems with `*.c`. 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.

14. Control flow — `while`

Rc's while statement looks like this:

```
while(newer subr.c subr.o) sleep 5
```

This waits until `subr.o` is newer than `subr.c` (presumably because the C compiler finished with it.)

15. 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 the first characters of `.` 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 tacks standard input to its end. With two, the first is appended to the second. Any other number elicits a usage 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
}
```

16. 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 {
    gre -e $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.

17. Command execution

Up to now we've said very little about what *rc* does 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, if the name contains a `/`, it is taken to be the name of a binary program and is executed using *exec*(2). If the name contains no `/`, then directories mentioned in the variable `$path` are searched until an executable file is found.

18. Built-in commands

Several commands are executed internally by *rc* because they are difficult or impossible 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 catenated 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 for `$x`.

`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 /usr/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.

19. 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,

```
cc junk.c >[2]junk.diag
```

saves the compiler's diagnostics 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

```
cc 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

```
cc 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,

```
cc 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 at 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,

```
cc junk.c >[2=]
```

will discard diagnostics from the compilation.

Arbitrary file descriptors may be sent through a pipe by typing, for example

```
cc junk.c |[2] grep -v '^$'
```

This deletes those ever-so-annoying blank lines from the C compiler's output. Note that the output of `grep` still appears on file descriptor 1.

Very 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.

20. 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 <<!
...
nls 2T-402 2912
norman 2C-514 2842
pjw 2T-502 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 it connected to the command's standard input when it is run.

Rc does variable substitution in here documents. The following *subst* 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.

Here documents may be provided on file descriptors other than 0 by typing, for example

```
cmd <<[4]End
...
End
```

21. Signals

Rc scripts normally terminate when an interrupt is received from the terminal. A function with the name of a signal, in lower case, is defined in the usual way, but called when *rc* receives the signal. Signals of interest are:

`sighup`
: Hangup. The controlling teletype has disconnected from *rc*.

`sigint`
: The interrupt character (usually ASCII del) was typed on the controlling terminal.

`sigquit`
The quit character (usually ASCII fs, ctrl-\) was typed on the controlling terminal.

`sigterm`
This signal is normally sent by *kill*(1).

`sigexit`
An artificial signal 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.

Signals will be ignored if the signal routine is set to `{}`. Signals revert to their default behavior when their handlers' definitions are deleted.

22. 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 ASCII nuls. (This is not quite as horrendous as it sounds, 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*.

There is not currently a way on Plan 9 to place functions in the environment, although this could easily done by mounting another instance of `#e` on another directory. The problem is that currently there can be only one instance of `#e` per process group.

23. 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 -ddp
}
```

24. 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 pbd{
  /bin/pwd|sed 's;.*/;;'
}
fn cd{
  builtin cd $1 &&
  switch($#*){
  case 0
    dir=$home
    prompt=($ps1 $tab)
  case *
    switch($1)
    case /*
      dir=$1
      prompt=('{pbd}^$ps1 $tab)
    case */* ..*
      dir=()
      prompt=('{pbd}^$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.

Function `pbd` is a helper that prints the last component of a directory name. 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.

25. Examples — *man*

The *man* command prints pages from of the Programmer's Manual. It is called, for example, as

```
man 3 isatty
man rc
man -t cat
```

In the first case, the page for *isatty* in section 3 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 /n/bowell/usr/man || {
  echo $0: Manual not on line! >[1=2]
  exit 1
}
NT=n  # default nroff
s='*' # section, default try all
for(i) switch($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=man'$s/$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`.

26. 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.

Again, if the result is Holmdel, it prints a message and exits. Otherwise it goes back to the top of the loop.

```
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
Piscataway
Reading
Short Hills
South Plainfield
Summit
Whippany
West Long Branch
!
while(true){
   lab=‘{/usr/games/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
   }
}
```

27. Discussion

Steve Bourne’s `/bin/sh` is extremely well-designed; any successor is bound to suffer in comparison. I have tried to fix its best-acknowledged shortcomings and to simplify things wherever possible, usually by omitting unessential features. Only when irresistibly tempted have I introduced novel ideas. Obviously I have tinkered extensively with Bourne’s syntax, that being where his work was most open to criticism.

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’etre* 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 as 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 properly 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 exactly when you want a syntax character to appear in an argument. `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 security hole [Ree88]. *System*(3) and *popen*(3) 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 not ever 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 people's 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.

Its lexical structure is harder to describe. I would simplify it considerably except for two things. There is a lexical kludge to distinguish between parentheses that immediately follow a word with no intervening spaces and those that don't that I would eliminate if there were a reasonable pair of characters to use for subscript brackets. I could also eliminate the insertion of free carets if users were not adamant about it.

28. 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 [Bou78], as are most of *rc*'s good features.

29. References

Bou78. S. R. Bourne, ''UNIX Time-Sharing System: The UNIX Shell,'' *Bell System Technical Journal* **57**(6), pp. 1971-1990 (July-August 1978).

Ree88. J. Reeds, ''`/bin/sh`: the biggest UNIX security loophole,'' 11217-840302-04TM, AT&T Bell Laboratories (1988).