ANDREY MIRTCHOVSKI AND
LATCHESAR IONKOV

why some dead OSes still matter

Andrey Mirtchovski has been a Plan 9 aficionado since, as an undergraduate student, he found Plan 9's Third Release disks hidden in the waste bin of a university server room. Since then he has devoted a significant amount of time to the system, because it simply gets things done with less code.

andrey@lanl.gov

Latchesar Ionkov is a Linux kernel developer responsible in part for the 9p kernel module, which allows Linux to speak Plan 9's 9p protocol. Latchesar has been instrumental in translating Plan 9's ideas into mainline operating systems such as Linux.

lionkov@lanl.gov

WITH ITS CURRENTLY UNCHALLENGED ubiquity, the Linux operating system has become the de facto standard research OS in academia. The consequent waning of general systems research has been well documented [2], with the number of alternative operating systems in existence that significantly depart from the UNIX model and that are actively in development rapidly approaching zero. In this article we will discuss one such alternative operating system that has refused to disappear completely, and we will attempt to evaluate the features of that OS that make it suitable as a vehicle in our research projects and as a research environment.

We describe and share our experience working with the "Plan 9 from Bell Labs" operating system. This article will not attempt to compare the virtues of this OS with other, well-established systems. Instead, we will examine the environment Plan 9 provides as it pertains to researchers in academia, graduate students, and even undergraduates taking their first steps in exploring the inner workings of their first operating system. The goals of Plan 9 are completely different from those of other free and open source operating systems: Whereas some aim to provide a useful UNIX environment and others are bent on total world domination, Plan 9 aims to provide a useful research environment for building distributed software.

Plan 9 from Bell Labs

Plan 9 is a not-so-fresh offering from the same group at Bell Laboratories that created UNIX. Plan 9 has been in existence for over 18 years, making it a tad older than the popular free UNIX variants. The first Plan 9 papers were published in 1990 and since then there has been a relatively steady stream of research publications using the OS as their base [1]. Both the OS and its ideas are active parts in several current research projects, ranging from porting it to the largest supercomputer currently in existence to fitting it into small embedded devices that can be carried in one's pockets.

During most of its existence the OS has flourished within the confines of the lab it was created in, taking part as the core of most of the research projects therein. Outside of Bell Labs, however, Plan 9 has

failed to gain widespread adoption. The reasons for this are several and can mostly be traced to the battle for open source software that raged throughout the 1990s. The first Plan 9 release was made available to a select few universities in 1990. A second release, made available in 1995, required external organizations and individuals to purchase the OS and manuals at the prohibitively high cost of $350 for a binary-only license. In 2000, Plan 9 marked its third release, this time as an open source operating system provided by its creators with all source code and at no cost. (Manuals can still be purchased separately from Vita Nuova [5], a company in Great Britain that maintains and distributes the Inferno OS, a cousin of Plan 9.) Unfortunately, the license Plan 9 was released under was slightly restrictive, for example requiring users to indemnify the new Bell Labs owner, Lucent, from any future lawsuits. As expected, this ruffled a few feathers in the free/open source camp (see Richard Stallman's article describing the issues with the license of the OS [3]). The license was completely opened and OSI-approved by 2002, for the fourth release of Plan 9. Since then the system has moved from a single release cycle to a continuous-update one, where changes to the OS and supporting applications are made available immediately, so the OS release number has not been bumped, even though the system is still being developed.

In many aspects the Plan 9 operating system was ahead of its time. Its creators anticipated the level of penetration that networks will have in computing and aimed to create a distributed environment that provided services to programs and end users regardless of their physical location as long as they were connected to the network.

PLAN 9 IS STILL RELEVANT

Even though Plan 9 has existed for nearly two decades, it still holds some relevance for today's operating system landscape. Plan 9 was innovative in many areas and integrated novel and interesting solutions deeply within the system, but its relevance does not necessarily stem from the fact that it's such a great OS: It is not. Other OSes that came into being about the same time, such as Amoeba [4], were making even greater leaps into the wild, containing a multitude of ideas, many of which—such as independent-of-origin computation, resource pooling, and virtualization—are now becoming much more relevant to computing. Perhaps that very boldness in accepting new concepts in the OS made those systems less practical; most of them have died from lack of developers and fresh users. Plan 9's fate is sure to be the same, as it has a relatively stable community of around several hundred users, but no new blood is coming in—which is exactly the rationale for this article.

Impracticality should be an accepted death sentence for every bold new operating system. What is troublesome, however, is that all the ideas coming from this vast research in distributed operating systems in the early 1990s have been lost to the new generation of programmers, students, and researchers. Students now entering the system software research and development field are faced with the same pool of choices now as they were 20 years ago: The choice is always among various UNIX offshoots. In fact, for many undergraduates the deciding factor as to which operating system they will dedicate their best years has been political, rather than technical: They pick from the set of free and open source operating systems the one that most closely matches their moral beliefs. The gap in operating systems and system software that was supposed to be filled by all that research in the last decade before the millennium has been filled by bloated middleware.

Plan 9 managed not to be involved in this political game. It came from an older time when source was closed and, when it had to, it successfully converted, with a few hiccups, to an open model allowing everybody to peruse its code as they see fit. What is keeping Plan 9 alive is those users who keep an open eye for problems and an open mind about their solutions. The reason Plan 9 is still relevant even after its user base has dwindled is that the main purpose for its existence is to explore and to examine problems and solutions in a networked, distributed environment. We can do so easily and without much effort because Plan 9 was built on three basic design principles that cohesively absorb and tie various parts of the system together: simplicity, clarity, and generality.

Simplicity

The initial goal with which Plan 9's creators set out to develop the system was to fix the problems that they perceived were inherent in UNIX. The main task they had to tackle was to simplify the system by peeling off all the communication layers that had been built on top of the core to handle tasks never envisioned by its creators, such as networking (sockets) or graphical user interfaces (X11). Plan 9 presents the programmer with a single protocol upon which all remote and local communication is based: 9p. The 9p protocol allows resources presented by processes locally or remotely to be accessed as a hierarchy of files and directories (which themselves are files) with a few standard operations such as open, close, read, write, and stat. The 9p protocol permeates the system fully, with absolutely all communication except memory access occurring over it.

Besides the fact that 9p allows clients and servers to share resources via a very simple but effective protocol, 9p has an extra feature that makes it very appealing to use: It is not transport-dependent. Plan 9 uses the protocol over TCP, IL, a protocol created by Bell Labs with 9p in mind and without TCP's overhead, and RUDP, a reliable UDP offshoot with in-order delivery. In our work here at LANL we have also written libraries that allow 9p to be transmitted directly over the DMA mechanisms available in the Cell for communication between its separate processor elements. We are currently working on a library that allows 9p to be spoken over the PCI-express bus in our next-generation hybrid supercomputer, which combines Cell accelerator cards with Opteron hosts.

We must mention that, apart from the simple communication protocol, the rest of the Plan 9 system is also an exercise in simplicity, in both the number of lines of code and the number of programs used to accomplish a task. The Plan 9 kernel for the PC architecture (by far the most used and most developed) consists of around 90,000 lines of C code, not counting the drivers for various hardware, plus another 20,000 lines of portable C code, which is shared among all architectures.

Given that this code is both readable and understandable, it comes as no surprise that new students are able to pick up Plan 9 rather quickly. Unfortunately, some students, particularly the ones already familiar with other free and open source operating systems, have expressed distaste for such a simple system. In our experience, undergraduate students tend to be more impressed by, even enamored with, graphical bells and whistles and are unable to give the system's simple design proper consideration. Indeed, most will consider a windowing system implemented in only 7000 lines of code spartan. What those students miss is that Plan 9's creators eschewed complexity, which left us with a system that is both easy to understand and easy to modify.

Clarity

The main method of interfacing with other programs is through files. A client program opens a file served by a process and reads to receive information or writes to send information. Plan 9 does not have an equivalent to the ioctl() system call, which cannot be issued across a network owing to the reliance on a local pointer to pass data. Instead, most servers by convention serve a file named ctl at the top of their hierarchy, which allows clients to control not only I/O but the general behavior of the servers.

The power of this method of exposed interfaces is enormous, as witnessed by the following examples, which illustrate how Plan 9's simple concepts are tied together into a cohesive environment.

The familiar concept of a UNIX pipe (i.e., a communication path linking a reader and a writer) is implemented in Plan 9 via a file server: A kernel device serves a single-level tree containing two files, data and data1. Writes to data can be read from data1 and vice versa. Moreover, those two files can be bound anywhere in one's namespace and even imported from a remote system, removing the need for a special tool such as netcat.

Networking in Plan 9 is also implemented as a file server, or, rather, several different file servers, each corresponding to a specific protocol. Those file servers are mounted together in /net on a system, and each may serve one or more subdirectories for each connection to a remote machine. Because Plan 9 directory structures can be served remotely and mounted on a remote machine (as the next section illustrates), a /net from one server can be used as the main interface of another. Since most networks nowadays are heterogeneous, this concept has mostly been used to provide access to the outside world to machines hidden on an internal network. One computer will have two interfaces—an internal one and an external one—with all machines on the inside that need access to the Internet mounting its external /net. Thus there is no need for specialized NAT software and packet translation. There are no extra provisions made to allow one machine to use another's network stack; instead, this ability simply comes from the fact that network stack interfaces are presented as files and that files can be imported from remote computers.

An extension of this idea is a small program called sshnet, which can be used to import a remote computer's network interface via a connection secured and encrypted by the SSH protocol. Sshnet imports the remote machine's TCP stack and presents it to the local system as a network stack like all others in /net.

The window managers and graphics subsystem in Plan 9 are also file servers. The window manager, rio, presents its control and communication interface in /dev. Those files correspond to the copy/paste buffer (a text file called snarf) and the whole screen of the display, as well as subdirectories containing information about each window on the system, including the text typed in and, indeed, the graphical representation of the window. To take a screenshot of one's desktop, one would simply have to cat /dev/screen. Since they're all simply files, naturally one can import another computer's window manager files and have local programs draw on a remote screen without any additional software requirements or tweaks. Furthermore, since they are all simply files, the window manager can be run recursively in one of its own windows.

Another example of the benefits of using files is the copy/paste buffer served by the Plan 9 window manager: It is a simple text file that acts as a buffer for

text, keeping whatever is written to it for all readers. Naturally, to communicate with a more sophisticated operating system, this file is not enough. For example, when running Plan 9 under the Parallels virtual machine emulator on OS X, one is unable to have text copied on the host system be pasted in Plan 9. One ingenious solution to this problem is to export the host system's copy/paste interface as a file, which can be imported by Plan 9 and mounted on top of the file served by the window manager, thus being made available to all programs running within that namespace. The program to complete all this is a simple 100-liner in C which speaks the 9p protocol on a TCP/IP socket and knows how to query and set the copy/paste buffer in OS X. It serves a single root directory with a single file in it, replacing the Plan 9 window manager with the OS X one. This has proved much easier than having to create special hooks within both Plan 9 and Parallels to create an internal interface to the copy/paste buffer hidden from the eyes of user-level programs.

Generality

In Plan 9 every resource of a system, be it hardware or software, is presented as a hierarchy of files. The system provides two means of manipulating hierarchies: mounting a resource and binding two mounted resources together. These two functions are designed to improve and extend the ability of a Plan 9 user to construct a namespace (or the set of file hierarchies) relative only to the user's own interests and uses. In other words, a private view of what is available on the system (or networks of systems) is available to be customized by individual user-level processes on the fly.

Mounting carries the same functionality as it does on UNIX systems and their descendants: A connection with a local or remote resource is initiated, the resource is attached to the issuer's namespace (the directory hierarchy starting at root), and standard file operations pertaining to the mounted directory are sent over the connection to the server. In Plan 9, however, a mount can be accomplished by anyone without requiring superuser privileges and without affecting the file hierarchy of other users. This allows, as will be shown later, a user to import a remote server's network interface (à la VPN) in one terminal on their desktop without affecting any other users on the system or even programs running in other windows on the same desktop. Importing a remotely served file system is trivial: A connection is made to the remote system, and the file descriptor from that connection is mounted locally. The 9p protocol is designed to handle authentication security transparently to all programs accessing resources on the system.

Binding in Plan 9 is the ability to join two or more directories together in the same hierarchy. This does not have a direct analog in UNIX, but it has turned out to be very useful in Plan 9. It allows one, for example, to build a source tree in a directory that does not allow the user to write to it, without having to copy it to a temporary place. In another example, Plan 9 has only a single directory for binaries, called /bin. When a user logs in, all binaries for the particular architecture that he or she is using, all scripts (global and local for that user), and all binaries that the user has installed privately are bound to /bin. The shell's equivalent of $PATH in Plan 9 contains only a single directory, /bin (see Figure 1).

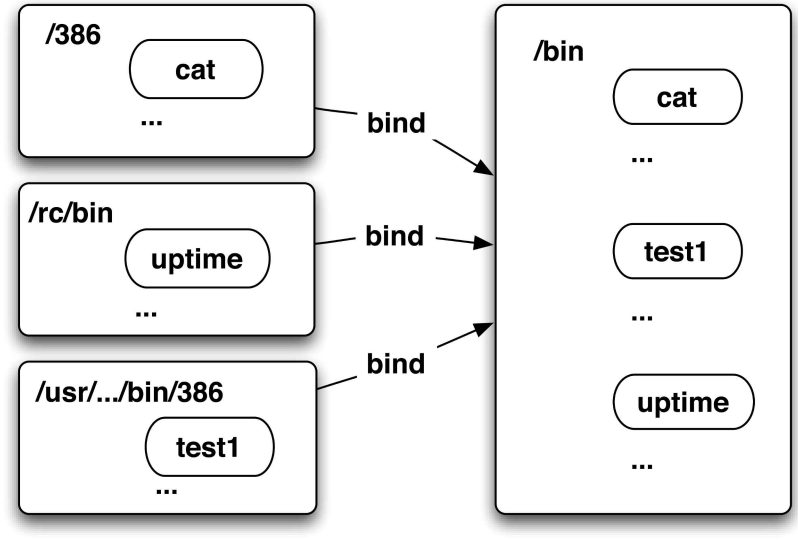

FIGURE 1: THE VARIOUS DIRECTORIES CORRESPONDING TO THE PARTICULAR ARCHITECTURE (IN THIS CASE, X86), SHELL SCRIPTS (RC), AND BINARIES COMPILED LOCALLY BY THE USER ARE BOUND TOGETHER IN A UNION DIRECTORY IN THE OTHERWISE EMPTY /BIN, THE ONLY DIRECTORY THAT THE SHELL TRAVERSES IN ITS PATH TO FIND BINARIES ON PLAN 9.

Perhaps the most striking example of the generality of Plan 9's ideas and how they tie together is the cpu command, which is used to connect to remote Plan 9 servers. We are used to thinking about such programs as gateways to remote machines. For example, when we ssh to a server we throw away everything on the local machine and accept the remote environment as our own. (The hack to tunnel X11 communications through SSH connections is the exception that proves the rule.) Plan 9's cpu command, however, makes it possible to connect not only two ports but two environments on two separate machines together. cpu serves the local namespace of the machine from which it was started under a directory (/mnt/term) on the remote machine. Since everything in plan 9 is a file, from there a script can automatically bind important files from the local system to where they should be expected by applications on the remote one. For example, the audio device is bound from /mnt/term/dev to /dev to allow any remote application to play audio to the local speakers. The same thing happens with the mouse and keyboard files and the graphical subsystem; thus any application run on the remote machine can draw to the local one. You can imagine this being extended for all devices: Via a simple one-line user command requiring no superuser privileges, one can, for example, print from any machine to a local printer.

As a consequence, Plan 9 system administrators do not set accounts and modify access lists for applications; instead, they modify file permissions, fully aware that if a user has permission to access a particular resource on a local system, that user can do so from any other computer that can connect to it.

EXAMPLES OF PLAN 9 IN THE REAL WORLD

The ideas stemming from Plan 9—namely, that all resources should be made available as files that can be accessed remotely by attaching them (or mount-

ing, which is the more widely accepted term) to a user or to a process's namespace—have been put in use in various forms here at the Los Alamos National Laboratory's Advanced Computing Lab and elsewhere. Once one becomes acquainted with the system's simplicity and, dare I say, beauty, it is not very hard to convert ideas to fit the Plan 9 model of development. This has saved us a tremendous amount of time in developing and testing new protocols, implementing clients and servers, measuring performance, and scaling. The fact that we have based our tool on Plan 9's ideas, but have not directly used Plan 9 in some of the cases, is another benefit of Plan 9's design ideas and their flexibility: One need not port the entire OS but only one basic element, the protocol, to allow one to benefit fully from the research that went into the system. In all cases Plan 9's ideas have given us the ability to rethink from the ground up the basic ideas and principles on which our software was based, redesign it in a new model, and implement something that, if not much faster than its predecessor, was much simpler to understand and fix. Plan 9, in this case, served only as the "mind expansion" enzyme to further our work.

Some of our work, which is strongly based on Plan 9's ideas, includes:

- Xcpu: a job starter for extreme scale clusters. We created Xcpu at LANL in order to be able to start and control jobs on the next generation of high-performance computing platforms, which are starting to appear on the horizon. These machines will consists of tens of thousands of heterogeneous nodes. Xcpu presents a file server interface to starting jobs on a remote machine. This interface includes a directory for copying binary and data files; control files for starting, stopping, and continuing an application; and control files for attaching the application to its standard I/O and monitoring its progress remotely.
- CellFS: a new programming model for the Cell Broadband Engine which allows its accelerated components (also known as SPE units) to communicate with the host processor and its memory via POSIX-like file-based operations. Since DMA transfers to the host processor are encapsulated in 9p, the programmer simply issues an open request via a library call to access and a read request to fetch data from main memory.
- KvmFS: via 9p, provides extended access, startup, and control of virtual machines running on compute nodes across the cluster. Again, using a simple set of file operations, administrators can set up a virtual machine, transfer its image to a node on the network, start it up, and control its execution (including migrating it to a third node over the network).

There are numerous other toy programs and prototypes in which we have found the 9p protocol and the Plan 9 way of thinking, "everything is a file," to be of great help in simplifying and breaking down the problem space into its components. We believe now that the ideas we have learned from Plan 9 are applicable in wide areas of our research.

Conclusions

We are not trying to make the argument that Plan 9 should be considered by any and all academic researchers and students for their work. However, by listing here the ideas of Plan 9, we hope to invite students and operating systems programmers to keep an open eye for ideas and implementations. By showing what is possible with a little imagination and creativity, we described a system that will not be easy to conceive with ideas coming from

the single-track mind of the successful, but antiquated, creations in the "real world." We invite students and professors to look around and dig deep into the history of operating systems, especially the ones that never became commercial successes because they were "too far out there." There is a great deal of research that went into new operating systems before the world got locked into commercialization, Plan 9 being only one such example, but without examining and evaluating those, we're bound to continue making one mistake over and over: that of complexity. By building layer upon layer of interfaces designed to hide and sidestep the shortcomings of the underlying system, we are putting ourselves into the corner of incremental research, where our goal becomes that of improving what's already there instead of throwing it away and replacing it with something better, guided by our experience and the ideas of others.

REFERENCES

[1] Plan 9 from Bell-Labs papers: http://plan9.bell-labs.com/sys/doc/.

[2] R. Pike, "System Software Research Is Irrelevant": http://herpolhode.com/rob/utah2000.pdf.

[3] R. Stallman, "The Problems of the (Earlier) Plan 9 License": http://www.gnu.org/philosophy/plan-nine.html.

[4] A.S. Tannenbaum et al., "Experiences with the Amoeba Distributed Operating System," *Communications of the ACM,* vol. 33, pp. 46–63, Dec. 1990.

[5] Vita Nuova: http://www.vitanuova.com.