Process Control in a Distributed Berkeley Unix† Environment

Ramón Cáceres

Computer Systems Research Group‡
Computer Science Division
Department of Electrical Engineering and Computer Sciences
University of California at Berkeley

ABSTRACT

The Berkeley Unix operating system is evolving into a more distributed computing environment. As such, applications written for it will increasingly consist of programs that spawn several processes, processes which may reside on more than one machine. A distributed system must provide facilities through which the processes that are part of such distributed computations can be controlled. There must be a way of terminating these processes, suspending their execution, restarting them, and otherwise notifying them of asynchronous internal and external events that require their attention. Unix signals, which can be viewed as software interrupts, allow processes to be controlled in this way. In the current implementation, however, their use is limited to processes within a single host. There must also be available facilities for debugging-style process control, such as inspecting and modifying a process's address space. Debugging-style process control is currently provided by *ptrace()*, which is also very limited in the context of distributed process control.

The process control scheme presented here allows processes to send each other process control information through the Unix interprocess communication facility. Processes acquire an endpoint for communication, or *socket* in Berkeley Unix terminology, that is used to receive the control information for the process. The use of a general message delivery system such as Unix interprocess communication permits a set of process control mechanisms to act among any set of processes, even when such processes reside on different machines.

† Unix is a trademark of AT&T Bell Laboratories

‡ This work was sponsored by the Defense Advanced Research Projects Agency (DoD), ARPA Order No. 4031, monitored by the Naval Electronics Systems Command under contract No. N00039-C-0235. The views and conclusions contained in this document are those of the author and should not be interpreted as representing official policies, either expressed or implied, of the Defense Research Projects Agency or of the U.S. Government.

Table of Contents

0. Introduction

Modern multiprogramming operating systems rely on the notion of a process to separate distinct units of task execution existing within them. In such systems, certain urgent events take place which force the interruption of normal process execution before most processes can run to completion. Some of these events, such as page faults and scheduling decisions, are taken care of by the operating system and do not involve the process code directly. However, the necessity to notify a process of certain other unexpected events requires that process control mechanisms be embedded into operating systems. These process control mechanisms typically provide ways for a process to be made aware of asynchronous events which can be internal or external to the process. They allow for process execution to be halted, restarted, or permanently terminated.

Another type of process control available on most systems is that which supports debugging activities. Here, the goal is to allow a user to supervise the progress of a program executing in the context of a particular process state. Some of the actions provided by debugging-style process control, namely starting and stopping a process, are common to both styles of control. However, debugging support must also include the ability to inspect and modify the address space of a process. This allows the debugging agent to look into the status of the program in question, and to set breakpoints where needed.

The steady evolution of computer systems towards distributed computing motivates the search for extended process control facilities, ones that deal effectively with the new distributed environment. Present systems are sorely lacking in such facilities, and new ones must be developed to meet the demands of future systems.

In a distributed system single programs will often be composed of several communicating processes, with these processes residing on different machines. Such distributed programs constitute an important asset to the system because without them the system could not take advantage of the distributed nature of its resources. Facilities that can effectively control distributed programs would therefore be a very useful part of a distributed system, both during the evolution of the system and later when application programs that use the new environment are written.

This report focuses on ways through which the Berkeley Unix process control facilities can be extended to a network of machines. Both general-purpose and debugging-style process control mechanisms are considered. This is part of a wider effort to provide an effective set of debugging tools for a distributed programming environment.

The scheme being offered allows processes to send each other process control information through the Unix interprocess communication (IPC) facility. Each process acquires an endpoint for communication, or *socket* in Berkeley Unix terminology, which is used to receive the control information for the process.

In addition, the report will address the obvious security problems that arise under such a scheme, since any process in the system could potentially exert control on any other process unless appropriate precautions are taken.

Naming schemes are also a consideration. At present, processes are named by an integer, or *process id*, which is unique within a host. However, this process id is not guaranteed to be unique within a cluster of machines. Therefore, an appropriate method of naming processes in a distributed environment must be agreed upon.

Section 1 provides the background for the ensuing discussion by describing the existing processing environment and process control mechanisms of Berkeley Unix 4.2BSD. In Section 2, some previous approaches to the problem of extending the Unix process

control facilities to a distributed system are presented. Section 3 introduces the new schemes, which use Berkeley Unix IPC to deliver process control information. This is followed by Section 4, where the proposed approach to using Berkeley Unix IPC for controlling processes is described in detail. Finally, Section 5 makes some suggestions as to what steps should be taken to further incorporate the process control scheme of Section 4 into Berkeley Unix.

1. Processes and Process Control in Unix

1.1. Unix Processes

A *process* is often informally defined as a program in execution. More specifically, a process can be considered as a single thread of instruction execution in the context of a well defined execution environment, or state. The concept of a process was used extensively for the first time by the designers of the MULTICS[Daley68] system and, independently, by the group that designed the THE[Dijkstra68] system. In Unix, processes exist as the fundamental units of task execution.

Processes should not be equated to programs. A *program* is a static entity, while a process is active and always changing. A program is a set of statements that describes how execution should proceed, and as such can be thought of as part of the environment of a process. On the other hand, with the help of the operating system, a program may divide itself into two of more lines of control, each with its own distinct process state. In this case, the program will be associated with more than one process and can be termed a *multiprocess program*.

Unix programs can split up in this manner by issuing the system call *fork(2)*†. *fork()* creates a new process (the child) that is an exact copy of the calling process (the parent). The two processes have identical initial states, including the code they will execute. Because of this identity, there must be a way for each process to determine its individuality, otherwise the forking mechanism would not be very useful. This is provided by the value returned from the *fork()* call. *fork()* returns the unique process id of the child to the parent process, and 0 to the child process.

Unix processes run in a very rich environment. In addition to closely related processes in the process hierarchy, a process's state contains objects such as the files and the sockets that it is presently reading or writing (sockets are endpoints for interprocess communication; they will be described later). These open files and sockets are accessed through a table of object descriptors that each process holds as part of its state. The first three descriptors in this table are said to point to the standard input, standard output, and standard error files, respectively.

Another important part of the environment of a process is the Unix kernel. The kernel provides a process with all the functionality usually associated with operating systems. This includes access to the file system and interprocess communication. Kernel functions are available to a process through a uniform interface: system calls.

Increasingly, Unix programs consist of multiple cooperating processes. This is sometimes done to exploit some inherent parallelism in the algorithms involved. Other times, this is done to take advantage of distributed system facilities by having the

† First instances of Unix commands and system calls appear in italics, with the section number of the Unix Manual[UserManual],[ProgrammerManual] in which they appear inside parentheses. Succeeding references to these programs and routines will also be in italics, but their section numbers will be omitted.

different processes reside on different hosts in a network. In any case, there must be a way for these processes to synchronize and communicate among themselves. In Unix, this task is performed by the IPC facility.

Process control mechanisms must be available that allow a process to be notified of asynchronous internal and external events, such as the execution of an illegal instruction or an interrupt from a user terminal. Such mechanisms must also allow a process to carry out debugging related actions on another. These include stopping and restarting the other process, and inspecting that process's address space. These activities are an essential part of any system, and in Unix are made possible by the *signal(2)*, *ptrace(2)*, and *wait(2)* facilities, described below.

1.2. Unix Signals

Signals are one-word (32 bits) chunks of data that are delivered by the kernel to a process to notify it of some asynchronous event. They are the software equivalent of hardware interrupts. Signals are sent between processes by the *kill(2)* system call. The sending process specifies the signal number of the signal to be sent together with the process id of the receiving process. After authentication (only processes with the same user id can send each other signals), the signal is added to the state of the receiving process. The next time this process is scheduled to be run, the scheduler will note that there are signals pending and notify the process.

Received signals can be ignored or caught and acted upon by the process, with some exceptions. Signals can be ignored by constructing a bit mask of those signals that are to be disregarded. For signals that are not ignored a process can specify, in the form of a signal handler, the steps to be followed when a particular signal arrives. If no handler is specified, the default action is taken; usually the receiving process is terminated.

Unix signals are an efficient means of process control, but they have one great disadvantage: they cannot be delivered across machine boundaries. Thus, even though their functionality is effective in carrying out most of the process control primitives mentioned earlier, their present implementation is severely lacking in the context of controlling distributed processes.

1.3. The Ptrace Facility

The Unix system call *ptrace()* provides a way for a process to control the execution of another process on a finer level of detail than that allowed by signals. *ptrace()* also allows the controlling process to examine and change the core image of the other process. *ptrace()* is mainly used for the implementation of breakpoint debugging in utilities such as *adb(1)* and *dbx(1)*.

A typical debugging session using *ptrace()* will follow the steps outlined below. First, the debugger process forks a child process. The child process then calls *ptrace()* with a 0 argument in order to initialize the debugging activity. This is done before the child changes its core image in order to execute the program to be debugged instead of just another instance of the debugger. The system call *exec(2)* is used to overlay a new executable image on top of the previous one.

Once *ptrace()* has been initially called, the debugger process places a series of calls to *ptrace()* which allow it to control and debug its child. The debugging session proceeds with the debugger periodically stopping and starting the debuggee, examining the debuggee's address space, modifying this address space, and inserting breakpoints in the code segment of this address space.

Unfortunately, the *ptrace()* facility has several drawbacks that make it a very limited and inappropriate mechanism for distributed process control and debugging.

First, only the immediate parent of a process in the process hierarchy is allowed to collect status information for the process. A debugger typically sends a control signal to the debuggee, waits for the process to stop by calling *wait(2)*, and then collects its status. It is the structure of the *wait()* call which imposes the restriction that a process may only be notified of changes of status in an immediate child. This limitation restricts the program to be debugged to only one process, since any children of the process being debugged will be unaccessible to the debugger. Distributed applications, however, are almost by definition multiprocess programs, and therefore the mechanism fails entirely where such programs are concerned.

Second, a process must agree explicitly to be debugged and must itself initialize the debugging session. This precludes dynamic binding of the debugger and the debuggee. There seems, however, to be no fundamental reason why a debugging session cannot be initiated "on the fly" provided the right environment exists for debugging. This could occur without explicit initialization by the process to be debugged and after this process has started execution.

In addition, the communication channel established by *ptrace()* has very low bandwidth and yet is very expensive. Two context switches (debugger-debuggee and back again) are necessary to transfer every word of data, since no multiple-word messages are allowed. The low bandwidth comes from the fact that a four word structure is allocated *per system* to carry *ptrace()* data. Therefore, all debugger processes (e.g., *adb*, *dbx*) must synchronize on this one data structure in order to guarantee that only one of them is using it at any one time. Again, there is no fundamental reason for this restriction that only one debugging request may be serviced within one system at any one time.

Finally, the process to be debugged must be put on a stopped state before it can be examined, usually by sending it a signal. However, that process may be ignoring signals (e.g., sleeping forever on a locked inode), in which case the complete debugging facility can fail[Killian84].

2. Process Control in a Distributed Environment - Previous Work

2.1. The Rexec Facility

In 4.2BSD, the *rexec(3X)* call permits a process to create another process on a remote host and to control its execution through signals. The original process specifies the machine where the new process is to be executed, and the file name of the executable image to be run. After authentication and process creation, the controlling process is returned a stream socket connecting it to the remote process. In turn, the remote process has its standard input and standard output directed through the other end of the IPC connection. Thus, the original process can send data to the new process's standard input through the connection and read its standard output in the same manner.

It is also possible to set up a second IPC connection between the controlling and the remote processes. This other connection is used for two purposes: to make available to the controlling process the standard error of the remote process, and to send signals to the remote process.

The *rexec()* approach uses a combination of mechanisms for distributed process control. It uses a daemon (the *rexecd(8C)* daemon) to initiate the remote execution and to set up the IPC connections. It also uses the IPC mechanisms to deliver control information across machines.

This facility has many limitations, especially in the area of process control. First of all, if the remotely executed process forks, there is no provision for sending signals to the child process, or any other control information. Thus, the fact that the remotely executed process can receive signals across machines is only a half-way solution to the distributed control problem. In addition, the *rexec()* mechanism does not include any debugging support such as a way of delivering data associated with *ptrace()*. Lastly, the remotely created process is restricted in what it can do. It cannot be considered a general purpose process because its standard input, output, and error are always tied to the sockets that connect the process with the controlling process on the originating machine.

In the light of these shortcomings, it is apparent that *rexec()*, while useful in many applications, does not create an appropriate environment for general remote process execution and control.

2.2. Processes as Files

Another approach to improving the existing Unix facilities for process control and debugging is to present processes as files. Processes as files were introduced to Unix Version 8 by T. J. Killian at AT&T Bell Laboratories[Killian84]. This solution is also suggested in the Berkeley Unix 4.2BSD manual page [ProgrammerManual] for *ptrace()*. It allows for many of the functions provided by *ptrace()* to be performed in a much more elegant way, very much in accordance with the rest of the Unix system.

The system introduces a new file system, */proc*. Each file in this file system corresponds to the address space of a running process. The names of the members of this file system are of the form */proc/nnnnn*, where *nnnnn* is the five digit unique process id of the process the file corresponds to. At present, *nnnnn* cannot be larger than 30,000, a limitation inherited from the PDP-11.

The entries in */proc* are accessed via the standard file I/O system call interface. A typical series of system calls will start with *open(2)*, which opens the */proc* file transparently, so that the process being affected is unaware that its address space has been opened for inspection and/or modification. The *open()* call can be followed by any combination of *read(2)*, *write(2)*, and *lseek(2)*. *lseek()* allows random access to the address space, while data may be transferred to or from the locations reached through *lseek()* by calling *read()* and *write()*, respectively. Finally, when no further operations are necessary, a call to *close(2)* will commit any changes made to the address space.

Process control in the */proc* implementation is possible through *ioctl(2)* calls. The requests available through *ioctl()* implement a superset of the operations performed by *ptrace()*, but with a more natural and elegant protocol. Among the operations possible through *ioctl()* requests are: fetching the kernel process table entry for a process (PIOCGETPR), and obtaining a read-only file descriptor for a process's executable image (PIOCOPENT). To these are added the equivalents of the *ptrace()* operations in the */proc* scheme. These operations include starting and stopping a process, and waiting for the debugged process to stop because of a breakpoint or other event (PIOCSTOP, PIOCRUN, and PIOCWSTOP, respectively).

The majority of the security problems associated with processes as files are automatically taken care of by the protection mechanisms in the file system, such as permission mode bits. This is a very attractive feature of the */proc* scheme.

The */proc* directory and its contents are used by a window-based interactive debugger developed by T.A. Cargill, also at AT&T Bell Laboratories. Because of the advantages of the processes as files approach over *ptrace()*, Cargill's debugger (named *pi*, for process inspector) avoids many of the shortcomings of ptrace-based debuggers. For

example, it can acquire multiple processes dynamically, without the explicit participation of the processes being affected. Binding the debugger to a process is simply a matter of calling *open()* on the appropriate */proc* file.

In spite of its elegance and semantics that closely agree with the Unix philosophy, the processes as files approach has some shortcomings in the context of distributed process control. First, the scheme as described in [Killian84] does not support */proc* objects or any of the operations on them across multiple machines. This can be corrected by incorporating the mechanisms into a distributed or remote file system. A distributed file system is implemented in the LOCUS system[Popek81],[Walker83], and work is near completion on a remote file system for Berkeley Unix. Peter Weinberger's Network File System (NFS) for Unix Version 8 at Bell Labs[Weinberger84] would also allow the */proc* directories to be accessible from any machine on the network. However, as things stand, the *ioctl()* facility is not yet operational across machines in NFS. This is not expected to be too difficult to correct, and once *ioctl()* works, all the operations on */proc* described above will follow trivially.

A suitable naming strategy must also be found to allow an extension of */proc/nnnnn* file names to a distributed context, while avoiding naming conflicts between processes residing on different hosts in the system. In the */proc* approach, this responsibility falls with the file system. When NFS is used, the naming across machines is of the form */n/machine/proc/nnnnn* (*n* stands for *network*, and *machine* for the name of the host where the particular */proc* directory actually resides).

Furthermore, the approach described in the paper does not completely address the remote signal delivery problem. As was explained above, delivering signals across machine boundaries is an important requirement of an effective distributed Unix system. Again, this can be easily fixed, this time by creating new *ioctl()* services that perform actions equivalent to the delivery of certain signals. This has recently been implemented with the PIOCKILL request, which sends a particular signal to a process specified through a */proc* pathname. Adding signals to the list of available *ioctl()* calls has the advantage that they will also extend to several hosts once the necessary steps are taken to perform *ioctl()* requests across machines.

More importantly, the */proc* solution does not provide a way for users to keep track of their processes if some of these processes do not reside on the same machine as the one the user originally logged into. Just as the *ps(1)* command is widely used today in single-host Unix systems, it will be important in a distributed system to provide the user with a way of finding out the status of processes belonging to that user, regardless of where these processes reside. As things stand, it would be necessary to search for processes with a particular user id through the */proc* directories or kernel process tables of all the machines in a network in order to collect the necessary information for a *ps()*. Clearly, this would be prohibitively expensive for any reasonably sized network.

Coupled with the fact that a */proc* implementation is not available as a base for further work on Berkeley Unix 4.2BSD, this last drawback justifies the pursuit of other solutions to the distributed process control problem. The following sections describe an approach that uses the Unix IPC mechanisms to implement new process control facilities for Berkeley Unix 4.2BSD.

3. Process Control Through Unix Interprocess Communication

3.1. Interprocess Communication in Berkeley Unix 4.2BSD

Unix interprocess communication (IPC) originally consisted only of pipes[Ritchie74], which allow processes to send each other data in a very simple and elegant way. Pipes, however, are limited in that they can only act between processes with a common ancestor in the process tree. Furthermore, they are only one-way channels between processes, and setting up two-way communication between processes usually requires two pipes.

Berkeley Unix 4.2BSD contains new interprocess communication facilities[Leffler83a],[Leffler83b],[Sechrest84]. The basic abstraction is the socket, which is an endpoint for communication to which a name may be bound. IPC can take place in two domains: the Unix domain and the Internet domain. In the Unix domain sockets are named with Unix pathnames; these names appear in the file system. At present, communication in the Unix domain is restricted to one host because file systems do not extend across machine boundaries. This situation is expected to change soon with the advent of remote file systems. The Internet domain, on the other hand, supports the addressing schemes and protocols described in the DARPA standards[Postel79],[Postel81a],[Postel81b]. As such, communication in the Internet domain can take place between processes on one machine, on different machines on a local network, and even between processes scattered throughout an internetwork.

Two styles of communication are available: datagrams, and stream communication. Stream communication is the style used by pipes. It is useful to think of the difference between these two styles by relating computerized communication channels to their human equivalents[Wecker79].

Much like letters and parcels in the postal system, datagrams consist of chunks of data transmitted devoid of any interrelation. Sequential or reliable delivery of these datagrams is not guaranteed by the protocol. Rather, providing this functionality is left up to the user of the datagram protocol. Datagram protocols are thus inadequate for some applications, but they are fast and efficient.

Pursuing our analogy between computer and human communication, stream or virtual circuit protocols can be compared to telephone conversations. A stream protocol establishes a sequential relationship between the blocks of information sent with the protocol so that no data arrives out of order, as in telephone communication. Furthermore, it guarantees the delivery of all the data sent. Since messages must be sequenced, a connection relating the messages to each other and effectively forming a circuit between the two parties in the communication has to be created (hence the virtual circuit nomenclature). Stream protocols are generally very useful, but they are expensive because of the number of services provided.

The ability of the Berkeley Unix IPC mechanisms to perform communication tasks between any two processes in a system suggests that they might be used to carry control information, such as signals and data of the type used by *ptrace()*. It is this idea that is carried forward in the new process control schemes described below.

The use of a general message delivery system such as the Berkeley Unix interprocess communication facility would permit a set of process control mechanisms to act between any two processes, even when such processes reside on different machines. In addition, this facility provides a high bandwidth channel for the transfer of debugger-related information such as portions of the address space of a process. These are marked improvements over *ptrace()*.

3.2. Omnipresent Control Sockets

3.2.1. Functional Description

With the idea of using the IPC mechanisms to carry process control information, Stuart Sechrest proposed to add to the state of each process a socket to which other processes could send control data. This socket was termed the *control socket* of a process. Figure 3.1 depicts where the control socket fits in the process state. Under this scheme, both signals and ptrace-type information were considered forms of interprocess communication, where the receiver would be forced to take appropriate steps to carry out the action requested by the message.

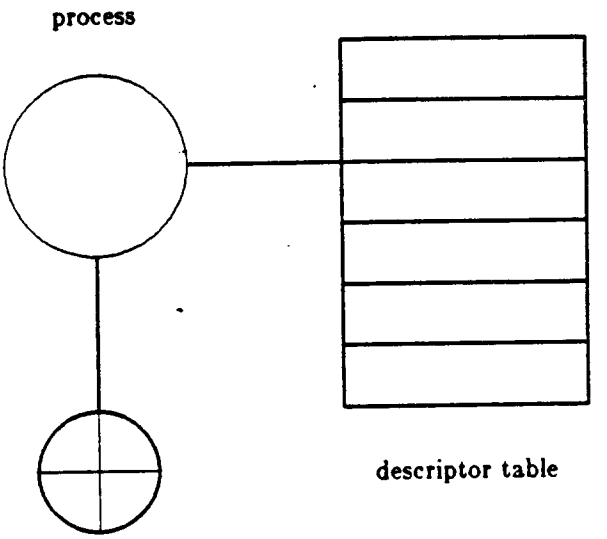

Figure 3.1 - The Control Socket

Keeping in mind what was considered to be the main shortcoming of the Bell Labs system, the control socket scheme included automatic reporting of important process events to some supervising agent. For example, when a process forked, the process id, host machine, and control socket of the new process was made known (again through the use of IPC) to a controlling process. This allows it to keep track of the components of a multi-process, and potentially multi-host, computation.

The first attempt to incorporate the control socket concept into Berkeley Unix involved giving each newly spawned process a control socket already created and ready to receive data. Such omnipresent control sockets are depicted in Figure 3.2. In order to provide many of the semantic advantages of the Bell Labs */proc* approach, it was decided to incorporate the notion of a control socket in the file system. This was made possible by creating the control socket in the Unix domain and binding a Unix pathname to the socket. Thus control sockets, and in a sense processes, were made to appear in the directory */proc*. The name chosen for each member of */proc* was, as in the Bell Labs scheme, the process id of the process involved.

All the control data for a process, including signals and ptrace-type data, would be delivered through the control socket. As has been pointed out earlier, the generality of the IPC mechanisms allows for this information to flow between any two processes, even when such processes reside in more that one host on a network.

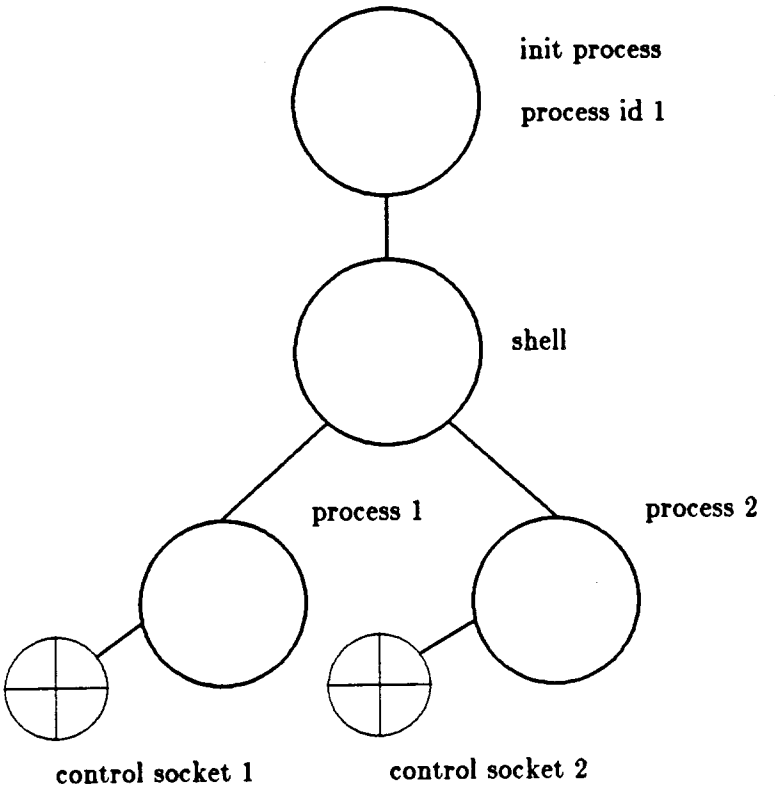

Figure 3.2 - Omnipresent Control Sockets

3.2.2. Initial Implementation

Implementing this scheme required several changes to the Unix kernel. First, each process had to be given a handle on the structures that comprise its control socket. This was done in the user area (*u.area*) of each process, where fields were added to hold the necessary information. The choice of the *u.area* was made because this area is readily available to every process as it executes in system space, when all the accesses to these structures would be made. The *u.area* of a process is not, however, easily accessible to other processes. This is not a drawback because the well known Unix pathname (*/proc/nnnnn*) of the socket is all that external processes need to know to send messages to the socket.

As mentioned in Section 1.1, new processes are created in Unix by the *fork()* system call. Since control sockets are associated with every process in the system, and these sockets must be ready to receive data as soon as the process comes into being, the *fork()* call was the logical choice for where to construct the control sockets. Thus a Unix domain socket was created and then bound to the pathname mentioned above within the *fork()* operation.

When a process dies, it is necessary to free all the system resources it holds. The system call *exit(2)* is executed whenever a process is to terminate execution. It is within *exit()* that resources such as the file descriptors held by a process are returned to the system. It was also there that the control socket and associated structures were deallocated so that they might become free when the process disappeared.

Control messages, those addressed to a control socket, needed to be intercepted in the kernel before they were delivered to the user process. This is because the requests these messages bring necessitate kernel actions on behalf of the controlling process. Therefore, another section of the kernel that had to be modified was the IPC delivery

code. The code was changed to recognize messages meant for a control socket and to interpret their contents according to a well-defined format.

For example, one message type was reserved for Unix signals. When the message in transit consisted of a signal, the IPC code would handle the message by delivering the appropriate signal to the process to which the message was addressed. Thus, signal delivery was transferred from the older mechanism to Unix IPC.

Presently, the Unix domain is restricted to one machine. However, work near completion will extend the 4.2BSD file system and thus the name space to a network of machines. It will not be difficult, and indeed desirable for many applications, to extend all present file system operations to work under the remote file system. This should include access to special files such as terminals, *ioctl()*, and, of course, operations on Unix domain sockets. The solution for remote signal delivery would then come for free once Unix domain IPC works across machines.

Once the system was creating control sockets for processes, an *ls(1)* of the */proc* directory showed a list of existing processes on the machine were the command was run. The next step was to test the new signal delivery mechanism. A program very similar to the Unix *kill(1)* command was written. The program, named *cskill*, took as arguments a Unix signal number and a process id, and proceeded to send the requested signal to the specified process via the control socket of the process. This program allowed processes to be started, stopped, and terminated (among other things) through the new process control facilities. It used Berkeley Unix IPC to deliver signals instead of the usual kernel channels.

3.2.3. Evaluation

At this stage of implementation and experimentation, several shortcomings of the omnipresent control socket approach became apparent. The most important objection to this scheme is related to the well known "making the many pay for the needs of the few" dilemma that has faced many system designers. Here, the creation of a socket and the binding of this socket to the Unix file name space added considerable cost to the operation of forking new processes. Creating new processes in Unix is already quite expensive due to the large amount of state with which they are associated. Considering that only a small percentage of all processes need debugging operations performed on them or make use of remote signals, incurring these costs for every process was unacceptable.

This prompted the search for another solution that would avoid adding complexity and expense to operations such as *fork()* unless it was strictly necessary. The resulting scheme, which uses on-demand control sockets, is outlined next.

3.3. On-Demand Control Sockets - The Process Manager Approach

The second attempt at incorporating the control socket notion into 4.2BSD involves the dynamic creation of control sockets as the need for them arises. This frees the majority of processes from the burden of the added distributed process control facilities, but provides the added functionality to those processes that need it. With on-demand control sockets, local signals are handled through the normal channels. Only remote signals and ptrace-type data are delivered through control sockets. This carries further the notion that the expense of setting up communication through control sockets should only be incurred when there is a real need.

In common with many aspects of computer science, providing dynamic facilities such as these implies more flexibility at the cost of added complexity on the part of the system. In this case, since control sockets are not always available in a well known and accessible

place such as the */proc* directory, the system must go through some contortions to initialize itself when the need for control sockets arises.

As with the omnipresent control socket scheme, it is necessary to keep track of the status of a distributed computation through notification of important events to some agent. The agent can thus keep a table describing the tree of processes that has been formed as a user's processes fork, exit, and possibly execute processes on other machines. With on-demand control sockets this agent must take on the added responsibility of dynamically setting up control sockets on different processes, and of routing remote signals and ptrace-type data to the appropriate control socket. This agent is termed the *process manager*.

The process manager with on-demand control sockets is the subject of Section 4. There, more information on this approach is presented, together with some implementation details.

4. A Process Manager Implementation

As mentioned in the previous section, a scheme that dynamically creates control sockets has many advantages over the omnipresent control socket approach. This section describes in detail the design and implementation of the process manager scheme with on-demand control sockets.

There is one process manager per user login session; it is itself a distributed program. A master process manager process is always created as part of the login procedure, and slave process managers are created on demand on those hosts to which a user's processes spread. An IPC connection is maintained between the master process manager and each of the slaves to allow them to easily exchange information. Stream communication is warranted in this case because the master needs frequent access to a communication path to its slaves, and because of the need for reliability across machines. In the remaining of this report, references to the process manager are to the combination of the master and slave process managers, unless explicit mention of master or slave is made.

The processes that comprise the process manager are user processes, but ones with special responsibilities. Like the shell, the master process manager sits high on a user's process tree and overlooks the evolution of this tree. The master process manager takes the place of today's shell in the process hierarchy. It is a child of the initialization process, *init(8)*, and the parent of the user's initial shell. If more shells are created, as is usual in the case of a window system, they would be descendants of the original shell and thus also of the process manager. The result is that the process manager is an ancestor of all processes originating from one user's login session. Figure 4.1 shows the role of the process manager in the process hierarchy.

The master-slave organization makes as much use as possible of the existing single-machine facilities by having the slave process managers use these facilities on behalf of the master. It also helps to reduce the amount of network traffic generated while performing process control duties by channeling short requests to the slave processes in a remote procedure call fashion.

A process manager's responsibilities include keeping track of a user's processes, taking part in remote signal delivery, and assisting in debugging sessions. These situations and the ways in which the process manager deals with them are explained below.

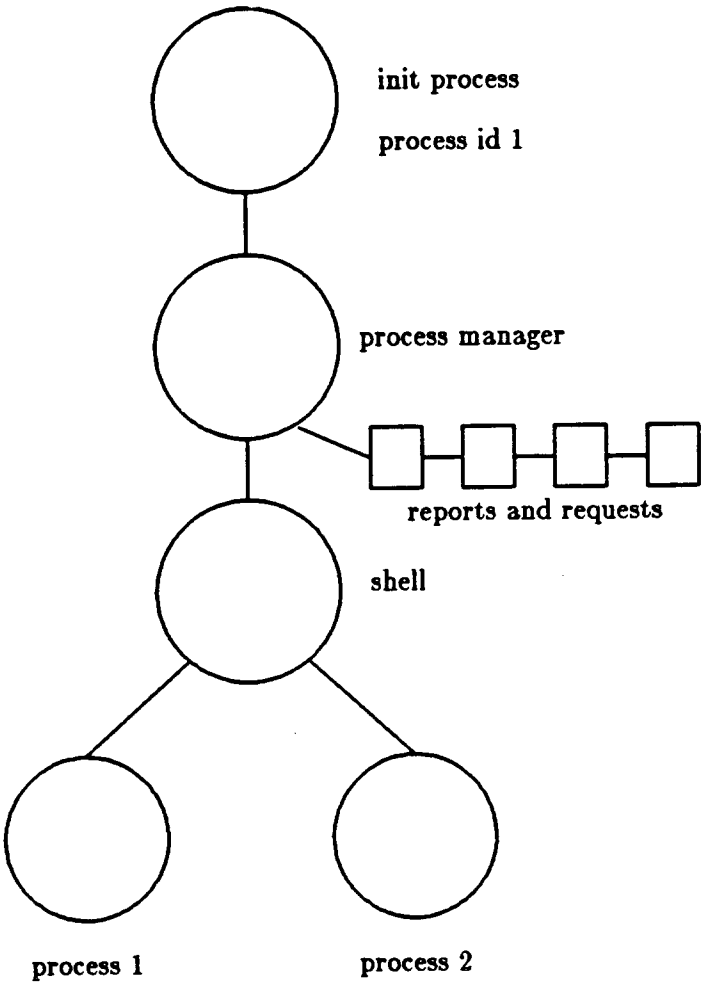

Figure 4.1 - The Process Manager

4.1. Keeping Track of a User's Processes

The major motivating factor for maintaining the state of a user's login session is the potential for user processes to be created on, and even migrated to, a different machine than where the login session started. As has been pointed out for the */proc* scheme, it would be too expensive in that case to dynamically search for a user's processes throughout the hosts on a network to perform everyday operations such as *ps()*.

In order to maintain itself informed of the progress of a user's computations, the process manager keeps a table with the process id and the host name of all of a user's processes. It also maintains information regarding the hierarchy of processes, i.e. parent-child relationships, in order to be able to reconstruct a user's process tree. This *user process table* is kept up to date by reports of status changes from user processes. Among the events that would be reported to the process manager are new process creations through *fork()*, and process terminations.

Here the tradeoffs between datagram and stream communication come into play. The reports in question are relatively sparse, and it would be wasteful of resources to set up a connection for that type of communication. Furthermore, there is again the problem of making the many pay for the few. Once the process manager is in place, all processes will rely on the status reporting mechanism for normal operation. Setting up a connection would burden all processes with a control socket and not just the ones being debugged or

using remote signals. This is exactly what the process manager approach is trying to avoid. On the other hand, using pure datagrams to send status reports to the process manager introduces unreliability into the operation, something that is unacceptable in this situation.

Fortunately, the Berkeley Unix IPC code guarantees reliable delivery of datagrams within a single machine. The process manager scheme uses this fact to its advantage by having processes report only to their local instantiation of the process manager, using datagrams with guaranteed reliability. If this is a slave process manager, it will forward the information to the master using the connection that exists between the two, which is again reliable. Therefore, the slave process managers are aware of the process status of processes within their own host, but it is only the master process manager that maintains a complete user process table.

4.2. Remote Signal Delivery Through Control Sockets

With the process manager approach, local signals are delivered through the usual channel, *kill()*. Only when a process desires to send a signal to a remote process are the new process control facilities used.

To send a remote signal, a process must request its local process manager to deliver the signal to the target process. This request is made through the same mechanism used to report status changes to the process manager, namely, reliably delivered intra-machine datagrams. If the local process manager is a slave, it must relay the request to the master process manager, who has permanent connections established with all slaves. The master process manager then has a choice. If the target process resides on its own machine, it can send the desired local signal through *kill()*. Otherwise, the master process manager must relay the request a final time to the slave on the target machine, who can then use *kill()*. Of course, if the process manager local to the process sending the signal is the master, it only takes one request to the appropriate slave process manager to do the job.

It is interesting to note that the levels of indirection present in the above transaction are a direct result of doing away with omnipresent control sockets on every process. If control sockets were always available, a remote signal would consist of a simple *sendto(2)* to the control socket of the target process. However, the disadvantages of having omnipresent control sockets described in Section 3.2 outweigh this gain in simplicity, especially since remote signals are expected to be a relatively rare occurrence. Furthermore, requiring that process control requests go through a central agent, the process manager, greatly simplifies the job of enforcing security. Not only is the system more secure, but the policies that dictate who is allowed to send control data to whom can be easily changed by simply changing the process manager code.

4.3. The Distributed Debugging Scenario

The process control mechanisms described here will find use as a basis for distributed debugging activities. The design of a multi-process distributed debugger for Berkeley Unix is in progress. The debugger can use the process manager and on-demand control sockets to send remote signals and perform ptrace-type operations on the processes being debugged.

First, in order to dynamically bind itself to the process or processes of interest, the debugger establishes communication with the master process manager for the user. It then proceeds to request debugging operations to be done on the process or processes it wishes to control. If only starting and stopping are desired, these can be carried out through the local and remote signaling mechanisms described earlier.

For accesses to the process's address space, the procedure is more complex. The debugger must place a special request for a ptrace-type operation with the master debugger, who then triggers the creation of the debuggee's control socket. The master debugger learns the name of the new control socket and relays that information to the debugger. The debugger can then directly connect to the control socket of the debuggee and proceed with the debugging session, using the IPC mechanisms for delivery of debugging commands and return data. Figure 4.2 shows the state of a single-machine debugging session once the debugger has contacted the process manager and subsequently established IPC connections with the processes of interest. Forcing the initial debugging request to go through the process manager allows for authentication of the requesting process. This prevents an arbitrary process to pose as a debugger and connect to the control socket of a process.

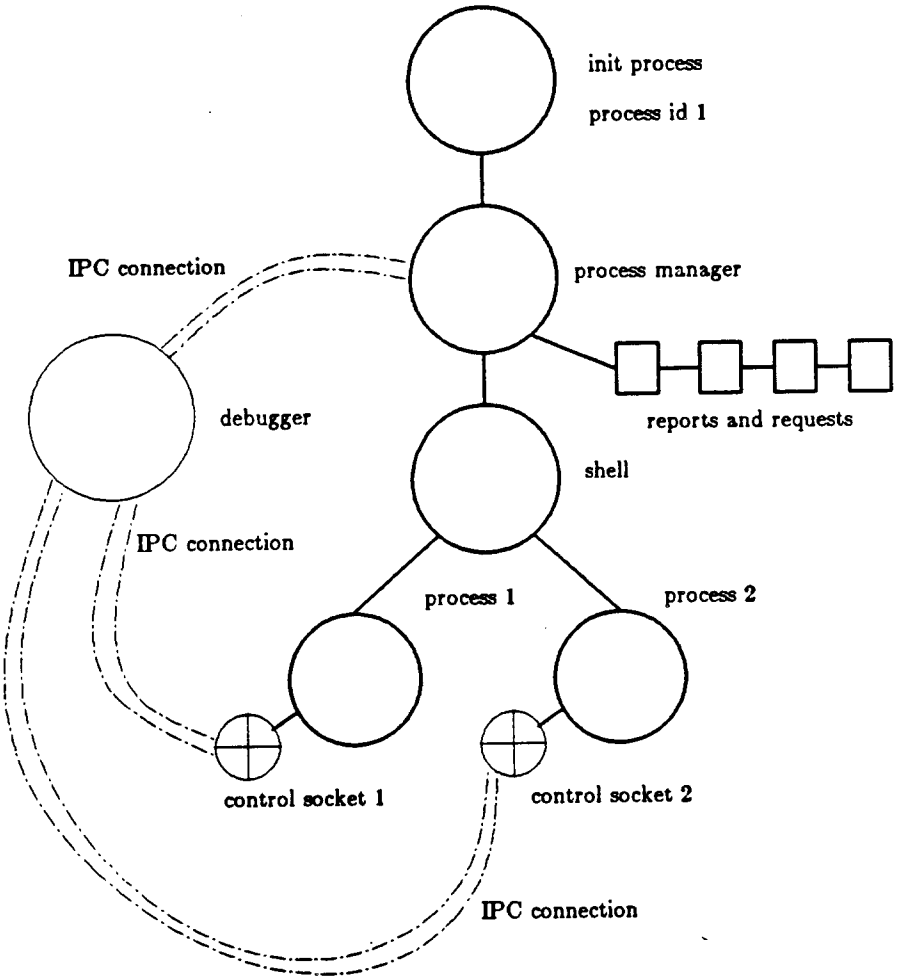

Figure 4.2 - Debugging Using the Process Manager

Again, the complexity of these transactions is a result of the absence of permanent and well-known control sockets. However, the cost of creating the control socket for a process and notifying the process manager of its address is only incurred once during the lifetime of a process, and then only if it is necessary to have a control socket present at all.

4.4. Implementation Details

Because of the dynamic nature of the transactions that involve on-demand control sockets, the implementation of this scheme is more complicated than that of omnipresent control sockets. However, as will be apparent below, less of a burden is placed on system activities and resources when the new mechanisms are not being used.

4.4.1. Kernel Changes

The first change to be made to the kernel is one that would permit processes to get a handle on the new process control facilities. In this case, processes must be able to efficiently communicate with their process managers. For this reason, the *u.area* for each process must be modified to include two fields: a pointer to the socket buffer of the local process manager's datagram socket, and the Internet address of the master process manager's stream socket. The datagram socket, as described earlier, is used by the process manager to receive status reports from all of a user's processes. The stream socket is used to establish a connection for longer-lived transactions, such as those encountered in debugging activities.

The next step is to set up the status reporting mechanism which the process manager uses to keep track of a user's processes. A report consists of a datagram delivered reliably within a single machine by appending a kernel *mbuf* to the message queue at the socket buffer of the local process manager. An *mbuf* is a fixed-size chunk of memory used within the kernel to hold and manipulate IPC headers and data. The ability of these buffers to be allocated and deallocated dynamically and to chain together to form queues make them very useful for many IPC applications.

The kernel must be modified so that a report is generated whenever an event takes place that causes an important change to the status of a process. For example, code must be added to *fork()* to notify the local process manager that a new process has been created. *exit()* must also be changed to send a report indicating that a certain process has terminated. It is not clear exactly how detailed a view of the world the process manager should have, so it may be necessary to add certain events to the list of actions that cause a report to be generated.

There must be a way for the process manager to trigger the creation of the control socket for a process. To this end, the new signal SIGDEBUG is introduced. When a process receives SIGDEBUG, it creates an Internet domain stream socket, binds an Internet address to it, and notifies the process manager of this Internet address. This way, the process manager can relay the address of the control socket to any interested parties such as a debugger.

4.4.2. Process Manager Code

The above descriptions of code changes refer to kernel code. However, much of the work in distributed process control will be done by the process manager itself, a user process. This is another advantage of the process manager approach over the omnipresent control socket scheme, because this makes the system easier to maintain and avoids undue growth of the kernel.

The process manager must first initialize the process control facilities by creating its own datagram and stream sockets, through which it will receive status reports and process control requests. It then forks a new process that will become the user's initial shell. Before the new process calls *exec()* to execute the shell, it initializes itself by filling in its *u.area* with handles to the recently created datagram and stream sockets of the process manager. This shell will spawn any succeeding user processes, and the *fork()* mechanism

will guarantee that the *u.area* of all these processes also contains the process manager socket information.

Once the initial shell is running, the master process manager goes into an infinite loop waiting for status reports or requests to come in through its sockets. In the case of reports, it will update its internal tables and return to reading the sockets. With requests, it must service the request before going back to listening to the sockets. Busy waiting is avoided by using the call *sigpause(2)*, which will pause a process until some specified signal is delivered, at which point the process will wake up. In this case the signal of interest if SIGIO, whose arrival signifies that data is pending on a socket.

It has been mentioned that slave process managers are created on remote machines as new user processes are formed or migrated there. The mechanism currently used for starting these slave process managers is *rexec()*. The limitations imposed on processes created through *rexec()*, described in Section 2.1, are not a problem for the special case of slave process managers. In particular, the fact that IPC connections are established from birth (one for input and output, and one for control and error data) is not a drawback here, because the slave process manager must maintain an open connection with its master in any case.

The process control facilities described here are not yet complete. The next section outlines some of the work that remains to be done before the process manager and the associated kernel code offer the desired functionality.

5. Future Research

The current naming scheme for Unix processes uses process id's that are unique only within one host, and carry no information as to where in the network a process resides. This has serious implications on the remote signal transactions, and indeed on the whole process control issue. Since current process id's are ineffective in a distributed context, process managers must implement their own naming strategy, separate from the one maintained by the rest of the Unix system. In the user process table, the process manager must associate a process id with the host name in which the process resides. This, in turn, forces requests to the process manager to include both host and process id information. The result is a messy solution that carries out virtually no information hiding, since anyone wishing to use the distributed process control facilities must be made aware of where processes reside.

This problem could be remedied by adopting a system-wide naming scheme that would guarantee the uniqueness of a process id throughout the network, a strategy that would also not break down when a process migrates. Such a scheme can be found in the LOCUS system[Popek81],[Walker83] and in Stanford's V-System[Cheriton84]. It is outside the scope of this work to select the process naming scheme for a distributed Berkeley Unix system. However, once one is adopted which satisfies the above requirements, the process manager code should be modified to use it. The interface to the distributed process control facilities will then become much more elegant.

In the area of remote process creation, an alternative to *rexec()* should be found. As described earlier, the processes formed through *rexec()* are not regular processes at all, since their standard input, output, and error are restricted to some predefined sockets. A better scheme for creating a remote process would be as follows: First, the local process manager is contacted with the request to create a process on a remote host. If there already exists a process manager on the machine where the remote process is to reside, the request is routed to that instance of the process manager. If one does not exist, a new slave process manager is created on the machine in question as described in the previous

section, and then the request is forwarded to it. The remote process manager then *forks*, and *execs* the requested process.

This way process managers will occasionally act as *process creation servers* on behalf of users. Processes created in this way will behave in exactly the same way as all other processes, with no restrictions. Furthermore, they will be children of an easily accessible process: the remote process manager on the machine where the process resides. This is in contrast to to processes created through *rexec()*, whose parent is the remote execution daemon, *rexecd()*. The proposed scenario is shown in Figure 5.1.

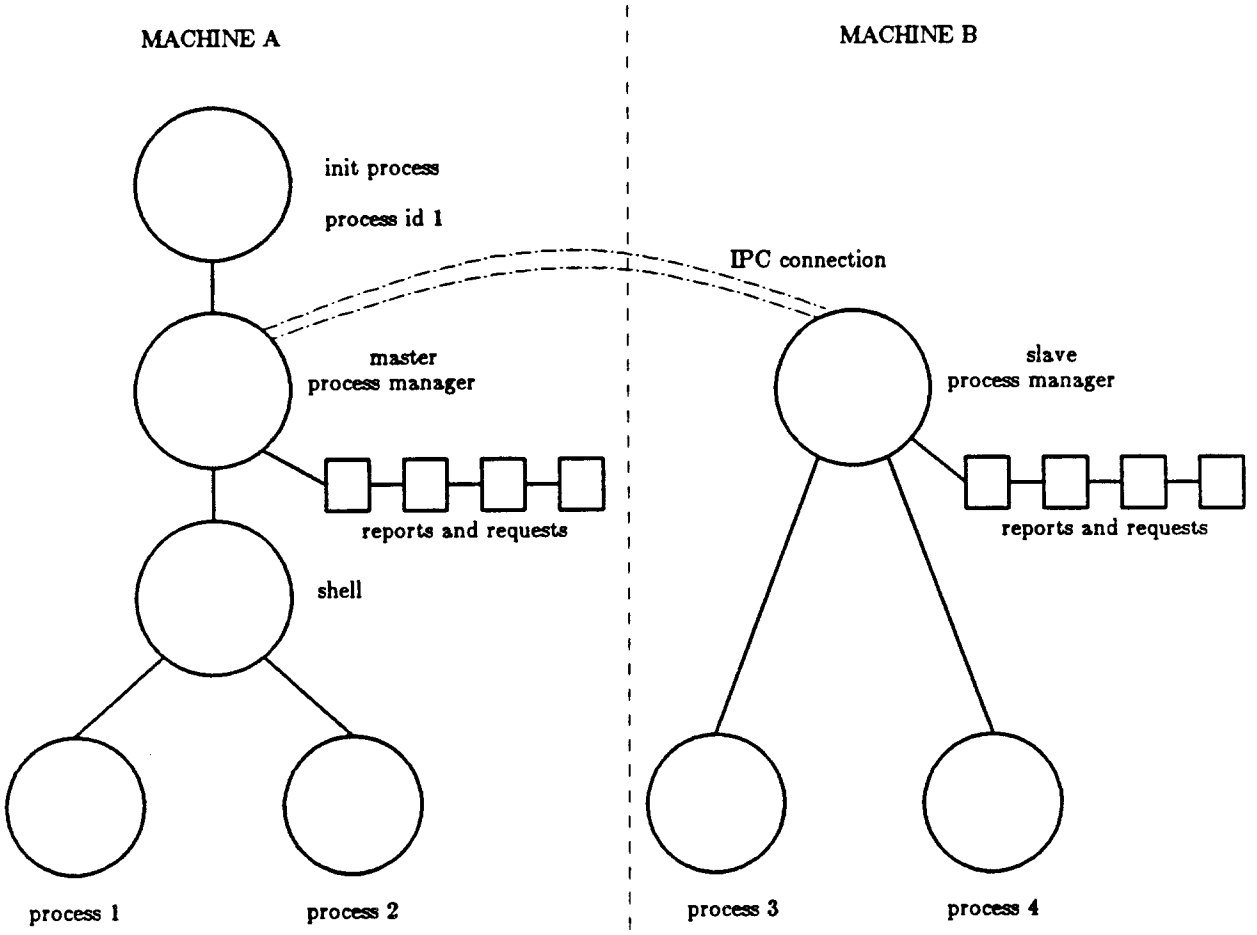

Figure 5.1 - The Distributed Processing Environment

The process manager scheme described here does not take into account a complication that is at present relatively commonplace: multiple login sessions by the same user throughout a network of machines. The process control facilities, and in particular the portion that keeps track of a user's processes, should be able to handle this case.

As things stand, process managers are associated with a single login session. If the same user logs in to a different machine, or more than once within the same host, the separate process managers will be unaware of each other. Furthermore, each process manager will have no information on the processes created as part of the other login sessions. Thus, operations such as *ps()* that rely on the status information kept by the process managers will not obtain a complete picture of the extent of a user's activities.

The advantage gained by using the data kept incrementally by the process managers, instead of repeatedly polling the entire system for information, will be lost if this data is incomplete.

As Berkeley Unix presents a more transparent view of its distributed resources, it is expected that multiple logins by the same user will become unnecessary, and thus rare. They will never completely disappear, however, and in some cases it will be necessary to be able to control all processes belonging to a particular user, regardless of which login session these processes are associated with. An example involves dealing with background or "stubborn" processes that remain from a previous login session. Here, a solution is for each newly created process manager to query the system as the login session is initiated for any orphaned processes with the same user id. The process manager can then inherit these processes from the system, in much the same way that the system inherited them when their original login session was terminated.

The case of simultaneous logins is more complicated, and reasonable approaches are not as obvious. Process managers can, as they are created, look through the system for other process managers belonging to the same user. From there, several strategies can be followed. One is for the new process manager to assume control and force any process manager it finds to become a slave. This is rather limiting and assumes that the logins have been originated by the same person, with related tasks to be performed by the separate login sessions.

Another approach would be for the new process manager to take note of any other process managers in existence, and to notify them of its presence. All process managers would then keep equal status, but they would be required to communicate in order for all of them to maintain a consistent view of the progress of a user's computations. Process managers could keep communication channels open in order to guarantee a fine grain of consistency. Establishing and maintaining this communication would be expensive in terms of network traffic and other resources such as sockets, but it could be feasible if the multiple login situation does not arise too often.

However, there is an alternative to constant communication between the process managers. If a complete view of the world by each process manager is not considered crucial, communication between process managers need only be sporadic. This communication can be limited to the initial interaction when a new manager is created, and to the case when an action to be performed by a process manager requires knowledge about processes belonging to other process managers. Such an instance takes place when a user requests a comprehensive *ps()*, one that must include all processes belonging to a user throughout the system.

It may seem at first glance that having a process manager poll the system for information incurs some of the same costs that the process manager scheme is trying to avoid. However, it is important to note that scanning the system is done only once in the lifetime of a process manager, and not every time data on remote processes is needed. After initialization, a process manager can go directly to the other process managers for data without having to broadcast for help.

Other than dealing with these three major issues, the process manager approach will have to evolve further as the notions of distributed process control are refined. New services may need to be added to the process manager code, with corresponding modifications to the Berkeley Unix kernel, if the process managers and control sockets are made to take on more responsibilities than those currently envisioned.

6. Conclusion

This report discusses the issues involved in building process control facilities for a distributed Berkeley Unix environment. The questions that arise when this problem is attacked are difficult ones, and by no means completely answered as yet. Several previously adopted solutions have been described, together with their known drawbacks. A new approach has also been presented, one that immediately addresses the shortcomings of the existing non-distributed process control mechanisms.

Of the existing solutions, the Bell Labs processes as files approach is very promising because of its elegant semantics and heavy Unix flavor. However, it has a serious shortcoming in that it does not address what is considered to be an important part of distributed process control: doing the bookkeeping necessary to keep track of the elements of a distributed computation.

The proposed solution uses control sockets and a per user process manager to create an environment where all the processes belonging to a user can be effectively controlled, even when such processes reside on more than one machine. It extends the Unix signal concept to a distributed environment and lifts the restrictions imposed by the *ptrace()* facility, including the single machine and single process requirements.

It would be useful to implement a processes as files scheme for Berkeley Unix and compare it with the process manager/control socket approach. If processes as files emerge as the clear winner in terms of performance and/or ease of use, it would be advantageous to retain the bookkeeping functions of the process manager, while substituting processes as files for control sockets to carry out signaling and active debugging functions.

In any case, the work performed on control sockets and process managers has proven very worthwhile. Not only do these ideas offer an interesting alternative to processes as files, but they also present a novel message-based approach to the problem of evolving the Unix programming environment to meet the growing demands of distributed applications.

Acknowledgements

I've had the pleasure of collaborating with Stuart Sechrest throughout the duration of this project. I want to thank him for all his patience and help, and especially for all those bright ideas. I would also like to extend my appreciation to Domenico Ferrari and Luis Felipe Cabrera for their guidance and time spent reading and commenting on drafts of this report. Finally, posthumous thanks to Bob Marley for his musical support.

References

[UserManual].
UNIX User's Manual, 4.2 Berkeley Software Distribution, Virtual VAX-11 Version, March 1984.

[ProgrammerManual].
UNIX Programmer's Manual, 4.2 Berkeley Software Distribution, Virtual VAX-11 Version, March 1984.

[Cheriton84].
D.R. Cheriton and T.O. Mann, "Uniform Access to Distributed Name Interpretation in the V-System," *Proceedings of the IEEE 4th International Conference on Distributed Systems*, pp. 290-297, San Francisco, May 1984.

[Daley68].
R.C. Daley and J.B. Dennis, "Virtual Memory, Processes, and Sharing in MULTICS," *Communications of the ACM*, pp. 306-312, May 1968.

[Dijkstra68].
E.W. Dijkstra, "The Structure of the THE Multiprogramming System," *Communications of the ACM*, pp. 341-346, May 1968.

[Leffler83a].
S.J. Leffler, W.N. Joy, and R.S. Fabry, "4.2BSD Networking Implementation Notes," Berkeley Technical Report UCB/CSD 83/146, July 1983.

[Leffler83b].
S.J. Leffler, R.S. Fabry, and W.N. Joy, "A 4.2BSD Interprocess Communication Primer," Berkeley Technical Report UCB/CSD 83/145, July 1983.

[Killian84].
T.J. Killian, "Processes as Files," *Proceedings of the Summer 1984 USENIX Conference*, Salt Lake City, Utah, June 1984 .

[Popek81].
G. Popek, B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudisin, and G. Theil, "LOCUS: A Network Transparent, High Reliability Distributed System," *Proc. of the Sixth Symposium on Operating Systems Principles*, pp. 169-177, Asilomar, Calif., December 1981.

[Postel79].
J. Postel, Editor, "User Datagram Protocol," RFC 768, Information Sciences Institute, Marina del Rey, California, August 1980.

[Postel81a].
J. Postel, Editor, "Internet Protocol," RFC 791, Information Sciences Institute, Marina del Rey, California, September 1981.

[Postel81b].
J. Postel, Editor, "Transmission Control Protocol," RFC 793, Information Sciences Institute, Marina del Rey, California, September 1981.

[Ritchie74].
D.M. Ritchie and K. Thompson, "The UNIX Time-Sharing System," *Communications of the ACM*, pp. 365-375, July 1974.

[Sechrest84].
Stuart Sechrest, "Tutorial Examples of Interprocess Communicatin in Berkeley Unix 4.2BSD," Berkeley Technical Report UCB/CSD 84/191, June 1984.

[Walker83].
B. Walker, G. Popek, R. English, C. Kline, and G. Theil, "The LOCUS Distributed Operating System," *Proc. of the Eighth Symposium on Operating Systems Principles*, pp. 49-70, Bretton Woods, N.H., October 1983.

[Wecker79].
S. Wecker, "Computer Network Architectures," *IEEE Computer*, pp. 58-72, September 1979.

[Weinberger84].
P.J. Weinberger, "The Version 8 Network File System (Abstract)," *Proceedings of the Summer 1984 USENIX Conference*, Salt Lake City, Utah, June 1984 .