USENIX Association

Software Tools Users Group

Summer Conference

Salt Lake City 1984

PROCEEDINGS

June 12–15, 1984

Salt Lake City, Utah, USA

Processes as Files

T. J. Killian

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

We describe a new file system, **/proc**, each member of which, **/proc/***nnnnn*, corresponds to the address space of the running process whose pid is *nnnnn*. Access to these files is restricted, via the normal file protection mechanism, to the process owner. *Lseek*(2), *read*(2), and *write*(2), allow inspection and modification of the process' image. Other services are available via *ioctl*(2), including stop/go on demand, selective intercepting of signals, and the ability to obtain an open file descriptor for the process' text file. The technical problems related to the implementation of **/proc** on a VAX† under the 8th Edition of the Unix‡ operating system have mostly to do with the paging system. Security issues are also considered.

The window-based interactive debugger *pi*, developed by T. A. Cargill, is the first major user of **/proc**. It can control multiple processes dynamically and asynchronously. We describe it briefly, and discuss its system interface.

Introduction

Any debugger is dependent on, and often limited by, its ability to access the address space of the debugged program. This is especially true in the case of interactive debugging under the Unix system, where the debugger and the debugged object are separate processes. The problems associated with the standard mechanism, *ptrace*(2), are well known:

- The object must agree explicitly to be debugged, and furthermore it can only be debugged by its immediate parent. Thus there is no dynamic binding, and children of the original object process cannot be handled.
- Before it can be examined, the object must be put in a stopped state, typically by sending it a *signal*(2). This can interrupt the object's own system calls, so the debugging is not transparent. Or the object may be ignoring signals (e.g., sleeping forever on a locked inode), so the mechanism can fail entirely.
- *Ptrace*(2) provides low bandwidth at high cost: two context switches per word of data transferred, an achievement equaled only by some text editors. Its protocol is arcane and unnatural compared to most other system calls.

We have tried to overcome these difficulties by providing an interface that is as uniform as possible, using an existing mechanism for accessing random data external to a process: the file system.

† VAX is a trademark of Digital Equipment Corporation.
‡ Unix is a trademark of AT&T Bell Laboratories.

Fig. 1: A sample /**proc** directory

```
-rw-------  1  root      14336 Feb 20  12:59 00001
-rw-------  1  root     528384 Feb 20  12:59 00002
-rw-------  1  root      12288 Feb 20  12:59 00019
                          .
                          .
                          .
-rw-------  1  tom       32768 Feb 20  12:59 02596
-rw-------  1  tac      106496 Feb 20  12:59 02652
-rw-------  1  root      14336 Feb 20  12:59 02801
-rw-------  1  tac       39936 Feb 20  12:59 02900
-rw-------  1  tac       23552 Feb 20  12:59 02910
-rw-------  1  tac      184320 Feb 20  12:59 02911
-rw-------  1  tom       33792 Feb 20  12:59 02912
-rw-------  1  tom       54272 Feb 20  12:59 02913
```

System-Call Interface

Fig. 1 shows the result of a typical "ls −l /proc." The name of each entry in the directory is a five-digit decimal number corresponding to the process id. The owner of the "file" is the same as the process' user-id; note that only the owner is granted permissions. The size is the total virtual memory size of the process. The time is not very useful: it is always the current time, for reasons discussed later.

The standard system-call interface is used to access /**proc**. *Open*(2) and *close*(2) behave as usual with no side-effects. In particular, the object process is not aware that it has been opened. Data may be transferred from or to any locations in the object's address space through *lseek*(2), *read*(2), and *write*(2). Reading and writing have slightly peculiar behavior due to the segmenting of the process' address space.

The *text segment* (see Fig. 2) will be read-only, if it was already shared at the time of the attempted write; otherwise, the text is marked impure, and subsequent writes are guaranteed to succeed. The *data* and *stack segments* are read/write in any case. The *user area* is read-only, except for locations corresponding to saved user registers. The system segment is not accessible. For simplicity in enforcing these restrictions, a single I/O operation may not cross a segment boundary; the byte count will be truncated if necessary. Note that for ordinary files, the entire file is either write-protected or not, and "holes" read as zeroes.

As with other special files, there are a number of services available via *ioctl*(2), in this case having to do with process control:

`PIOCGETPR` fetches the object's `struct proc` from the kernel process table. Since this information resides in system space, it is not accessible via a normal read.

`PIOCSTOP` sends the signal `SIGSTOP` to the object, and waits for it to enter the stopped state.

`PIOCWSTOP` simply waits for the object to stop.

`PIOCRUN` makes the object runnable again after a stop.

`PIOCSMASK` specifies (via a bit mask) a set of signals to be "traced"; i.e., the arrival of such a signal will cause the object to stop. A mask of zeroes turns off the trace. There are two side-effects: (1) a "traced" process will always stop after exec'ing; and (2) the "traced" state is retained after the object is closed, although the mask bits themselves are lost.

`PIOCCSIG` clears all of the object's pending signals.

`PIOCEXCLU` marks the object's text segment as impure, so that subsequent writes will succeed. This ioctl fails if the text is already shared.

Fig. 2: Address structure of **/proc**/*nnnnn*

```
           |                        |
0x80000000 |     System segment     |
           |                        |
-----------+------------------------+
           |                        |
0x7ffff000 |       User area        |
           |                        |
-----------+------------------------+
           |                        |
    (*)    |     Stack segment      |
           |                        |
-----------+------------------------+
           |                        |
           |     [Non-existent]     |
    (*)    |                        |
-----------+------------------------+
           |                        |
    (*)    |      Data segment      |
           |                        |
-----------+------------------------+
           |                        |
0x00000000 |      Text segment      |
           |                        |
-----------+------------------------+

(*) addresses computed from segment sizes
```

`PIOCOPENT` provides, in the return value of the ioctl, a read-only file descriptor for the object process' text file. This allows a debugger to find the symbol table without having to know any path names.

All system calls are interruptible by signals, so that, for example, an *alarm*(2) may be set to avoid waiting forever for a process that may never stop. Any system call is guaranteed to be atomic with respect to the object, but, as with ordinary files, there is nothing to prevent more than one process from trying to control the same object.

Implementation

The interface to the file system was provided by P. J. Weinberger, who introduced the notion of a *file system type* to support his network file system [1]. In a sense, this is a generalization of the top level of the *pipe*(2) mechanism, in which, e.g., the kernel's read routine calls a special procedure if the file descriptor refers to a pipe. In Weinberger's scheme, the kernel has a well-defined set of internal entry points (*read, write, open, close,* etc.) for each file system type, and uses the appropriate one transparently. A special *mount*(2) command is used to associate a particular file system type with a given leaf of the directory hierarchy. Weinberger in fact made an early attempt at an implementation of **/proc**, but the communication mechanism was too similar to that used by *ptrace*, making it impractical.

In our implementation, the calling process accesses the object directly. This requires the cooperation of the swapper, the scheduler, and the paging system. A flag in the object's `struct proc` informs the system globally that the object is undergoing I/O via **/proc**. The swapper recognizes the object as a candidate for being swapped in, and will not swap it out as long as the flag is set. The scheduler will not run the object; in particular, the effect of any signal or wakeup on the object will be delayed until I/O is completed. Finally, hooks have been added to the paging system so that the object's pages may be brought in on demand by the calling process. Thus I/O via **/proc** can take place under almost any circumstances. The exceptions occur where the object is waiting for an actual virtual-memory event, i.e., it is exec'ing, paging, forking, or exiting. In these cases, the I/O call returns an error, and the caller must try again.

Because the caller and object must both be swapped in during I/O, there is the possibility of deadlock on very small systems. In practice, this is unlikely to happen under 8th Edition Unix because being swapped in implies only that the user area and page tables are present; the remainder of the process can page in and out as necessary. In any case, I/O via **/proc** is always interruptible, even in an otherwise deadlocked situation.

Reading the **/proc** directory and *stat*'ing(2) its members present special cases. The information returned is made up exclusively from the kernel's process table. Some potentially useful data, like process times which reside in the user area, are not returned for efficiency reasons, since a swap might be required to get access.

Security

The most obvious security loopholes are plugged by the file system itself. The standard protection mechanism prohibits free-for-all access to every process, and the file system type itself does not support things like *mv*(1), *rm*(1), *chmod*(1), etc. Some more subtle problems are the following: **/proc** provides a way of reading a file which has execute, but not read, permission; this is easily solved, since the inode of the object process' text file is available at the time the object is stat'ed or opened, and permissions can be checked. If a process opened by **/proc** exec's a program with setuid bits, there is the potential for any user to become the super-user; we take care of this by not honoring the setuid bits in such a case. Note that if the open is attempted after the exec, the process owner will have already changed and the normal protection applies.

Finally there is the problem of corrupting shared text, which has already been discussed. This problem is actually solved over-zealously, as a convenience: if an opened process exec's, it is automatically given impure text, on the assumption that a debugger is waiting in the wings.

The *pi* system interface

T. A. Cargill has developed an interactive, window-based debugger called *pi* (process inspector). It can be bound dynamically to multiple processes, and control them asynchronously. Binding is particularly simple: *pi* opens the object process, gets hold of the text file via `PIOCOPENT`, and reads the symbol table. It is frequently useful to have control of the object before it has actually begun executing. This is easily done, given the semantics of `PIOCSMASK`. Fig. 3 shows the program *hang*, which takes any command as argument, and starts it up in the stopped state.

Pi obtains status information and data from a process using ioctl's, or a combination of seeks and reads. Other operations require a more complicated series of primitives. *Single-stepping* is accomplished as follows (the object is assumed to be already stopped):

1) Arrange, via `PIOCSMASK`, for the object to stop on `SIGTRAP`.
2) Set the `TRACE` bit in the object's `PSL`. This involves a seek, read, and write in the object's user area.
3) Issue `PIOCRUN`.
4) Wait for the object to return to the stop state, by issuing `PIOCWSTOP`.
5) Issue `PIOCCSIG` to clear the `SIGTRAP`. (If the object has other signals pending which should be preserved, they can be read via `PIOCGETPR`, and resent.)

Single-stepping over a function call (`CALLS` instruction) is only slightly harder. If the instruction stepped was a call, one sets the `TRACE` bit in the `PSL` saved on the stack and issues `PIOCRUN`; the `SIGTRAP` will then be taken upon return.

Breakpointing is similar to single-stepping. In step (2) above, one writes a `BPT` instruction at the desired location.

Fig. 3: The *hang* program

```
#include <stdio.h>
#include <signal.h>
#include <sys/pioctl.h>

main(argc, argv)
int argc; char **argv;
{
        int pfd; char procnam[16]; long mask = (1<<(SIGSTOP-1));
        FILE *ttyerr;
        ttyerr = fopen("/dev/tty", "w");
        if (argc <= 1) {
                fprintf(ttyerr, "Usage: %s cmd [args...]\n", *argv);
                exit(1);
        }
        sprintf(procnam, "/proc/%05d", getpid());
        if ((pfd = open(procnam,0)) < 0) {
                fprintf(ttyerr, "cannot open %s\n", procnam);
                exit(1);
        }
        ioctl(pfd, PIOCSMASK, &mask);
        close(pfd);
        fprintf(ttyerr, "%s\n", procnam);
        fclose(ttyerr);
        execvp(argv[1], argv+1);
        perror(argv[1]);
        exit(1);
}
```

Summary

We have described a uniform mechanism for transparent, dynamic communication between a debugger and its debuggee, along with a set of primitives for process control. It is successful largely because it fits so well into the Unix model. We feel that it is probably ill-suited for general inter-process communication, though it should work well for specialized application-oriented debuggers, or other programs which would benefit from clean access to dirty data structures. For example, we have written a version of *ps*(1) which is about four times faster than the standard one.

Acknowledgements

It is a great pleasure to acknowledge many useful and stimulating discussions with T. A. Cargill, D. M. Ritchie, and P. J. Weinberger.

References

[1] Weinberger, P. J. "The Version Eight File System," in *Proceedings of the Summer 1984 USENIX Conference*, Salt Lake City, Utah.