Unix as an Untrusted Library

Tom Pinckney
MIT Laboratory for Computer Science
pinckney@lcs.mit.edu

1 Introduction

Unix was born a small simple operating system. Today it's a large, complicated, feature-rich system. As more and more features are added, the overall system complexity increases and performance and reliability suffer. Further, the pace of technical innovation is slow since modifying a complex kernel is hard. Thus, few of the ideas produced by systems research over the past twenty years have been incorporated into mainstream Unices.

Pushing more, new, or better abstractions into the kernel is not the right thing to do. First, this can only be done by very few people rather than anyone who uses the system. Second, writing kernel code is hard since it must securely multiplex many application's competing demands. Third, composing many features together in the kernel usually forces all applications to pay for all features even though each application only depends on a few features. For example, copy-on-write and memory mapped files are critical for those applications that use them, but needlessly slow and complicate the VM system for all applications.

We propose implementing Unix as a set of untrusted libraries that each application can link in, customize, or completely replace. We have developed a set of libraries, collectively called *ExOS*, that provides a large subset of the POSIX interface. The rest of this paper will describe our design, present several micro-benchmarks and finally discuss applications that exploit the fact that ExOS is merely an untrusted library linked with the application.

2 ExOS

Effectively, we want to give each application its own "single-user" version of Unix that the application can do with as it wishes. This requires a new style of kernel that gives direct access to the underlying hardware without abstracting it. Kernels must also allow applications to allocate resources by physical name and expose revocation of these resources. Finally, the kernel must insure that applications only use those resource they are permitted access to. This style of kernel allows ExOS to abstract a machine's resources in an applications specific manner, yet prevents buggy or malicious applications from harming each other. We call kernels that are written this way *exokernels* [1].

Traditional Unix kernels function as a safe central site to keep state and to make global resource scheduling decisions. ExOS has no equivalent place since every application has it's own library. Complicating matters further, each application's ExOS may not trust the others since each may have been arbitrarily modified.

The three immediate problems posed by this design are how to perform global process scheduling, how to divide up the physical memory of the machine under high load, and where to place centralized state such as file descriptor and process tables. We briefly consider possible solutions, both for trusting and untrusting applications.

First we consider scheduling. Each application is allocated a single timeslice from the kernel. A timeslice corresponds to some amount of running time on the processor. Cooperating processes can schedule themselves by communicating through shared memory and yielding their time slices to each other. For example, a process waiting for some I/O event can yield its timeslice to some non-blocked process. Mutually untrusting processes can all trust a single scheduling server that allocates many time slices and hands them out to processes according to some scheduling policy. Of course, different groups of processes can each have their own scheduling servers.

Applications divide physical memory amongst themselves based on their knowledge of total demand for pages from all applications and kernel imposed quotas. Since allocation and revocation of pages are exposed to applications, each application is explicitly aware of how many pages it has and how strong demand from other applications is, based on the frequency of revocations and the number of free pages in the kernel. Thus, cooperating processes can balance their memory needs against global memory needs. Further, making applications aware of their use of physical pages allows more efficient use of that memory. For example, when memory is scarce, an application can chose to discard it's least valuable pages first. Finally, kernel quotas prevent malicious or buggy applications from monopolizing memory.

Shared state, such as file descriptor tables, can simply be allocated in writable shared memory that each application maps. Memory can be mapped high into the address space to reduce the chance that wild writes will corrupt it. Untrusting applications can map state read-only and very quickly IPC to servers that will guarantee well-formed updates.

Using some of these techniques, we have implemented three variants of ExOS, each with increasingly sophisticated virtual memory. This provides an easy way for application-writers to chose the features they need without having to modify any code. The first variant uses linear page tables stored in a

Supervised by M. Frans Kaashoek. Supported by ARPA contract N00014-94-1-0985 and by a NSF National Young Invesitgator Award.

System	appel1	semaphore	signal	select
ExOS	19	7	42	.4
Ultrix	262	358	56,640	63

Figure 1. Several Unix micro-benchmarks; times are in microseconds

single page. Its interface is richer than other virtual memory system we know of: it provides flexible support for aliasing, sharing, disabling and enabling of caching on a per-page basis, specific page-allocation, DMA, etc. The second ExOS variant provides the first implementation of clustered page tables to manage large sparse address spaces. Clustered page tables were proposed by Talluri et al, who could only study their benefit in simulation as the complexity of the solaris kernel prevented an actual implementation. Finally, the third version adds paging to disk or over a network. [2]

The speed of ExOS is very good. Figure 1 lists several micro-benchmarks comparing ExOS running on an exokernel to Ultrix 4.2, a stable well-tuned Unix. All tests were performed on a Decstation/125 with a 25Mhz MIPS R3000 processor with only a single user on the machine.

appel1 is a vm-intensive benchmark that measures the time to segfault and then protect a page and un-protect another using mprotect. This is representative of the type operation many VM intensive systems such as software DSM and garbage collectors depend on for good performance.

semaphore is the time to acquire a semaphore, update a shared word of memory, and then release the semaphore. ExOS manages its own scheduling, so each application delays giving up its timeslice if that would cause the application to be de-scheduled in the middle of a critical section. Efficient user-level locks are easily built on top of these optimistic critical sections.

signal is the time for two processes to ping-pong signals back and forth. Ultrix doesn't provide a mechanism for donating timeslices so each process sends a signal to it's partner and spins waiting for a response which won't come until the other process is scheduled. ExOS easily allows each process to send a signal and then immediately donate the rest of the current timeslice to the receiving process.

select is the time for a process to poll two file descriptors referring to UDP sockets to see if either has any data waiting to be read. The ExOS version of this code performs the select in 9 cycles since it can be completely specialized and inlined to this particular case.

3 Applications

We have built numerous applications using ExOS, including a remote login daemon, a command shell, and a prototype web server. Space constraints prohibit an in depth look at these. Instead, we present three interesting examples of how applications can extend ExOS to their own particular needs.

The first is a TLB monitoring program. ExOS runs on MIPS chips which have a software TLB. ExOS is responsible for handling TLB-refill requests, so an application can simply modify ExOS to record the address being faulted on. The MIPS exokernel provides a kernel-level cache of TLB entries to avoid going to user level on every miss. The TLB-monitor disables this cache for the application being studied and records every TLB miss. This required changing seven lines of code in ExOS and resulted in only a two to three time slowdown, depending on the locality of the application being monitored. This is far better than would be possible under Unix, if it would be possible at all.

The second program uses application specific page coloring. Since ExOS is responsible for allocating a program's physical pages, the program can simply chose to allocate memory that takes the machines cache into account. We wrote a little program that performs a database join-like operation of comparing a single page of data to each of 64 other pages. The application allocates the single page to one partition of the cache while allocating all 64 other pages to a second partition, thus limiting cache pollution. We compared this application to other variants which did not use application specific knowledge about which colors to use for pages. The worst case running time for the application specific version was never worse than the best cases of the non-application specific versions. Traditional page coloring methods have not been implemented in conventional systems even though it only took about 40 lines of code to add minimal coloring to ExOS. Further, even if conventional systems did have coloring, they would still be blind to this application specific knowledge.

Finally, our third application provides a transactional virtual memory, similar in spirit to RVM as described by Satyanarayanan et al. The implementation they describe, however, is not integrated with the virtual memory system. On commit, a transaction's data is written to a stable log. Thus, the memory pages containing the committed data are in reality "clean" since their data is all on disk. However, there is no way to tell the VM system this so the VM manager dutifully writes out these pages to disk when it needs to swap them out. We implemented a version of RVM that is integrated with the VM code so it avoids this unnecessary page out. Under heavy loads this can substantially reduce the amount of data paged out. This optimization is part of a wide class of problems encountered when applications cache data in virtual pages that the underlying VM system blindly pages.

4 Conclusions

We have described implementing Unix entirely as an untrusted application-level library. ExOS, our library, is reasonably complete and very well performing. Moving traditional Unix kernel code to application level enables customizing and extending the code on a per-application basis.

References

[1] D. R. Engler, M. F. Kaashoek, and J. O'Toole Jr. Exokernel: an operating system architecture for application-specif ic resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, December 1995.

[2] D. R. Engler, M. F. Kaashoek, and T. Pinckney. AVM: application-level virtual memory. Submitted to OSDI '96.