The Process File System and Process Model in UNIX System V

Roger Faulkner – Sun Microsystems
Ron Gomes – AT&T Bell Laboratories

ABSTRACT

We describe the process file system **/proc** in UNIX System V Release 4 and its relationship to the UNIX process model abstraction. **/proc** began as a debugger interface superseding *ptrace*(2) but has evolved into a general interface to the process model. It provides detailed process information and control mechanisms that are independent of operating system implementation details and portable to a large class of real architectures. Control is thorough. Processes can be stopped and started on demand and can be instructed to stop on events of interest: specific machine faults, specific signals, and entry to or exit from specific system calls. Complete encapsulation of a process's execution environment is possible, as well as non-intrusive inspection. Breakpoint debugging is relieved from the ambiguities of signals. Security provisions are complete and non-destructive.

The addition of multi-threading to the process model motivates a proposal for a substantial change to the **/proc** interface that would replace the single-level flat structure with a hierarchy of directories containing status and control files. This restructuring would eliminate all *ioctl*(2) operations in favor of *read*(2) and *write*(2) operations, which generalize more easily to networks.

Introduction

The process file system represents all processes in the system as files in a directory conventionally named **/proc**. This concept was first introduced by Tom Killian in the research Eighth Edition UNIX system [1]. In System V Release 4 (SVR4) the concept has been refined from a simple replacement for the *ptrace*(2) system call into a general interface to the UNIX process model abstraction.

A typical "ls –l /proc" is shown in Figure 1. The name of each entry is a decimal number corresponding to the process id. The owner and group of the file are the process's real user-id and group-id, but permission to open the file is more restrictive than traditional file system permissions. The reported "size" is the total virtual memory size of the process; system processes such as process 0 and process 2 have no user-level address space, so their sizes are zero.

Standard system call interfaces are used to access **/proc** files: *open*(2), *close*(2), *lseek*(2), *read*(2), *write*(2), and *ioctl*(2). Data may be transferred from or to any valid locations in the process's address space by applying *lseek* to position the file at the virtual address of interest followed by *read* or *write*.

A process file contains data only at file offsets that match valid virtual addresses in the process. I/O operations with a file offset in an unmapped area fail. I/O operations that extend into unmapped areas do not fail but are truncated at the boundary. This includes writes as well as reads.

Information and control operations are provided through *ioctl*. A few of the *ioctl* operations are:

PIOCSTATUS	Get process status.
PIOCSTOP	Direct process to stop and ...
PIOCWSTOP	Wait for process to stop.
PIOCRUN	Make stopped process runnable.

```
-rw-------   1 root     root           0 Oct 31 10:06 00000
-rw-------   1 root     root      208896 Oct 31 10:06 00001
-rw-------   1 root     root           0 Oct 31 10:06 00002
                              ...
-rw-------   1 rrg      staff     131072 Oct 31 10:06 00206
-rw-------   1 weath    staff     749568 Oct 31 10:06 00370
-rw-------   1 raf      staff     651264 Oct 31 10:06 00393
```

Figure 1: A sample **/proc** directory

PIOCSTRACE	Define set of traced signals.
PIOCSFAULT	Define set of traced machine faults.
PIOCSENTRY	Define set of traced syscall entries.
PIOCSEXIT	Define set of traced syscall exits.
PIOCGREG	Get values of process registers.
PIOCSREG	Set values of process registers.
PIOCMAP	Get virtual address mappings.

This list is not exhaustive. Some of these operations are explained in more detail and additional ones are introduced in the following sections. Others are omitted entirely for brevity. The SVR4 *proc*(4) manual page provides complete details.

Process Address Space

SVR4 incorporates a new Virtual Memory (VM) architecture (derived from SunOS) that provides processes much greater control over the structure and content of address spaces [2, 3]. A process executes in a virtual address space consisting of a number of *memory mappings* (contiguous virtual address ranges). Associated with each mapping are a virtual address, a length, and a set of flags describing permissions (read, write, execute) and other attributes.[1] The traditional notions of text, data, and stack do not appear explicitly in this model but are subsumed by more general notions.

New system calls permit a process to map objects (generally files) into and out of its address space (*mmap*(2), *munmap*(2)) or to change the protections on a mapping (*mprotect*(2)). A mapping can be *private* (**MAP_PRIVATE**) or *shared* (**MAP_SHARED**). Modifications to a shared mapping are reflected through to the mapped object and appear in the address space of all other processes with a shared mapping to that object. Modifications to a private mapping affect only the address space of the process making the change and are invisible outside that address space.

The fact that a mapping is "private" does not mean that the implementation prohibits memory-sharing among processes that are mapping the same object. In fact, private mappings are implemented so as to provide *copy-on-write* semantics. Multiple private mappings to an object share the same memory pages until a process attempts to modify such a shared page, at which time the page is copied and the copy replaces the original in the address space.

[1]The granularity of a mapping is a system-specific page size, typically a small multiple of 1024 bytes. There is more to the VM architecture than is presented here. For example, individual pages can be mapped with different permissions and to different underlying objects.

Within this model a "text" segment is nothing more than a private executable mapping to the code portion of an executable file, *i.e.* an **a.out**. A "data" segment is a readable and writable private mapping to that portion of an **a.out** containing initialized data. A "stack" segment is a read/write mapping into which the stack pointer points, but is otherwise undistinguished (and in fact a sophisticated application can have multiple stacks). The system provides suitably-behaving anonymous objects to which mappings may be applied in the construction of other segments (*e.g.* "bss", uninitialized zero-filled memory). Shared libraries are implemented by mapping the code and data of a shared library executable file into the address space of a process.

The **PIOCMAP** operation extracts the memory map of a process. Figure 2 shows a typical memory map, obtained by a simple tool that reports the contents of the map structures returned by **PIOCMAP**. The list contains a number of writable mappings (presumably data) and a number of mappings that are read-only and executable (presumably code), from both the **a.out** itself and a shared library that has been mapped.[2]

```
80000000  26K read/exec
80008000   6K read/write/exec
80009800  74K read/write/exec/break
C0020000   4K read/write/exec/stack
C1000000 148K read/exec
C1026000   4K read/write/exec
C1027000   2K read/write/exec
C1028000   2K read/write
```

Figure 2: A Typical Memory Map

What may not be apparent from this list is that all the mappings are private (this is generally the case unless processes explicitly arrange to communicate with one another through a shared mapping). In particular the code portions are **MAP_PRIVATE** mappings with read and execute permissions. What happens if an attempt is made to store into a code portion? The process itself can't do this directly (reasonably so) because it doesn't have write permission on the mapping, but a controlling process can write the address space through the **/proc** interface. In this case the system will permit the write, and copy-on-write semantics will be provided where necessary. In this way breakpoints can be planted in

[2]Note that "stack" and "break" mappings appear in the list despite all the disclaimers. The operating system is prepared to grow one mapping (the initial program stack segment) automatically and another (the break segment) on explicit request by the *brk*(2) system call. A process-control application can sometimes make use of this information so it is provided in the **PIOCMAP** interface.

code, or data modified, without corrupting either the **a.out** file being executed or the address space of other processes that may be executing the same code.

Process Context

The execution context of a process (at least the portion deemed relevant) is described by the **prstatus_t** structure, which a controlling process can request at any time. Elements of this structure describe signal state, the contents of processor registers, process and session ids, and scheduling state (running or stopped, with more detailed information about stopped processes). The structure is returned by the **PIOCSTATUS** request or as an optional side-effect of the process-stop requests **PIOCSTOP** and **PIOCWSTOP**; it is designed to contain the information most frequently needed by a controlling process such as a debugger. Other data structures and operations exist for details of process state that are less frequently used, such as the contents of the floating-point registers, the information needed by *ps*(1), and the signal actions for every signal. Process state can be modified in controlled ways; for many of the operations that "get" state information there is a corresponding "set" operation. Thus, for example, the floating-point registers are fetched into a structure of type **fpregset_t** by the **PIOCGFPREG** request and are modified by the **PIOCSFPREG** request.

An important difference between this style of interface and that provided by the research prototype is the presentation of a complete and consistent *process model* as independent as possible of internal system implementation details. Formerly it was necessary to examine and directly manipulate the *user* and *proc* structures of the target process in order to effect state changes; this tied a process-control program to details that could (and did) change between releases of the system and was a functional improvement over *ptrace* only to the extent that it provided greater bandwidth and the ability to control unrelated processes. A primary goal was to remove these dependencies from the interface; secondary goals were to ease debugger development, improve portability of applications, and reduce the number of system calls routinely made by a debugger.[3] This has an associated cost in that there are more operations and data types for the programmer to master, but the cost is small in comparison with the resultant improvements in capability, consistency, portability, and efficiency.

[3]The goal of debugger efficiency, though irrelevant in many situations, becomes important in the implementation of features such as conditional breakpoints, for which "breakpoints per second" is a realistic measure of performance.

Events of Interest

A process executes in an environment established by and enforced by the UNIX kernel. Natural points of control for a process are where it enters and leaves the kernel, specifically, system call entry and exit, machine faults, and receipt of signals.

Events of interest are specified through the **/proc** interface using sets of flags. Signals are specified using the POSIX signal set type, **sigset_t**. Machine faults and system calls are specified using analogous set types **fltset_t** and **sysset_t**. Like signals, faults and system calls are enumerated from 1; there is no fault number 0 or system call number 0.[4] The SVR4 implementation provides for up to 128 signals, 128 faults and 512 system calls.

A traced process stops when it encounters an event of interest or when it is directed to stop, normally because the controlling process issued a **PIOCSTOP** request. It may also stop for reasons external to **/proc**; the competing mechanisms for stopping a process are *ptrace* and job-control stop signals.[5] (The **/proc** stop directive is independent of signals.) Ignoring the competing mechanisms, points in the kernel at which a process may stop are illustrated in Figure 3.

A stop on system call entry occurs before the system has fetched the system call arguments from the process. A stop on system call exit occurs after the system has stored all return values in the traced process's data and saved registers. This gives a debugger the opportunity to change the system call arguments before processing occurs and to manufacture whatever return values it wishes the process to see. In addition, a process that is stopped on system call entry can be directed to abort execution of the system call and go directly to system call exit. This combination of facilities enables complete encapsulation of the system call execution environment of a process so that, for example, older system calls or alternate versions of them can be simulated entirely at user level. (This is one way in which obsolete facilities could be supported "forever" without cluttering up the operating system.)

Stopping on machine faults and on system call entry and exit is straightforward; the process simply enters the kernel and stops. Stopping on receipt of a signal is more involved.

There are basically two points in the kernel where signals are detected: when the process is returning to user level and when the process is sleeping at an interruptible priority within a system call.

[4]System call number 0 exists in some UNIX system implementations as the "indirect" system call, but this only provides an alternate method for passing the real system call number.

[5]*ptrace* is made obsolete by **/proc** but is still required by the System V Interface Definition.

The kernel function *issig*( ) handles both cases.

Just before a process returns to user level, it checks for the presence of a signal to be acted upon and then acts on it by executing:

```
if ( issig() )
        psig();
```

If there are non-held and non-ignored signals pending for the process, *issig*( ) promotes one of them from pending to current and returns true. If the action for the signal is SIG_DFL, *psig*( ) terminates the process, possibly with a core dump. Otherwise, *psig*( ) modifies the saved registers and the user-level stack so that the process will enter the signal handler for the current signal when execution is resumed at user level. Job-control stop signals are treated differently; the default action for these signals is taken within *issig*( ).

Within an interruptible sleep, *issig*( ) is called to determine if the system call should be terminated with EINTR. If so, the process returns to *syscall*( ), perhaps stopping on syscall exit along the way, to ask the question again. Since there is already a current signal, another signal is not promoted by the second call to *issig*( ).[6]

issig( ) handles all cases of stopping the process due to receipt of a signal as well as the case of stopping the process due to the presence of a **/proc** stop directive. This includes stopping the process by the competing mechanisms. The complete logic of *issig*( ) is illustrated in Figure 4.

A process may stop twice due to receipt of a job-control stop signal, first on a signalled stop if the signal is being traced and again on a job-control stop if the process is set running without clearing the signal. A job-control stop is not an event of interest to **/proc**. Such a stopped process can be restarted only by sending it a SIGCONT signal. However the process can be directed to stop via **/proc** so that, when restarted by SIGCONT, it stops again on a requested stop before exiting *issig*( ). **/proc** gets the last word.

A similar situation holds for *ptrace*. When controlled via *ptrace*, a process stops on receipt of any signal, whether or not that signal is included in the set of signals traced via **/proc**. If the signal is traced via **/proc**, the process must be set running through **/proc** before it can be manipulated by *ptrace*. Even though the process is logically set running, it remains stopped on the signalled stop and cannot be set running again through **/proc**; *ptrace* has control. After *ptrace* sets the process running, it will stop again on requested stop before exiting *issig*( ) if it was directed to stop through **/proc**.

[6]Older UNIX systems did not use the current signal concept and consequently suffered a race condition in which the signal detected by *issig*( ) might not be the signal actually delivered to the process by *psig*( ). This caused a variety of problems, including a possible panic of the operating system if *psig*( ) attempted to deliver an ignored signal. For debuggers the consequence was that all signals except perhaps one had to be cleared on restarting a process after a stop, not just the signal that caused the stop.

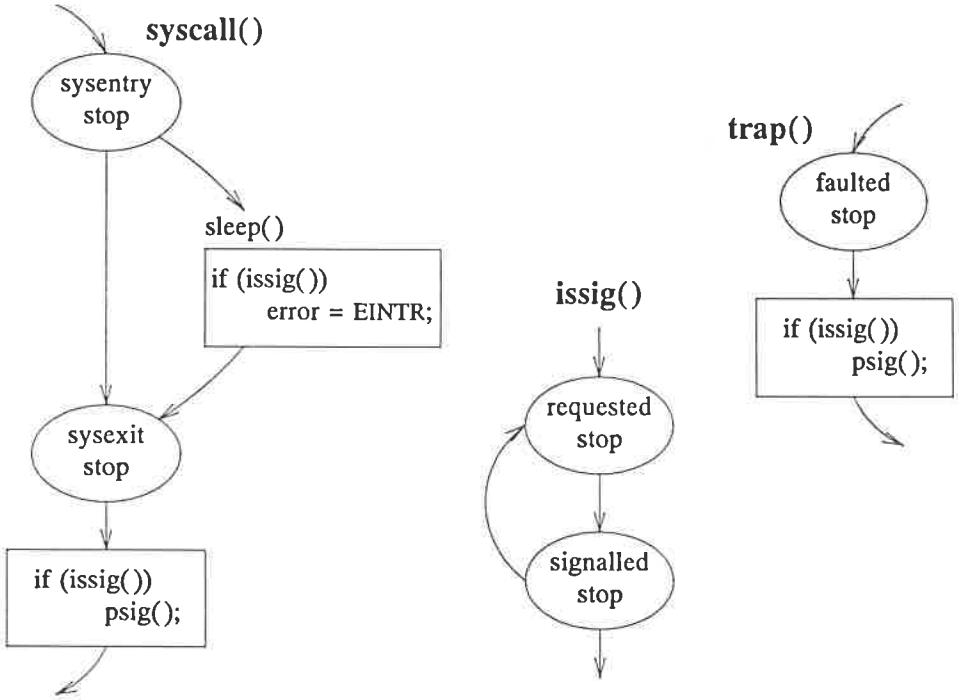


Figure 3: Events of Interest

/proc gets the first and last words.

Sending SIGCONT to a stopped process sets it running only if it is in a job-control stop; neither *ptrace* nor /proc can restart a job-control stopped process. All three mechanisms peacefully coexist by virtue of the delicate balance maintained in *issig*(), with cooperation from *setrun*().[7]

An important consequence of having all signal-related stopping confined to *issig*() is that a signal received while asleep in an interruptible system call need not cause premature exit from the system call when the process is set running again.[8] (The current signal can be cleared by the debugger. It is automatically cleared on a job-control stop; the SIGCONT signal that restarts the process will be discarded unless it is being caught.) Since the reason for sleeping may have gone away, *sleep*() must return normally to its caller when a stopped process is restarted without a current signal. Non-interruption of sleeping system calls relies on all callers of *sleep*() to test the reason for sleeping and to call again if the condition is still true, typically:

```
while ( condition )
        sleep(...);
```

This is a fine point and a fruitful source of kernel bugs.

Because a requested stop is performed in *issig*(), a process can be directed to stop while it is sleeping and set running again without disturbing the system call. The process can also be directed to abort the system call without having to send it a signal.

Breakpoints

The /proc interface does not directly implement the concept of a process breakpoint, but it provides sufficient mechanism for a debugger to do so. Breakpoints can be installed in a process by a debugger using the *read* and *write* operations on the process address space to replace the machine instruction at each breakpoint address with an illegal user-level instruction. Most systems designate one instruction as the approved "breakpoint" instruction,

[7]The *ptrace* and job-control stop mechanisms have always been in conflict. Job-control stops used to be disabled when a process was controlled by *ptrace*.

[8]UNIX systems used to arrange for signalled stops to occur within *psig*(), thereby forcing EINTR failures of interruptible system calls.

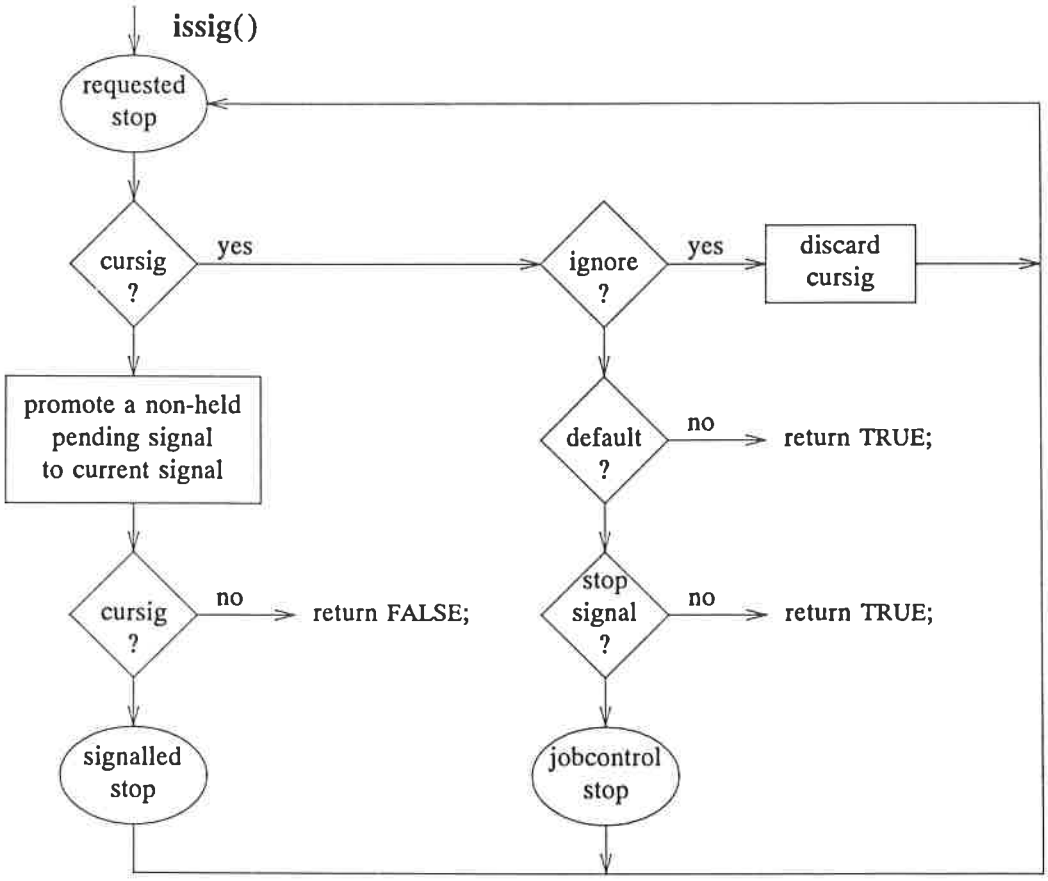


Figure 4: Process Control in issig()

but all that is really needed is one that causes a trap to the kernel. On architectures with variable-length instructions, the length of the breakpoint instruction should be that of the shortest instruction in the instruction set (to avoid overwriting the instruction following the breakpoint). The execution of the breakpoint instruction should leave the program counter with a known value relative to the breakpoint address in all cases, preferably the breakpoint address itself.

When the controlled process executes a breakpoint instruction, it takes a machine fault, FLTBPT if the instruction is the approved breakpoint instruction, otherwise FLTILL or FLTPRIV for a general illegal or privileged instruction. The process will stop on a faulted stop if the debugger has specified the particular fault as an event of interest. Otherwise the process is sent a signal, normally SIGTRAP or SIGILL. If the signal is not being held (blocked) by the process, the process will stop on a signalled stop if the debugger has specified receipt of the particular signal as an event of interest. The essential difference between stop-on-fault and stop-on-signal is the phrase, "if the signal is not being held."

A signal does not cause a process to stop when it is generated, only when it is received by the process. Also, any signal can be sent to a process by another process (subject to permissions). Lastly, there can be more than one signal pending for a process at one time. Signals are too overloaded in semantics and mechanism to be used reliably for breakpoint debugging. Machine faults are not used for inter-process communication and cannot be intercepted or held by a process; stop-on-fault is the preferred method for fielding breakpoints.

Controlling Multiple Processes

When a controlled process creates a child process, the controlling process may wish to add the new process to its set of controlled processes or it may wish to let the new process run unmolested. In either case some action must be taken.

To take control of new processes, a debugger can set the *inherit-on-fork* flag in the original process and arrange to trace exit from the *fork*(2) and *vfork*(2) system calls. When the controlled process *forks*, the child inherits all of the parent's tracing flags and both parent and child stop on exit from the *fork*. The debugger sees the parent's stop on exit from *fork* and uses the return value (the pid of the child) to open the child's **/proc** file. Because the child stopped before executing any user-level code, the debugger can maintain complete control.

To allow new processes to run unmolested, the debugger can simply reset the inherit-on-fork flag so that new processes start with all tracing flags cleared. However, if breakpoints have been set any new process will inherit them and possibly malfunction. In this case the debugger must arrange for the controlled process to stop on entry to as well as exit from *fork* and *vfork*. When the controlled process stops on entry to *fork*, the debugger lifts all the breakpoints and sets the process running. The child starts running with no tracing flags and no breakpoints. The parent stops on exit from *fork* and the debugger can replant all the breakpoints. Special care must be taken with *vfork* because the address space is shared between parent and child until the child *exit*s or *exec*s. **/proc** provides sufficient mechanism to deal with this case efficiently.

Miscellaneous

Tracing flags can remain active for a process when its process file is closed, allowing a process to be left hanging and later reattached by a debugger. This behavior is changed by setting the *run-on-last-close* flag. When this flag is set and the last writable **/proc** file descriptor for the process is closed, all of the tracing flags are cleared and, if the process is stopped, it is set running. This can be used by a controlling process to ensure that its controlled processes are released even if it itself is killed with SIGKILL.

Given a virtual address in the controlled process, the **PIOCOPENM** operation returns a read-only file descriptor for the underlying mapped object, if any. This enables a debugger to find executable file symbol tables, including those for shared libraries attached to the process, without having to know pathnames.

The **PIOCCRED** and **PIOCGROUPS** operations return complete credentials information for the controlled process.

Finally, the **PIOCGETPR** and **PIOCGETU** operations return, respectively, the *proc* structure and *user* area for the controlled process. These operations are provided for completeness but their use is deprecated because a program making use of them is tied to a particular version of the operating system. Their very existence reveals details of system implementation and their continuation into the new world of multi-threaded processes is doubtful.

A number of things that might be useful to know about a process are not provided through the **/proc** interface, such as its file creation mask. Our approach has been to provide information and control operations for the most common things that a debugger needs, and for things that a process cannot discover or do to itself through system calls. For the remainder, a debugger can force a process to execute system calls on the debugger's behalf without the process's knowledge or consent.

It is worth noting that the SVR4 implementation of **/proc** works correctly with Remote File Sharing (RFS) [4]. With appropriate permission it is possible to inspect, modify and control processes running on

any machine in an RFS network. This extension of capability "for free" to any machine in the network applies to any resource that is accessible within the file system name space and is an additional justification for implementing resources this way.[9]

Integrity and Security

The interface distinguishes operations that modify process state or behavior (such as a request to write the registers) from those that merely inspect process state (such as a request for process status). The former are regarded as "read/write" operations and the latter as "read-only." A **/proc** file can be opened for exclusive read/write use (if O_EXCL is specified in the *open*(2)); in this way a controlling process can avoid collisions with other controlling processes. Read-only opens are unaffected in this case.

All I/O and control operations are guaranteed to be atomic with respect to the traced process. Copy-on-write is performed by the system excepting only bona-fide shared memory; writing to one process will not corrupt another process executing the same executable file or shared library. This applies in general to **MAP_PRIVATE** VM mappings.

Permission to open a **/proc** file requires that both the uid and gid of the traced process match those of the controlling process; setuid and setgid processes can be opened only by the super-user. When a traced process *exec*s a setuid or setgid executable file, the set-id operation is honored but the file descriptor held by the controlling process becomes invalid; no further operation on that file descriptor will succeed except *close*(2), thus enforcing security without modifying process behavior.[10] When the set-id *exec* occurs, the traced process is directed to stop and its run-on-last-close flag is set. A controlling process with appropriate privilege can reopen the named **/proc** file to retain control of the target; just closing the invalid file descriptor clears all tracing flags and sets the set-id process running.

Implementation

The implementation of **/proc** as a set of "files" is facilitated by the Virtual File System (VFS) architecture of SVR4 which is derived from the *vnode* feature [5] of SunOS and subsumes the File System Switch (FSS) of earlier releases of System V. VFS permits the coexistence on a single system of several disparate *file system types* (fstypes) by providing a clean separation of file system code into *generic* (file system-independent) and *specific* (file system-dependent) pieces with a well-defined but narrow interface between the pieces. (Generic code is viewed as "upper-level" and specific code as "lower-level.") Typically the set of fstypes on a system will include conventional disk file systems and network file systems as well as more outlandish things such as **/proc**. In general any resource can be made to appear within the file system name space if it makes sense to view it that way.

The fundamental data structure manipulated by generic code is the *vnode* (virtual node), which is the system's internal representation of a file and provides the handle by which file manipulations are performed. A vnode contains both public and private data. The public data in a vnode consists of information that is maintained by the upper level or that does not change over the life of the file (such as the file type); private data is opaque to the upper level and is implementation-specific (such as a list of block addresses for a disk file).

The upper level requests the creation of vnodes by the lower level, and these vnodes are subsequently supplied as operands to other file operations. The set of vnode operations includes *open*, *close*, *read*, *write*, *ioctl*, *lookup*, *create*, *remove*, and many more. The developer of a file system type provides the code that implements the necessary set of vnode operations for that type.

Within this framework the construction of the fantasy world (the illusion that processes are actually files) is straightforward. System call references to **/proc** files result in the invocation of lower-level code to create and maintain **/proc** vnodes. For example, an attempt to open **/proc/2846** results in a call to *prlookup* which searches the system process structures for process id 2846 and (if such a process exists) constructs a vnode for it. The upper-level code associates this vnode with the open file descriptor, and subsequent applications of *read*(2), *write*(2), and *ioctl*(2) result in calls to *prread*, *prwrite*, and *prioctl* to perform the requested process I/O or control operation. Similarly, a command like *ls*(1) that wants to read the contents of the **/proc** directory will apply *readdir*(3) to it; this results in a call to *prreaddir* which examines the system process structures and satisfies the system call by constructing a set of directory entries naming all the processes in the system.

The intimate connection with process control requires some code in addition to the usual VFS plumbing; in this respect **/proc** is an unconventional file system and not an "add-on." Most of this code deals with the interaction between signals and process stopping and appears in *issig*() (discussed above in more detail). Minor changes were made in a few other places including the system-call handler (to stop the process on system call entry or exit), the user trap handler (to stop the process when it incurs a machine fault), the scheduler (to suspend a process

[9] Needless to say, a debugger that takes advantage of this facility must be prepared to deal with all of the problems inherent in heterogeneous networks.

[10] This differs from the more intrusive behavior with *ptrace*, in which set-id flags are ignored if the target performs an *exec*(2).

undergoing /proc I/O), *exec*(2) (to invalidate /proc file descriptors to set-id programs), and *exit*(2) (to inform /proc of the death of a process).

Implementation of /proc I/O requires that one process be granted access to the address space of another. This was a troublesome problem in the original research prototype because the memory management code of the underlying system made it difficult for one process to incur a page fault on behalf of another. The new Virtual Memory architecture simplifies this problem. VM provides a model of memory management in which machine-dependent details are isolated in a separate layer.

In particular, each process has an associated *address space* (''as'') data structure to which a set of standard operations may be applied. One such operation is **as_fault**, which performs page-fault processing for a specified range of addresses. Given this operation, all that is necessary for inter-process I/O is for the controlling process to apply **as_fault** to the address space of the target process, map the target pages into its own address space, and copy the data between the two addresses.

Overall, a high degree of portability was achieved in the implementation of /proc. The VM abstraction hides many of the details of memory management. Machine-specific /proc VFS code is confined to a single source file containing less than 10% of the total /proc-related code. Assuming a complete implementation of the VM primitives and of the generic porting base, porting should require only the specification of a few details such as the code for fetching register contents. (There is a presumption here that the process model accommodates all ''interesting'' machines.)

Applications

The SVR4 *ps*(1) command is implemented using /proc. Special provision was made for it in the interface; the **PIOCPSINFO** operation returns everything that *ps* might want to display about a process. The logic of *ps* is to read the /proc directory, open each process file in turn, issue the **PIOCPSINFO** request, close the file, and print the result if appropriate according to the *ps* options. Because *ps* runs with super-user privilege and the process files are opened read-only, the *open*s always succeed and no interference is created for controlling and controlled processes. Because all the information for a process is obtained in a single operation, each line of *ps* output is a true snapshot of the process, even though the complete listing is not a true snapshot of the whole system.

The interception of system calls with /proc is at the heart of *truss*(1), a command that traces the execution of a process, producing a symbolic report of the system calls it executes, the faults it encounters and the signals it receives. *truss* can be applied to running processes or used to start up commands to be traced, and will optionally follow the execution of child processes as well. Because it requires no symbol information and is applicable at any time to an arbitrary process (even *init*), it can often be used to find out what a misbehaving program is *really* doing even if source is unavailable and the executable file symbol information has been stripped. *truss* output can be startling.

truss is constrained by the security provisions of /proc, so that it can be applied only to ordinary (non-set-id) processes owned by the user. Moreover if the traced process *exec*s a set-id program *truss* loses control; the process continues normally, with correct credentials, but no longer under control of *truss*. If *truss* is run by the super-user, all permissions are granted and any process and all its children can be traced. *truss* will not alter the behavior of a process other than by slowing it down. (Of course, just slowing it down can affect behavior if the process uses *alarm*(2) or other real-time mechanisms.)

The interface is clearly intended to facilitate a sophisticated debugger and has already supported the development of several prototypes. Such a debugger is planned for a future release of the system. In the meantime the use of *ptrace* is being phased out; the standard debuggers *sdb*(1) and *dbx*(1) have been rewritten in SVR4 to use /proc (and, for *sdb*, to add a few new capabilities, such as the ability to grab and debug an existing process).

Proposed Extensions

A number of new facilities have been proposed for inclusion in future releases of the system. We describe a few of them here (though note that there is no promise that any of these will actually be provided anytime soon).

By appropriately defining what it means for a /proc file to be ''ready'' it would be possible to permit /proc file descriptors to be used with the *poll*(2) system call. This would make it much easier for a debugger to wait for any one of a set of controlled processes to stop on an event of interest while also waiting for events such as keyboard input from the user. It would offer more flexibility for multi-process debugger implementations than the current method of waiting for only a single process to stop; this flexibility will be even more important when there can be multiple threads of control within a single process.

/proc currently gives short shrift to performance aspects of the process model. A resource usage interface has been proposed, along with an interface to a process's page data whereby a performance monitor can sample page-level referenced and modified information for a process on intervals at will.

A generalized data watchpoint facility has been proposed and designed, based on the VM system's ability to re-map read/write permissions on individual pages of a process's address space. It can be implemented on any architecture capable of running SVR4 and can take advantage of specialized hardware when available. The interface accepts specification of watched areas of any size, down to a single byte. The traced process stops only when a watchpoint really fires; the system takes care of the details of recovering
from machine faults taken due to references to unwatched data that happens to fall in the same page as watched data.

It is possible, with the addition of a small but ugly wart on the **/proc** interface, to eliminate *ptrace* from the operating system and implement it as a library function built on **/proc**. The difficult part is not with *ptrace* itself, but rather with the requirement that a process stop via *ptrace* be reported to the parent via *wait*(2).

The current implementation does not permit a debugger to directly map the address space of the traced process via *mmap*(2); access is possible only through explicit *read* or *write* system calls. Permitting *mmap* would provide no new capability *per se* but would allow very high-speed inspection or modification of the target's address space. Such a facility is under consideration.

Proposed Restructuring

The evolution of the operating system toward a process model incorporating shared address spaces and multiple threads of control places some strain upon the interface in its current form. A new structure is under consideration that would change the **/proc** file system from a flat structure to a hierarchical one containing a number of sub-directories and additional status and control files. The programming interface changes from one in which *ioctl*(2) operations are applied to open file descriptors in order to effect process control and interrogate process state to one in which process state is interrogated by *read*(2) operations applied to appropriate read-only status files and process control is effected by structured messages written to write-only control files. (A structure similar in concept but different in detail appears in Plan 9 [6].)

The change in model has a number of advantages independent of multi-threading considerations. Removing the dependence on *ioctl* simplifies the implementation of **/proc** in a network environment. The unstructured nature of *ioctl* operations and the variability of operand sizes and I/O directions make it difficult to cleanly separate the client/server interactions; *read* and *write* don't share these problems. In addition the use of a control file to which structured messages are written makes it possible to combine several control operations in a single *write* system call; this can improve the performance of some applications for which the number of system calls is a bottleneck.

Of more relevance for the process model is that a directory hierarchy is a natural structure in which to present the relationship between a process and the individual threads-of-control that share its address space. Thread-ids of sibling threads appear as subdirectories within a hierarchy that has the process-id at the top.

Outstanding Issues and Future Work

/proc completes a long-incomplete process-model/debugger interface. Unfortunately, a process model interface built into the kernel can of necessity deal only with kernel interfaces. The Application Binary Interface (ABI) was introduced in SVR4. The ABI is not a kernel interface but a user-level shared library interface, with the shared library being provided by the purveyor of the system.

With the advent of the ABI, programming interfaces move from the kernel level to the shared library level. This is especially true for multi-threaded applications in an environment in which user-level threads may be multiplexed onto a smaller set of kernel threads. A debugger that deals with the user-level threads model must have access points in the threads library of the same power as the system call interfaces that **/proc** provides for kernel-level threads. A generalized shared library interface control mechanism would benefit debugging of applications in general.

As always, debugging lags development.

Acknowledgements

Dave Weatherford has been a serious user of **/proc** and has given us consistently good advice over the years.

Jonathan Shapiro conceived of and did most of the design for the proposed generalized watchpoint facility.

Much useful advice was given to us by other developers both at Bell Labs and at Sun Microsystems during the development of SVR4. In particular we thank Bill Shannon.

We especially wish to thank the following people who gave us their support in bad times as well as good: Phyllis Eve Bregman, Sheila Brown-Klinger, Blaine Garst, Barbara Moo, and Marilyn Partel.

References

[1] T. J. Killian, *Processes as Files*, Proceedings of the USENIX Association Summer Conference, Salt Lake City, June 1984, pp. 203-207.

[2] R. A. Gingell, J. P. Moran, W. A. Shannon, *Virtual Memory Architecture in SunOS*,

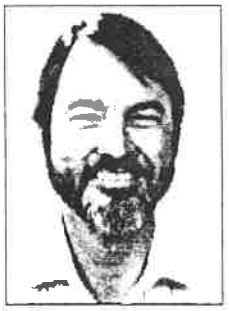
Proceedings of the USENIX Association Summer Conference, Phoenix, June 1987, pp. 81-94.

[3] Joseph P. Moran, *SunOS Virtual Memory Implementation*, Proceedings of the European UNIX User's Group, London, UK, April 1988, pp. 285-300.

[4] A. P. Rifkin, M. P. Forbes, R. L. Hamilton, M. Sabrio, S. Shah, K. Yueh, *Remote File Sharing Architectural Overview*, Proceedings of the USENIX Association Summer Conference, Atlanta, June 1986, pp. 248-259.

[5] S. R. Kleiman, *Vnodes: An Architecture for Multiple File System Types in Sun UNIX*, Proceedings of the USENIX Association Summer Conference, Atlanta, June 1986, pp. 238-247.

[6] R. Pike, D. Presotto, K. Thompson. H. Trickey, *Plan 9 from Bell Labs*, Proceedings of the UKUUG Conference, London, UK, July 1990, pp. 1-9.



Roger Faulkner grew up in North Carolina. He received a B.Sc. in Physics from N. C. State University in 1963 and a Ph.D. in Physics from Princeton University in 1967, then joined Bell Laboratories. He was seduced by computers while using them to solve physics problems and joined the Murray Hill computer center in 1970. During 1970-1975 he learned about the UNIX system by osmosis from denizens of the attic. During 1976-1980 he was actively involved in the inner workings of the UNIX kernel, after which he left Bell Labs to live and work in New York City. He returned to UNIX work in 1984 with a mission from God to create the one true debugger. He failed in this but succeeded in making Ron Gomes create the one true debugger interface. He hopes that others will use it for worthy purposes. His current work involves system support for debugging of multi-thread, multi-process, multi-machine, and multi-national applications.

Ron Gomes is a native Canadian, about which no more need be said. He received a B.Sc. in Mathematics from McGill University (Montreal) in 1975 and an M.Sc. in Computer Science from the University of Toronto in 1977. He has been working on and with UNIX systems since 1975 and knows what *dsw* means. Previous employers have included the University of Toronto, Bell-Northern Research, and HCR Corporation, for whom he worked on projects including operating system support and enhancement, kernel ports, and compiler development. Since 1984 he has been with the System V development organization of AT&T Bell Laboratories, now UNIX System Laboratories, Inc., where he contributed to the design and development of System V Releases 3 and 4, notably in the area of file system architecture. His current work involves multi-processor systems and in particular the further development of the process model.