CSC2227 (Winter 2008)
Fletcher: Final Project Report

Eric LaForest

Contents

List of Figures

List of Tables

1 Introduction

Amongst the various operating system design options, I find the microkernel architecture particularly compelling due to its careful separation of services into separate, mutually protected modules. This organization makes each module smaller and easier to reason about, improving the odds of a correct implementation.

However, this division also introduces the need for communication across modules. Since modules are protected from each other by placing them in disjoint virtual memory spaces and/or privilege levels, any OS operation not entirely contained inside a single module will require a system transition from user mode to kernel mode and back. This overhead is thus omnipresent and may affect system performance.

2 Goal, Motivation, and Previous Work

This project seeks to explore the possibility that an alternative system design, in terms of processor architecture and memory model, could reduce the latency of user/kernel mode switching. *Specifically, can the fast subroutine call mechanism of a second-generation stack architecture be used to implement a comparably fast mode switch?*

The speed of the mode switch between user and kernel is important to the overall performance of an operating system as it places a lower limit to the time taken to perform a system call (syscall) and thus to the speed of I/O operations [Hay02]. Thus, the speed of syscalls is measured as an indicator of potential system performance (Table 1).

Various approaches have been used to keep syscall times to a minimum. The Xen paravirtualization system [BDF$^+$03] allows the installation of 'fast' exception handlers for syscalls that do not alter the global system state and thus do not require entering the hypervisor, causing no mode switch.

The L^{4}Linux microkernel [HHL$^+$97] transforms syscalls into Inter-Process Communication (IPC) between processes, adding several hundred cycles. However, L^{4}Linux enjoys extremely efficient IPC by remapping virtual memory pages instead of copying. This optimization compensates for the overhead in all but the simplest syscalls.

Finally, at the other extreme, the Singularity operating system [HHA$^+$07] completely eliminates mode switches by running all processes in the same address space and privilege level, using strong compile-time language-based checks to ensure safety, resulting in extremely fast IPC and syscalls. However, this approach requires some strong constraints within processes, such as being unable to load or generate code at run time.

All of the above works acknowledge that virtual memory can pose a problem for syscall performance, as switching modes means entering a different virtual memory space, which requires changing the page table and flushing the Translation Look-aside Buffer (TLB). This overhead can add several hundred cycles to a mode switch [Lie95] and negatively impact the performance of systems which are I/O-bound in some form, such as IPC in microkernels or disk and network traffic in servers.

Paper	Operating System	syscall μs	cycles	Platform
[FHL+96]	Fluke	2	400	200MHz Pentium Pro
[BSP+95]	DEC OSF/1	5	665	133MHz Alpha AXP
[BSP+95]	Mach 3.0	7	931	133MHz Alpha AXP
[BSP+95]	Spin	4	532	133MHz Alpha AXP
[BDF+03]	XenoLinux 2.4.21	0.46	1104	2.4GHz Xeon
[BDF+03]	Linux 2.4.21 (UP)	0.45	1080	2.4GHz Xeon
[HHA+07]	Singularity	-	91	2GHz Athlon 64 X2
[HHA+07]	FreeBSD 5.3	-	878	2GHz Athlon 64 X2
[HHA+07]	Linux 2.6.11	-	437	2GHz Athlon 64 X2
[HHA+07]	Windows XP SP2	-	627	2GHz Athlon 64 X2
[HHL+97]	Linux 2.0.21	1.68	223	133MHz Pentium
[HHL+97]	L^4Linux 2.0.21	3.95	526	133MHz Pentium
[HHL+97]	MkLinux 2.0.28	15.41	2050	133MHz Pentium

Tab. 1: Comparison of simple system call times

3 Design

The entire system exists inside a Linux process as a structural, cycle-accurate stack machine simulator without any knowledge of the host system. The virtual hardware of the simulator supports a small extensible language run-time which provides an interactive environment capable of compilation. Both hardware and software required changes from their original implementation [LaF07] as part of the project execution.

3.1 Hardware

3.1.1 Memory Model

Figure 1 shows the structure of the address space. At the bottom is the installed RAM while memory-mapped I/O sits near the top. Unused memory space is mapped by the kernel to disk blocks[1]. The entire memory is flat and physically addressed, thus a unique address is sufficient to identify anything within the address space. In terms of performance, this is comparable to a virtual memory which never faults and never experiences any TLB misses.

[1] This is rather restrictive for 32-bit systems, but given a 64-bit address space and the current 3-year doubling period for disk storage [HP07, 6.2], this should allow for up to a century before the address space runs out in a simple system.

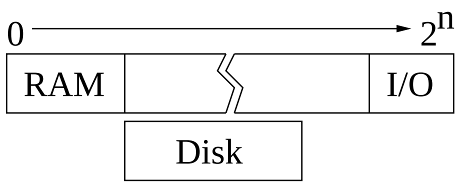

Fig. 1: Structure of the system address space

3.1.2 Peripherals

The system peripherals are memory-mapped at the top of the address space.

Input/Output Ports These ports are used to read in commands and write responses to the console. For simplicity, they block the simulator until the host system can respond.

Cycle Counter When read, the cycle counter returns a 32-bit value which is incremented every cycle. The counter can be set by writing a new value.

Disk Read Port A write to the Disk Read Port sets the current disk block number. Subsequent reads return one integer from that block until it is complete. Further reads begin reading the next block. Disk access latency is not simulated.

Disk Write Port A write to the disk write port sets the number of the block to be overwritten. The next 128 writes[2] will be written to that block, whereupon the port expects a new block number. A read returns zero.

3.1.3 Stack Machine

The simulator implements a simple stack machine (Figure 2) derived from past commercial designs [MT95][Fox98] and from previous undergraduate work [LaF07] . The Data Stack (DS) is the core of the system, where ALU operations are performed and subroutine parameters placed in a Reverse Polish Notation manner. The Return Stack (RS) holds the return address during the execution of subroutines.

The stacks are not data structures in memory, but actual hardware devices which cannot be randomly addressed[3]. Loads and stores take their address from the Address Register (A) and their data from DS. There is a single port to main memory (MEM), capable of one load or store per cycle. Finally, there are the usual Instruction Register (IR) and Program Counter (PC) registers.

[2] 128 integers = 512 bytes = one disk block

[3] The depth of the stacks is arbitrary. Past research [Koo89] has shown that 16 elements is sufficient for most code. This design uses generous, 32-deep stacks to avoid having to deal with overflows.

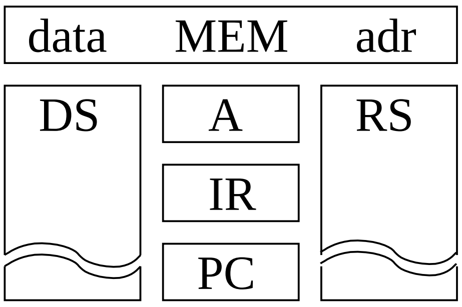

Fig. 2: Simplified block diagram of a stack machine

Subroutine Calls on a Stack Machine The key feature of interest in stack machines is the speed at which subroutine calls can be performed. Most processors must flush some registers to a stack in memory and move the function parameters to certain predefined registers before calling a subroutine. This method takes several instructions and memory accesses to complete.

On the other hand, on a stack machine, the parameters of a subroutine are on the Data Stack, usually placed there as the result of previous computations by the caller. Calling a subroutine requires only to save the current Program Counter on the Return Stack and fetch the first subroutine instruction into the Instruction Register. The called subroutine them simply computes with the values found on the Data Stack, as if it had been in-line code. Returning from a subroutine restores the Program Counter and resumes the fetching of instructions from the caller. Both call and return each require only two cycles and two loads from memory.

3.1.4 Virtualization

To support a clean division between user and supervisor mode, I opted to virtualize the entire machine as per the Popek & Goldberg criteria [PG74], which define the conditions required to present to a user process an interface that is identical to the native machine while preventing the process from usurping its resources[4]. In fact, except for memory and privilege constraints, the user process runs unhindered on the bare hardware.

Figure 3 shows the major components added: The upper and lower range of accessible memory are defined by the contents of the Upper Bound (UB) and Lower Bound (LB) registers. Any out-of-bounds memory access will cause a trap to the kernel. The entry point to the kernel is held in the Trap Program Counter (TPC). The Mode Bit specifies if the stack machine is in User or Supervisor mode.

[4] There is no periodic interrupt in this system so a user process can run for indefinite lengths of time, but this is not a factor in the experiments.

Additionally, I added a privileged Return To User (RTU) instruction to transfer from Supervisor mode to User mode. Attempting to execute RTU in User Mode causes a trap.

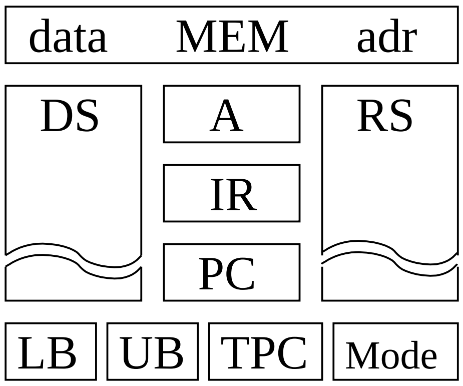

Fig. 3: Additional hardware to support virtualization of a stack machine

3.1.5 From Subroutine Call to Trap Handling

While in User Mode, if a memory access is outside the range delimited by LB and UB, or if the RTU instruction is executed, the machine will trap to the kernel. The state of the machine after a trap is shown in Figure 4: The LB and UB registers are set to encompass the entire address space, preventing any memory access traps from occurring while in Supervisor mode. The Mode Bit is set to Supervisor, enabling the normal operation of the Return To User (RTU) instruction. The previous contents of LB and UB are pushed onto the Data Stack, while those of IR and PC are pushed onto the Return Stack. The PC is loaded with the contents of TPC, and the IR is filled with the first instruction of the trap handler. The A register is untouched. While in Supervisor mode, executing RTU will perform the same steps in reverse, restoring the state of the user process.

One exception is that the saved PC and IR must be swapped by the trap handler before executing RTU. This is so the contents of the IR are popped from the Return Stack and restored in the last cycle of RTU, since overwriting IR will immediately begin the execution of its new contents. This could be avoided by saving IR and PC in the reverse order during a trap, but this complicates trap handling as the instruction that caused the trap would be buried under the saved PC and could no longer be directly moved to the Data Stack for manipulation.

The net effect of these operations is to implement a subroutine call whose saved state includes the memory bounds and current instruction of the caller in addition to the usual Program Counter. Both traps and return from traps take the same amount of time and bandwidth as subroutine calls and returns: two cycles and two loads[5].

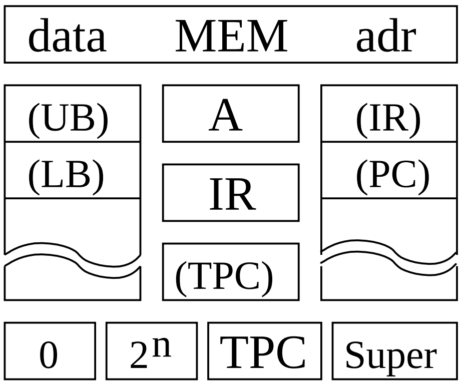

Fig. 4: Machine state after a trap.

3.2 Software

The core software is a minimal, Forth-like language [LaF07]. The kernel is composed of functions which read input, look-up names, define names, and compile stack machine opcodes. The main loop consists of reading the name of a function, looking it up in a dictionary, and executing the associated code. This code may itself define names and compile code, thus extending the language kernel. There is no further syntax.

The bare language kernel is extremely spartan and unsuitable for general programming, thus a number of extensions are considered “built-in” for the purpose of this project and used to create the actual system software. These extensions include if/else constructs, arithmetic operations, simple strings, linear memory allocation, code compilation, etc… The language kernel and its extensions compose the kernel of the operating system and execute in Supervisor Mode.

3.2.1 Cycle Counter

The first extension deals with using the Cycle Counter to measure intervals. Typical use first resets the counter (to avoid wraparound issues), stores a copy of the current counter value, stores a copy of the final counter value after the measured action, and computes the time interval, taking internal overhead into account.

[5] This sounds like a lot to have happen in two cycles, but note that each stack can independently push/pop one item per cycle, while the memory performs a load or store. Only simple one-register-transfer-per-cycle steps are assumed.

3.2.2 Process

The structure of a user process is shown in Figure 5. The process header contains the address of the previous and next process in a ring[6], the Process ID (PID) of this process, the Upper (UB) and Lower (LB) Bounds of memory which the process is allowed to access, and the Entry Point of the process code.

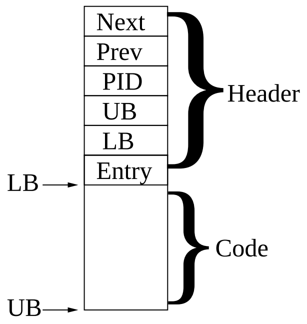

Fig. 5: Structure of a user process

The process extension creates a `current_process` pointer, a sequential PID creation function, functions to allocate a process header and read/write its fields, and process creation functions. The following code outlines the specification of a process:

```
begin_process foo
 : some_function_1 ... ;
 : some_function_2 ... ;
 ...

 enter_process some_function_2
end_process
```

The functions defined inside the process compose the Code section. The UB and LB fields are set to enclose it.

3.2.3 Disk Driver and Page Cache

The disk driver extension defines a single disk block buffer and block read/write functions to fill/flush the buffer. The disk driver can also print the contents of the block buffer to the console.

The page cache extension translates memory addresses to a disk block number and an offset within it. If the block is not already loaded in the buffer, it fills the buffer then returns the integer at the correct offset. Write caching and dirty buffer flushing are not implemented.

[6] This feature is currently unused. All tests were done using a single runnable process.

3.2.4 Trap Handler

The trap handler is the top-level function in the operating system kernel. It defines an entry point for traps, some functions to save and restore process state, system call and trap handling functions, various range checks, and functions to start a process and return to it after a trap.

The trap handler's first action is to identify the instruction causing the trap. This divides the traps into two types: plain traps and syscalls. If the trapping instruction is a load or a store, then the trap handler verifies that the access to the memory location is allowable. If so, the kernel performs the operation and returns to the process. The whole trap is transparent to the process.

If the trapping instruction is Return To User (RTU), then the kernel treats the trap as a syscall. The topmost entry in the Data Stack is verified to be a valid syscall number, then used to index into a table of syscall functions. The kernel performs the syscall, leaves its return value(s) on the Data Stack, and returns to the process.

4 Evaluation

The performance of mode switching is evaluated by measuring the round-trip time for a trivial syscall. Under UNIX-like systems, the simple syscall is `getpid()`, as it consists of only reading a constant field in the header of the current process.

4.1 Syscalls Under Linux

Unfortunately, the nature of syscalls under Linux has changed since about version 2.5 of the kernel, which complicates the measurement. Before Linux 2.5, the i386 syscalls used the 0x80 software interrupt. Under the Pentium IV, this method suffered a strong performance penalty [Hay02] (see [BDF$^+$03] entries in Table 1 for an example) and the syscall mechanism was moved to the `sysenter/sysexit` instruction pair [Bro07][Gar06], specifically designed for fast mode switches. This new syscall code is placed at a fixed address in a read-only page mapped near the end of a process's virtual memory. Furthermore, this page now also holds read-only information such as the current time or the PID of the process, eliminating the syscall outright for these data and reducing the glibc version of `getpid()` to reading a memory location.

Thus three versions of `getpid()` must be measured: reading directly from userspace, performing the syscall via `sysenter`, and via interrupt 0x80. The syscalls were measured by modifying the `lat_syscall null` benchmark from the lmbench-3.0-a7 suite, while the direct read version was measured in a custom program since it is too fast for lmbench to measure[7].

4.2 Stack Machine Syscalls

The implementation of syscalls on the stack machine take advantage of the fact that RTU is a privileged instruction. A user process places the syscall number on the Data

[7] It took 40 *billion* iterations to get a reliable measurement. The loop overhead was verified and negligible.

Stack and executes RTU, causing a trap. The kernel, upon seeing that RTU is the cause of the trap, jumps to the syscall. For `getpid()`, the kernel will leave the PID on the stack before returning to the user process.

4.3 Stack Machine Traps

As an alternative to syscalls on the stack machine, if a process knows the address of its PID, it can try to read it directly. Since the PID is located in the process header, which is outside the memory bounds of the process code, a read attempt will cause a trap. The kernel verifies that the address is within the process's own header and if so, performs the read before returning to the user process.

Similarly, if the address is beyond the installed RAM, then the kernel translates this address to a disk block number and an offset within it, loads the block into a buffer if it isn't already, and reads the value at the block offset before returning to the user process. From the process's point of view, the disk is ordinary RAM, only slower and persistent.

The Linux kernel does not have a similar trap mechanism, except perhaps for page fault handling. There was not enough time to investigate this comparison.

5 Results

The performance of mode switches on the stack machine is compared against that of a Linux 2.6.20.6 kernel running on a 2.2 Ghz[8] Athlon 64 X2. Both machines run in single-user mode. Time is compared as clock cycles to abstract away relative clock speeds and architectural differences[9].

Syscalls and Traps Table 2 compares the stack machine `getpid()` syscall and trap against the three versions of `getpid()` on Linux. The stack machine syscall compares favourably against both Linux syscall mechanisms. The stack machine trap is a little slower due to the necessity of checking the address of the trapping read.

The performance of the stack machine also compares favourably against the systems listed in Table 1. Most notably, the syscall performance exceeds that of Singularity, which performs syscalls without mode switches.

The cost of a stack machine syscall can be better understood by subtracting the time taken by a 'null' syscall which returns immediately without action. This overhead counts for 68 cycles, leaving 13 as actual work done by `getpid()`. This roughly agrees with the measured cycles for the direct read version of Linux `getpid()`.

[8] Precisely: 2211 MHz

[9] This view is based on the assumption that a clock cycle represents a basic register-transfer operation, such as a register move or an addition, regardless of machine type since it's ultimately constrained by physics. This also accounts for multi-cycle pipelined operations.

Linux	μs	cycles	Stack	cycles	Speedup	Stack	cycles	Speedup
int 0x80	0.1494	330	syscall	81	4.07	trap	98	3.37
sysenter	0.0819	181	-	-	2.23	-	-	1.85
direct	0.0047	10	-	-	0.12	-	-	0.10

Tab. 2: Comparison of `getpid()` performance

Disk I/O As further comparison, Table 3 compares the time taken to read one byte[10] from a buffered disk block. The Linux result comes from the lmbench-3.0-a7 `lat_syscall read` benchmark, which repeatedly reads a byte from `/dev/zero`. The stack machine result comes from a user process sequentially reading a number of memory locations mapped to disk blocks. The first read trap causes a kernel buffer to be filled, while subsequent traps return directly from the buffer, which is the number reported here.

Although the result initially seems very favourable, close inspection of the Linux kernel code reveals that the permission checks are done by the Virtual File System (VFS) at each read, and not when the file is opened. This, plus additional size checks, differentiates the operation too much to be directly comparable to the stack machine. However, there does seem to be sufficient room to implement such checks and still remain competitive with Linux.

Note that the stack machine time includes an integer division *in software* to map an address to a disk block and offset[11], while the Linux kernel avoids this overhead altogether since it keeps a position counter for an open file, and would have the use of a hardware divider otherwise.

Linux	μs	cycles	Stack	cycles	Speedup
`read()`	0.2788	616	trap	105	5.87

Tab. 3: Comparison of buffered disk read performance (1 byte)

[10] Actually one integer (4 bytes) for the stack machine, but this being its fundamental addressing unit, should not be a significant difference.

[11] (block number, block offset) = (quotient, remainder) = (read address - disk base address) / block size. Block size is 128 (integers).

6 Discussion

Although the comparison was more complicated than anticipated, I believe that there is sufficient evidence that a stack architecture can improve syscall performance. I find it particularly exciting that the stack syscalls were faster than those in Singularity [HHA+07], without the need for process constraints and compile-time checks.

Despite this improvement, most of the overhead of a syscall or a trap (64 cycles, on top of the base hardware cost of four cycles) is spent shuffling state on the stacks, suggesting that there is room for improvement in the mode switching mechanism.

However, the comparison was made in ideal conditions for both machines. Under a multiprogrammed load, the Linux system would suffer longer mode switches due to conflicts in virtual memory mappings and the consequent TLB flushes [Lie95]. On the other hand, the flat memory of the stack machine would not suffer any such penalty, having instead to do more work when communicating across processes or when allocating memory.

7 Further Work

The initial version of the trap handling code immediately saved all process state to memory, restoring it at the end. This was found to be too time-consuming. The next version was designed such that the state of a trapping process was not saved to memory, but kept on the stacks for the duration of the trap.

Although faster, shuffling the state around the stacks to access buried data and to preserve the value of the Address Register, which is not copied during a trap, is now the source of most of the overhead in the trap handler.

Reduced Trap State In hindsight, the Upper and Lower Bounds registers do not need to be pushed onto the Data Stack during a trap, as the kernel already knows which process was running and thus can fetch these values from the process header when checking memory accesses and before returning to user mode.

This simplification would also allow the trap mechanism to instead save a copy of the Address Register onto the Data Stack, avoiding many operations in the trap handler to temporarily push and pop it from the stacks to preserve its value when the kernel must access memory.

Fewer Trap Sources Another source of overhead in the trap handler is the initial check to see which instruction caused the trap. This is used to distinguish between syscalls, caused by Return To User (RTU), or a memory access trap, causes by loads and stores. Having only a single trap source would eliminate the need to extract the instruction and use it in a table lookup.

Possible candidates are RTU, which would reduce mode switches to syscalls only, or calls and jumps, which would trap when branching to code in the kernel or another process. In the latter case, which flow control instruction causes the trap is irrelevant, and the kernel would only have to alter the memory bounds before returning to user mode. As long as a userspace trapping mechanism exists, either through memory trap or through a privileged instruction, then these changes will not break the virtualization of the stack machine.

Another benefit of not having to check the trapping instruction would be that the trap mechanism could place the saved PC and IR on the Return Stack in the correct order expected by RTU when returning to user mode, eliminating the need to swap them in the trap handler (see 3.1.5, Trap Handling).

Non-Virtual Memory Management Although the stack machine's flat memory model avoids the mode switching overhead of TLB flushes, it also means that processes can never share memory[12] and that new memory cannot be allocated to a process except by memory range extension. This prevents fast Inter-Process Communication (IPC) through page sharing [HHL$^+$97] and the usual `malloc()` and `fork()` behaviour.

However, the fast mode switches made possible by the same stack machine system, along with the fact that the kernel is actually an extensible language kernel, may provide some of the flexibility lost by not having virtual memory.

For example, signalling another process could be accomplished by placing the signal value on the Data Stack and doing a call to the remote process, with the kernel handling the context switch. More complex IPC would require the kernel to copy a buffer across processes.

Similarly, if the normal sequential fetching of instructions is exempted from memory traps, then a process could perform a syscall requesting permission to change its memory bounds in order to operate on remote data. Once done, the process can ask the kernel to restore its usual memory bounds. This change would allow for true shared buffers.

This mechanism should be secure, since calls and branches are still guarded by the memory bounds, and attempts to 'fall through' to hostile code placed after a process can be thwarted by appending a syscall at the tail of a process at creation time, just outside of its normal memory bounds to make it inaccessible to the process.

[12] Unless they were contiguous and had overlapping memory bounds. But that is inflexible and complicates process structure.

References

[BDF$^+$03] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield, *Xen and the art of virtualization*, SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles (New York, NY, USA), ACM, 2003, pp. 164–177.

[Bro07] Andries Brouwer, *Some remarks on the linux kernel*, website, 2007, http://www.win.tue.nl/ aeb/linux/lk/lk.html.

[BSP$^+$95] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers, *Extensibility safety and performance in the spin operating system*, SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles (New York, NY, USA), ACM, 1995, pp. 267–283.

[FHL$^+$96] Bryan Ford, Mike Hibler, Jay Lepreau, Patrick Tullmann, Godmar Back, and Stephen Clawson, *Microkernels meet recursive virtual machines*, Operating Systems Design and Implementation, 1996, pp. 137–151.

[Fox98] Jeff Fox, *F21 CPU*, web page, 1998, http://ultratechnology.com/f21.html.

[Gar06] Manu Garg, *Sysenter based system call mechanism in linux 2.6*, website, 2006, http://manugarg.googlepages.com/systemcallinlinux2_6.html.

[Hay02] Mike Hayward, *Intel p6 vs p7 system call performance*, Linux Kernel Mailing List, December 2002, http://lwn.net/Articles/18412/.

[HHA$^+$07] Galen Hunt, Galen Hunt, Mark Aiken, Manuel Fähndrich, Chris Hawblitzel, Orion Hodson, James Larus, Steven Levi, Bjarne Steensgaard, David Tarditi, and Ted Wobber, *Sealing os processes to improve dependability and safety*, EuroSys '07: Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2007 (New York, NY, USA), ACM, 2007, pp. 341–354.

[HHL$^+$97] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Jean Wolter, and Sebastian Schönberg, *The performance of micro-kernel-based systems*, SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles (New York, NY, USA), ACM, 1997, pp. 66–77.

[HP07] John L. Hennessy and David A. Patterson, *Computer architecture: A quantitative approach*, 4th ed., Morgan Kaufmann Publishers Inc., 2007.

[Koo89] Philip J. Koopman, *Stack computers: the new wave*, Halsted Press, 1989.

[LaF07] Charles Eric LaForest, *Second-generation stack computer architecture*, Independent Studies thesis, University of Waterloo, April 2007.

[Lie95] Jochen Liedtke, *Improved address-space switching on Pentium processors by transparently multiplexing user address spaces*, Arbeitspapiere der GMD No. 933, GMD — German National Research Center for Information Technology, Sankt Augustin, September 1995.

[MT95] Charles H. Moore and C. H. Ting, *MuP21 – a MISC processor*, Forth Dimensions (1995), 41, http://www.ultratechnology.com/mup21.html.

[PG74] Gerald J. Popek and Robert P. Goldberg, *Formal requirements for virtualizable third generation architectures*, Commun. ACM **17** (1974), no. 7, 412–421.