Linux Process Control via the File System

Jonathon T. Giffin George S. Kola

Computer Sciences Department
University of Wisconsin, Madison
1210 W. Dayton St.
Madison, WI 53706-1685 USA
{giffin,kola}@cs.wisc.edu

Abstract

We report on the implementation of process control and inspection via the process file system on Linux. Our work extends the existing Linux process file system by adding four files that collectively give read-write access to a process's address space and allow processes to be controlled and their state observed. Process control follows the model of control in recent versions of the Solaris operating system. The existing Linux process control mechanism, the ptrace system call, changes a process's parent pointer to that of the attached tracing process. Our implementation subsumes ptrace and does not violate this critical data element. Further, this new controlling method allows multiple tracing processes, the ability to detach from a stopped process and leave it stopped, and the ability to trace a subset of system calls, signals, and hardware faults. We detail the four files added to the process file system and describe the control techniques. We examine the delays this enhanced file system adds to critical paths in the kernel, finding that selective system call tracing adds only one clock cycle to the common case of an untraced process. Performance gains are impressive, improving upon ptrace by 374% by allowing fine-grained system call tracing.

1 Introduction

Process control, or the ability to inspect a running process and alter its execution, is a basic requirement for system utilities such as debuggers and security tools. Debuggers must be able to read a process's current state and address space and take at least rudimentary controlling actions, such as inserting breakpoints. A security tool such as a sandboxing utility may require further controlling opportunities. It may monitor system calls, inspecting and changing arguments passed to the calls. All such actions necessitate the ability to observe the current state of a running process and to impose changes in its execution.

Some Unix-based systems, including Solaris and IRIX, implement process control as a virtual file system [3,8]. The file system represents each running process by either a single file or a collection of files, depending upon the system. One process, termed the *tracing process*, ascertains the state and modifies the execution of another process, the *inferior process*, by accessing the files representing the inferior process with standard file system operations: reads, writes, lseeks, polls, and ioctls. Writing to these virtual files does not store data in the file system but rather passes directives to the kernel indicating what control action is to be applied to the inferior process. Some files may allow a tracing process to write directly to the inferior process's address space. Information indicating current process state may be read from the files. A poll on a virtual file for a process returns when the process stops due to a controlling action by the tracing process, serving as notification of an event. Historically, these process file systems are mounted on /proc and provide a rich set of control primitives.

Alternatively, the system call ptrace exists on some systems, such as Linux, as a process control mechanism [4]. A tracing process calls ptrace with arguments identifying the inferior process and specifying the controlling action to be applied. When using ptrace, the kernel inserts the tracing process as the parent of the inferior process. When the inferior process stops due to a controlling action, the kernel sends a signal to the tracing process. Upon receipt of the signal, the tracing process calls the wait system call to receive the stopping event information and can then take appropriate controlling action. The actions supported by ptrace are typically a subset of those available on systems implementing process control via the file system. Systems like Solaris that support process control via /proc do not provide kernel support for ptrace but implement it as a library call that uses the process file system.

The existing Linux kernel implements only minimal support for process control via the ptrace system call and does not support process control through the file system. Linux does provide a /proc virtual file system, although process-specific data is typically limited to that required for a specific program like `ps`. Our work adds process control to the existing Linux process file system and improves tracing efficiency with increased kernel support. Taking the Solaris process file system [8] as our model, we modify a stock Linux 2.4.17 kernel to include four new virtual files in the /proc directory for each running process:

- *ctl*. A tracing process writes special commands to this file to alter the execution of the inferior process.
- *pstatus*. A process reads detailed execution state from the `pstatus` file.
- *as*. The `as` file allows read-write access to the process's virtual address space.
- *sigact*. A tracing process reads the current disposition of an inferior process's signals from this file.

This process control implementation provides the following:

- Efficient reading and writing of process memory.
- Multiple tracing processes per inferior process.
- Detailed and structured process execution status.
- Specification of a subset of signals, system calls, or faults of interest.
- Leaving a stopped process stopped when detaching.

The performance of our enhanced kernel differs little from the performance of an unmodified kernel. In fact, our reduction of the inefficiencies imposed by ptrace improves the performance of certain inferior processes. We do increase the kernel per-process memory requirements slightly, adding to the cost of initializing a process. See Section 4 for a detailed description of performance impact.

We tailored our objectives to meet particular shortcomings of ptrace. We acknowledge other solutions were possible; for example, we could have modified the implementation of ptrace itself to increase its usefulness. However, we believe that the file system is well suited to support process control. We now discuss the shortcomings of ptrace and the motivation behind implementing greater support for process control via the /proc file system.

The implementation of ptrace on Linux is deficient–it alters the state of the inferior process, potentially changing the process's behavior. The kernel inserts the tracing process as the parent of the inferior process, overwriting its original parent. Two consequences of this data alteration are detrimental to tracing utilities. First, this breaks any program

that expects notification via wait when a child process stops. The notification will be sent instead to the tracing process. Second, the use of the parent pointer limits the number of simultaneous tracing processes to one, for there is only one such pointer. A programmer wishing to debug a tracing tool cannot do so by attaching a debugger to the tool's inferior process and observing the actions taken upon the process by the tool. In contrast, process file system implementations allow a many-to-many relationship between inferior and tracing processes.

The programmer could approximate simultaneous tracing by stopping the inferior process, detaching her tool, and then attaching the debugger to the inferior process to observe the previous changes made by her tool. However, the implementation of ptrace does not allow tracing processes to cooperate in this manner. When a tracing process detaches from an inferior, the inferior resumes its execution. Thus, when attaching the debugger, the programmer may not see the state of the inferior process present when her tool detached. Our /proc implementation remedies this problem by allowing the tracing process to specify whether the inferior process should remain stopped, resume execution, or be killed by the kernel when all tracing processes have detached.

The minimalist nature of ptrace produces large overheads in user code that could be avoided with greater kernel support. If a tracing process wishes to trace *any* system call, it must trace *every* system call. The situation is worse for signals. A tracing process has no choice: if it attaches to an inferior process, it must trace *every* signal delivered to the inferior. When a signal or system call forwarded to the tracing process is of no interest, it will return the signal or system call back to the inferior process for execution. The inferior process suffers overhead from its initial stop, task switches first to the tracing process and then back to the inferior, and its resumption of execution. Greater kernel support avoids these overheads by passing a signal or system call to the tracing process only if it is specifically designated to be of interest. In fact, our experiments in Section 4 reveal a speedup of 374% with such kernel support.

Writing to an inferior process's address space is likewise inefficient. Ptrace supports writing to the address space only in word-size data blocks. To write a larger block of data, a tracing process must make repeated calls to ptrace. Our `as` file allows arbitrarily-sized reads and writes of the inferior process's address space.

We could have changed the implementation of ptrace to both remedy these faults and add richer functionality. We chose instead to enhance the existing /proc virtual file system for four reasons. First, when considering how to best change ptrace, we found ourselves naturally shifting to a file system view of a process. Operations that would be required of a boosted ptrace mirrored standard file system operations. For example, suppose we alter ptrace so that a tracing process is not inserted as the parent of the inferior process but instead kept on a list of processes waiting for the inferior process to stop. The kernel adds a tracing process to this list and puts it to sleep when it calls ptrace with some new argument. When the inferior process stops, the kernel awakens all tracing processes kept on this list and their calls to ptrace return. To allow a process to trace multiple inferior processes, the ptrace argument could specify a set of processes, and the ptrace call returns when any process in the set halts. This very nearly describes polling on an array of file descriptors.

Second, such modifications to ptrace would dramatically change the semantics of its use. Existing programs written for ptrace would be incompatible with future systems.

Lastly, shifting from process control via ptrace to control through the file system follows the trend of many UNIX-style systems. Using the file system interface on Linux allows users of other systems to more easily port their tracing programs to Linux and lets Linux users leverage code written for other systems.

2 Process File Systems In Other UNIX Systems

Kernel support for a rich and efficient set of control primitives used via the file system reflects a common design trend in other UNIX-style systems. We briefly review four other systems: System V Release 4, Plan 9, IRIX, and Solaris. Solaris provides the most extensive support for process control, so our implementation mirrors its design.

In System V Release 4, a process is represented by one virtual file in the /proc file system named by its process ID. This representation is identical to that used in Solaris through version 5 [7]. The kernel supports process control via standard file system calls applied to this file [1]. Reading from and writing to the file accesses and updates the inferior process's virtual address space. A tracing process controls the inferior process by calling ioctl on the virtual file and passing the control instruction as an argument. As in our work, a subset of system calls or hardware faults may be traced.

IRIX 6.5 is virtually identical [3]. Each process is represented by a file that allows read-write access to the process's address space and the passing of control arguments using ioctl. Tracing processes can poll on the file descriptor for this file, with the poll returning when an event of interest has occurred.

Plan 9 models everything as a file, including the running processes, and allows process control via the /proc file system [6]. A tracing process controls the inferior process by writing commands to the inferior process's `ctl` file. The operating system makes remote debugging trivial by importing the /proc file system from one machine into another. It allows reading and writing of a process's address space via the file system. The process control mechanism does not allow selective tracing of system calls or hardware faults.

Beginning with version 6, the Solaris /proc file system took the following structure: Each process is represented by a directory containing a collection of files for process control and inspection [8]. A tracing process controls the inferior process by writing commands to the inferior process's `ctl` file. It reads detailed state from files such as `status` and `sigact`. The `as` file gives a tracing process read-write access to an inferior process. The kernel allows selective tracing of system calls, hardware faults, and signals. It also allows multiple processes to control an inferior process.

Solaris takes the hierarchical structure one step further by including a subdirectory `lwp`, itself containing a subdirectory for every lightweight process within the inferior process. These subdirectories each contain files to inspect and control the execution of an individual thread of execution.

Existing Linux kernels allow an inferior process's address space to be read from the `mem` file in the process file system. However, the tracing process must first attach to the inferior process with ptrace before this file may be read. This implementation limits readers to one and makes observation of the address space and actual process control mutually exclusive operations. Our `as` file is essentially the existing `mem` file with implementation changes to support multiple tracing process and address space writing.

File	Access	Description
pstatus	read	Contains detailed process state.
sigact	read	Contains the current signal dispositions for the inferior process.
as	read / write	Provides read/write access to the inferior process's address space.
ctl	write	A tracing process controls an inferior process by writing commands to the inferior process's `ctl` file.

Table 1: The four files we have added to each process directory in the Linux process file system.

3 Linux Process File System Additions

In this section, we discuss our additions to the process file system in Linux 2.4.17. We have included four new virtual files that permit access to detailed process state and fine-grained control over its execution. The additions correspond generally to the structure of the process file system on Solaris 8, with the Solaris `status` file named `pstatus` here due to a name conflict. Table 1 lists the four new files, *pstatus*, *sigact*, *as*, and *ctl*, and a description of each.

A tracing process controls an inferior process by specifying events of interest and then taking action when such an event occurs. Events include crossing a system call entry or exit point, a signal delivery, a hardware fault, or a stop directed by the tracing process. Following a stop, the tracing process may resume the inferior process or alter its execution by modifying register values or canceling the action that produced the stop. The remainder of this section describes the relations between these actions and the four virtual files we have added.

3.1 The pstatus File

The read-only `pstatus` file gives detailed information about an inferior process when it is stopped on an event of interest. A tracing process reads a structure from the file, and the structure fields give information about the process and the stop. Fields indicate what type of event occurred (e.g. signal received) and specific details for that event (e.g. SIGCHLD). Further fields show more general low-level process details, including the general register values and the process's credentials. The tracing process can use the information read from this file to determine what controlling action to take upon the inferior process.

A tracing process can view the following data about an inferior process at any point during the execution of the inferior process:

- Identification: the process id, the parent id, the process group id, and the session id;
- Memory layout: the starting virtual address and size of the process's heap and the virtual address of the process's stack;
- Signals: the set of blocked signals, the set of pending signals, and the set of signals being traced by one or more tracing processes;
- Faults: the set of traced faults;
- System calls: the set of traced entry points and the set of traced exit points.

Further data is meaningful when the inferior process stops on an event of interest:

- Stop information: a flag value indicating the type of stop, with a detailed reason;
- Trace trigger: the system call number, fault number, or signal number being traced that generated the stop;

- Registers: the values of general and floating point registers.

This file permits asynchronous notification of an event of interest, allowing a tracing process to trace multiple inferior processes and take action when any of the inferior processes stops. The poll system call will return and indicate priority data is available when an event occurs in the inferior process. This definition of priority data for files in the process file system matches that used by IRIX [3] and Solaris [8].

3.2 The sigact File

The `sigact` file is a read-only file showing the current disposition of signals in the inferior process. A tracing process reads an ordered array of `struct sigaction` from the file, each element indicating the action taken for a single signal. We separate this data from the `pstatus` file because it is not expected to change frequently during the inferior process's execution. Conversely, the data in the `pstatus` file changes following every stop of the process on an event of interest. Separating data based upon frequency of change increases the efficiency of read operations.

3.3 The as File

The `as` file allows read-write access to an inferior process's address space. Similar to the existing `mem` file, the `as` file does not first require a tracing process to be attached to the inferior process using `ptrace`. As a result, our file allows multiple readers and writers to have the file opened at any one time. We use a multiple reader / single writer semaphore to ensure synchronization among individual reads and writes; multiple reads or writes may be interleaved with reads and writes from other tracing processes. A tracing process requiring exclusive access to the inferior process's address space may open the file exclusively (see Section 3.5).

3.4 The ctl File

A tracing process writes specially-formatted commands to the `ctl` file to specify events of interest and actions to be applied to the tracing process. This file is equivalent to the `ctl` file in the Solaris 8 /proc file system, although we have implemented a subset of the commands understood by Solaris. These commands appear similar to the arguments given to the `ptrace` call: an identifier indicating the requested control operation and any argument required for that operation. The tracing program writes the identifier and any required argument to the `ctl` file of the inferior process with the standard write call.

Our set of commands includes the following:

PCDSTOP: Directs the inferior process to stop. We use the `ptrace` stop technique of sending the inferior process a SIGSTOP signal. If the tracing process is tracing SIGSTOP, the inferior process will stop first at the receipt of the signal and again when the signal is delivered.

PCSTOP: Directs the inferior process to stop and waits for the stop to complete. The write of this command will not return until the inferior process has stopped.

PCWSTOP: Waits for the inferior process to stop. This is a synchronous notification of reaching an event of interest, blocking the write until the event occurs.

PCTWSTOP: Waits for the inferior process to stop, with millisecond time-out. The write of this command blocks until either the inferior process stops on an event of interest or the time-out expires.

PCRUN: Restarts a stopped process. In the trivial case, the inferior process is simply restarted. However, this command takes an argument that may change the execution of the process based upon the event of interest that stopped it. The tracing process may clear the signal received by the inferior process if it is tracing that signal. Similarly, a traced hardware fault may be cleared. In both cases, execution continues as though the signal or fault never occurred. When the inferior process stops at the entry point of a system call, the tracing process may abort that call, forcing it to return with an error code. Lastly, the tracing process may turn on single-step execution in the inferior process.

PCSTRACE: Sets the traced signals. The tracing process uses this command to define the set of signals upon whose receipt the inferior process will stop. Tracing signals is not an all-or-none affair as with `ptrace`, but allows fine-grained control over the set of signals forwarded to the tracing process.

PCCSIG: Clears the current signal. If the inferior process is stopped at the receipt of a traced signal, this command clears the signal so that it will not be delivered upon restart of the process.

PCSSIG: Sets the current signal. If the inferior process has stopped due to a signal, this command will change the signal that will be delivered to the process upon restart.

PCKILL: Sends a signal to the inferior process. This command sends a specified signal to the inferior process regardless of the current state of the inferior process. Although most commands carry meaning only for stopped inferior processes, such state distinctions are meaningless for this message.

PCUNKILL: Withdraws a pending signal. A pending signal is withdrawn, even if it is not the next signal to be delivered.

PCSHOLD: Defines the set of held, or blocked, signals.

PCSFAULT: Sets the traced faults. This command defines the set of hardware faults that stop the inferior process. The stop will occur before the kernel dispatches a signal indicating the fault to the process. Page faults cannot be traced.

PCCFAULT: Clears the current fault. The current fault is cleared without restarting the inferior process. When execution does resume, the kernel will neither deliver the signal for the fault nor execute the fault handler. The instruction that generated the fault will be retried by the processor; optionally the tracing process may handle the fault so the instruction will succeed upon resumption of the inferior process.

PCSENTRY: Define the traced system call entry points. This set defines the system calls whose entry points are events of interest. The inferior process will stop after entry to the general system call service routine but before the kernel executes code for the specific system call. The tracing process may alter the arguments passed to the system call by overwriting architecture-specific registers.

PCSEXIT: Define the traced system call exit points. Similar to the previous command, this message defines the set of system calls with events of interest at their exit points. The inferior process stops after the specific system call

code executes but before control exits kernel-space. The return value from the system call may be changed by overwriting architecture-specific registers.

PCSET: Sets inferior process mode of execution. The tracing process writes an argument to the control file containing flag values describing how the inferior process should execute:

- Run on last close. When all tracing processes have detached from the inferior process, all internal tracing state is reset and the process is set running. In the absence of this flag value, the inferior process maintains its tracing state and execution state after the last detach. A tracing process can detach from a stopped inferior process and leave it stopped, allowing another tracing process to later attach to the inferior process and resume its execution.
- Kill on last close. If this flag is set, the inferior process will be sent SIGKILL following the detach of the last tracing process. If both run- and kill-on-last-close are set, the file system favors kill.

  PCUNSET: Turns off any flag value previously set by PCSET.

  PCSREG: Sets the general registers of the inferior process.

  PCSFPREG: Sets the floating point registers of the inferior process.

  PCSFPXREG: Sets the extended floating point registers of the inferior process.

3.5 Exclusive Access

Our process file system supports multiple simultaneous tracing processes for any one inferior process. If the tracing processes are not cooperative, commands written to the control file may become interleaved, sending the inferior process into chaos. When opening any `/proc/<pid>` file for writing, a tracing process may specify that it must hold exclusive write privileges while it holds the file. The commands read by the kernel will be exactly those written by the single tracing process holding exclusive access. The exclusive privileges apply not only to specific file that was opened, but to all `/proc/<pid>` files that may be opened for writing. This prevents one tracing process from writing to an inferior process's address space while another tracer writes commands to the control file. Stated differently, opening a file with exclusive write access limits the number of attaching processes to one. This process must detach by closing the file before any other tracing process can attach.

An exception exists for self-opens, the special case where a process attaches to itself. Although a process may open its own /proc files for writing just as any other tracing process, the kernel treats exclusive access differently for these self-opens in two ways. First, a process cannot open itself exclusively; and, second, a self-open will always succeed, even if another tracing process holds exclusive write access. In the absence of the former, a process could attach to itself exclusively to prevent external observation and control. Allowing such an action violates security as process monitoring becomes impossible without kernel modifications. The latter constraint prohibits an external tracing process from preventing a self-open by the inferior process.

Stock Linux 2.4.17 kernel	417
Kernel with enhanced /proc	418

Table 2: Clock cycles executed for unimplemented system call.

```
pushl %eax
SAVE_ALL
GET_CURRENT(%ebx)
testb $0x20, tsk_ptrace(%ebx)
jne tracesys_entry
...
```

Figure 1: System call entry point code. We add only the two instructions in boldface.

3.6 Security

Processes that exec a setuid program are vulnerable should a tracing process continue to trace the new program, now executing with different privileges on the system. The kernel must impose adequate security restrictions to prevent such a shift of control. Solaris has an elegant solution to this problem, invalidating the vnodes for any open /proc files. We must implement a similar solution in Linux. Although currently unimplemented, the kernel must revalidate the tracing process's credentials when the inferior process execs a new program. Should the credentials not match, the kernel will close the /proc files open by the tracing process.

4 Analysis

In this section, we analyze the performance impact our enhancements have upon kernel and tracing tool performance. The nature of this work more readily lends itself to theoretical impact analysis rather than to an experimental methodology, and our discussion follows this approach. Our analysis of the kernel examines the added overhead from system call, signal, and fault tracing. Where we did measure overhead, we ran the kernel on a 667 Mhz Pentium 3 machine with 128 MB of memory.

We measured the additional overhead of system call tracing by measuring the number of clock cycles required to execute an unimplemented system call with no traced calls. This measurement is valuable, as it indicates the overhead of untraced execution, the common case. The kernel handler for an unimplemented call returns immediately, allowing an accurate measure of the overhead of switching to kernel-mode, checking for system call tracing, and returning to user-mode. The assembly language instruction `rdtsc` returns the value of the clock cycle counter of the processor [2]. We used this instruction to measure overhead in clock cycles. We used the `cpuid` instruction to flush the processor's pipeline before probing the clock cycle counter. As per Intel's recommendations, we warm the `cpuid` instruction before use. Table 2 shows the clock cycle counts for calls to an unimplemented system call on our modified kernel and on an unmodified kernel. We generated these counts by averaging 1000 cache-warm calls to the system call. As shown, the overhead is minimal: the tracing test completes in just one extra clock cycle.

We expected this good result. Figure 1 shows the assembly language code snippet executed at the entry point of all system calls. We have added only two instructions, shown in bold, that test for system call tracing through the /proc file system. The `testb` instruction tests a one bit flag value in an element of the process structure for the process generating the system call. If the bit is set, the `jne` instruction jumps to the specified label. This code reflects optimization for the common case of untraced execution by using only this simple, fast test on the system call critical path. When system calls are untraced, the code follows straight-line execution and benefits from cache locality and instruction prefetch. This code checks only if another process is tracing any system call entry or exit point. If any call

```
if (current->pr_flags & (1UL << trapnr)) {
...
}
```

Figure 2: Hardware fault critical path trace check code.

is being traced, the jump is taken and only then is the more expensive test made to determine if a process is tracing this particular system call.

Faults and signals have no unimplemented or null member, so we were unable to measure the trace check overhead. We instead examine the additional code on the critical path, demonstrating that the additions are slight. When a fault occurs, the kernel immediately tests the fault number to determine if a process is tracing the fault (Figure 2). For implementation reasons, this code fragment is written in C, but the additional code in the critical path is the `if` statement. This statement corresponds to three assembly language instructions: a bit shift, a bit test, and a conditional jump. Note that by testing the fault number immediately, the bit shift instruction is the only additional cost beyond the flag test used in system call tracing. We then expect the overheads to be comparable.

Signals, unfortunately, incur slightly greater cost on their critical path. Figure 3 shows the C language test for signal tracing and the code for a function used in the test. As with faults, this test immediately checks the signal number against those traced by another process. The test operates on a structure containing an array of bit flags, requiring additional instructions on the critical path. All signals being delivered to all processes, including untraced processes, incur this overhead.

By including kernel support to trace a subset of system calls, we eliminate the task switch overheads ptrace suffers when forwarding an untraced call. We measured the process tracing overhead incurred when a process traces a subset of system calls, but not the current call. With its all-or-none semantics, ptrace will forward this untraced call to the tracing process, although the tracing process will immediate return the call to the inferior process for execution. Because we allow tracing processes to specify the subset of calls of interest, our implementation performs better than ptrace. We measured this improvement by timing the system call latency in a process traced using either ptrace or our enhanced /proc file system. The inferior process calls an unimplemented system call so that the kernel system call handler will return immediately. In both scenarios, the tracing process traces a subset of system calls not including that called by the inferior process. Table 3 compares the overhead incurred by the two tracing techniques. By testing the system call number inside the kernel and discovering it is untraced, our implementation is 374% faster than ptrace. The cost to stop the inferior process, switch tasks from the inferior process to the tracing process and back, and resume the inferior process contribute to the extra ptrace overhead. We incur only the cost of checking the system call number to determine if it is of interest to any tracing process.

```
if (sigismember(&current->pr_traced, signr)) {
...
}

inline sigismember(sigset_t *set, int signr) {
    return 1 & (set->sig[signr/32] >> (signr % 32));
}
```

Figure 3: Signal trace critical path trace check code.

Stock Linux 2.4.17 kernel, using ptrace	6758
Kernel with enhanced /proc, using /proc control	1808
Speedup	374 %

Table 3: Clock cycles executed for an untraced system call when other system calls are traced. The overhead values indicate time in the inferior process between the `int 0x80` kernel interrupt and receipt of control back from the kernel.

5 Further Work

We highlight three areas of further work: architecture independence, multiprocessor testing, and three additional process control options.

We have confined our efforts to the x86 architecture. Much of the internal support for process control is low-level and architecture specific, so we must write and test equivalent code for the other processors supported by Linux before submitting our code for review. Confining our additions to the x86 architecture is a possible, though undesired, solution.

Our testing is incomplete. We have focused on uniprocessor systems, a small subset of the machines running Linux. Tools such as Paradyn, a parallel debugger, rely upon support for multiprocessor computers. We have architected our code to support the synchronization and serialization required by multiprocessors but have not tested it in such machines.

Our implementation includes a large subset of the Solaris process control functionality, but three controlling techniques remain missing. First, we do not yet include memory references to watched areas as events of interest. On Solaris, a tracing process may specify memory ranges so that the inferior process stops at any reference to an address in this range. We considered this a low priority task and did not reach it in our implementation time frame. Second, we have not implemented a ptrace compatibility mode, or an operational mode that mimics the semantics of the ptrace system call. This mode would allow ptrace to be implemented as a library call that uses /proc to control files. Ptrace-specific code could then be removed completely from the kernel unless required to implement the compatibility mode. We did not implement this setting due to time constraints. Third, we do not allow a tracing process to abort a blocked system call in the inferior process. We note that this constraint tempers the effectiveness of debugging tools, but could find no reasonable implementation method given the nature of blocking system calls in Linux. We expect that significant kernel modifications will be required to allow a blocked process to unwind out of a blocked system call and remain optimistic that intensive research may uncover a reasonable solution.

6 Conclusions

We have implemented kernel-level support for a /proc file control interface on Linux. This interface allows tracing processes to read detailed process state from, and write commands to, virtual files in the file system. Our process control implementation supports both a larger set of operations and more finely-grained control than ptrace. Moreover, we add support for multiple tracing processes and for leaving a stopped process stopped upon detach. We do not modify critical fields of process data structures, allowing our process control to be used on a wider class of programs than ptrace. The inferior process state available for reading from the file system has also been greatly augmented, present-

ing tracing processes with a detailed view of the inferior process's state. The additional overhead from our kernel enhancements is minimal, although the performance gains are dramatic.

7 Acknowledgements

We thank Vic Zandy and Alex Mirgorodskii of the Paradyn team for their motivating suggestions and examples of process file system use in Paradyn.

8 References

[1] R. Faulkner and R. Gomes, “The Process File System and Process Model in UNIX System V”, *USENIX Conference Proceedings*, Dallas, Texas, January 1991.

[2] Intel Corporation, *Using the RDTSC Instruction for Performance Monitoring*, 8 May 2002, `http://cedar.intel.com/software/idap/media/pdf/rdtscpm1.pdf`.

[3] IRIX 6.5 Reference Manual, Section 4, proc.

[4] Linux Reference Manual, Section 2, ptrace.

[5] Linux Reference Manual, Section 5, proc.

[6] R. Pike, D. Presotto, K. Thompson, and H. Trickey, “Plan 9 from Bell Labs”, *Summer 1990 UK Unix Users Group Conference*, Summer 1990.

[7] Solaris 5 Reference Manual, Section 4, proc.

[8] Solaris 8 Reference Manual, Section 4, proc.