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.

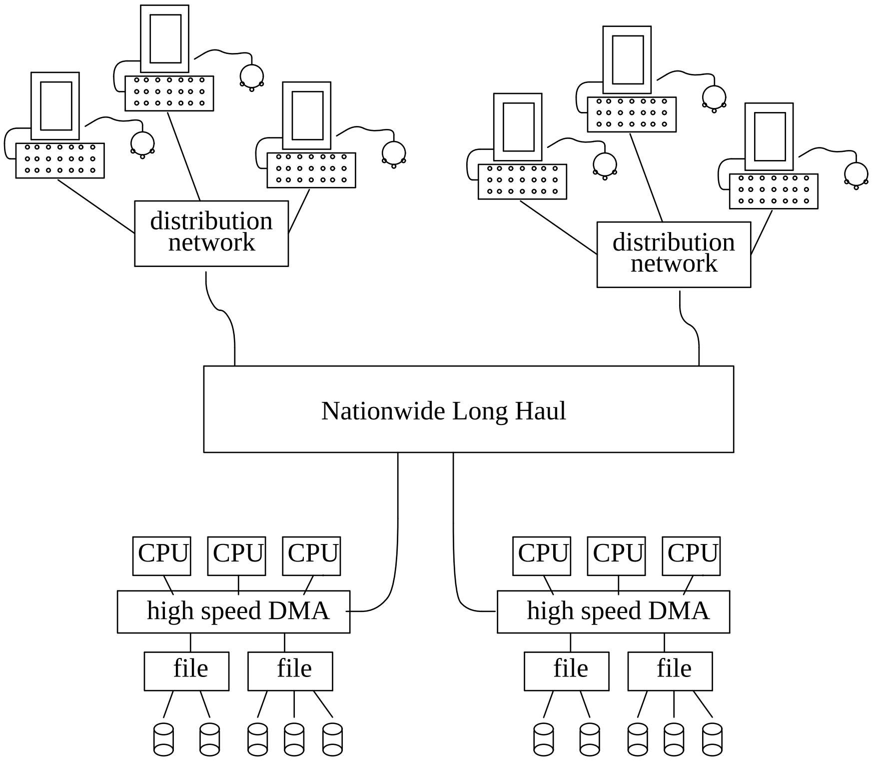

Figure 1 - Plan 9 Topology

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).