Kernel Mode Linux: Toward an Operating System Protected by a Type Theory

Toshiyuki Maeda and Akinori Yonezawa

University of Tokyo, 7-3-1 Hongo, Bunkyo-ku, Tokyo 113-0033 JAPAN
{tosh, yonezawa}@yl.is.s.u-tokyo.ac.jp

Abstract. Traditional operating systems protect themselves from user programs with a privilege level facility of CPUs. One problem of the protection-by-hardware approach is that system calls become very slow because heavy operations are required to safely switch the privilege levels of user programs. To solve the problem, we design an operating system that protects itself with a type theory. In our approach, user programs are written in a typed assembly language and the kernel performs type-checking before executing the programs. Then, the user programs can be executed in the kernel mode, because the kernel knows that the type-checked programs do not violate safety of the kernel. Thus, system calls become mere function calls and can be invoked very quickly. We implemented Kernel Mode Linux (KML) that realizes our approach. Several benchmarks show effectiveness of KML.

1 Introduction

One problem of traditional operating systems is that system calls are very slow. This is because they protect themselves from user programs by using hardware facilities of CPUs. For example, the Linux kernel [8] for the IA-32 CPUs [7] protects itself by using a memory protection facility integrated with a privilege-level facility of the CPUs. The kernel runs in the kernel mode, the most privileged level, and user programs run in the user mode, the least privileged level. System calls are implemented by using a software interruption mechanism of the CPUs that can raise a privilege level in a safe and restricted way. This software interruption and associated context switches require heavy and complex operations. For example, on the recent Pentium 4 CPU of the IA-32, the software interruption and the context switches are about 132 times slower than an ordinary function call. Recent Linux kernels for the IA-32 CPUs, in fact, use a pair of special instructions, *sysenter* and *sysexit*, for fast invocation of system calls. But this is still 36 times slower than an ordinary function call.

The obvious way to accelerate system calls is to execute user programs in the kernel mode. Then system calls can be handled very quickly because no software interruptions and context switches are needed because user programs can access the kernel directly. However, if we naively execute user programs in the kernel mode, safety of the kernel is totally lost, because the user programs can perform any privileged action in the kernel mode.

In this paper, we propose an approach for protecting an operating system kernel from user programs not with the traditional hardware protection facilities, but with static type-checking. In our approach, user programs are written in a typed assembly language (TAL) [12], which is an ordinary assembly language (except for being typed) that can ensure type safety of programs at the level of machine instructions. Then, the kernel performs type-checking before executing them. If the programs are successfully type-checked, the kernel can safely execute them because the kernel knows that the programs never perform an illegal access to its memory. Moreover, in this paper, we show that the type-checking can ensure safety of the kernel at the same level as the traditional protection-by-hardware approach.

Based on our approach, we implemented an operating system, called Kernel Mode Linux (KML), that can execute user programs in the kernel mode. The notable feature of KML is that user programs are executed as ordinary processes of the original Linux kernel (of course, except for their privilege levels). That is, the memory paging and the scheduling of processes are performed as usual. Therefore, user programs that consume very large memory or enter an infinite loop can be safely executed in the kernel mode. We also conducted several benchmarks on KML and the result shows that system calls are invoked very fast on KML.

The rest of this paper is organized as follows. Sect. 2 formally describes that the hardware protection can be replaced with a type-based static program analysis without losing safety. Sect. 3 describes the implementation of Kernel Mode Linux. Sect. 4 presents the result of some performance benchmarks. Sect. 5 mentions related work. Sect. 6 concludes this paper.

2 Formal Arguments

To show that the protection-by-hardware approach can be replaced with static type-checking, we first define an idealized abstract machine that has a notion of privilege levels of CPUs and system calls. Then, we show that the protection-by-hardware approach of traditional operating systems actually ensures safety of the kernel. Next, we define our typed assembly language for the abstract machine and show that its static type-checking ensures safety of the kernel. The following argument is almost the same line of the arguments of TAL [12, 11], FTAL [5], TALT [3] etc. The fundamental difference between the previous approaches and ours is that our abstract machine has an explicit notion of privilege levels and an operating system kernel.

2.1 Abstract Machine

Figure 1 defines the syntax of states of the abstract machine. The machine state S is defined as a tuple of code memory, data memory, a register file, a program counter, a privilege level and a kernel. The code memory C represents execute-only memory for instructions. The data memory D represents mutable memory

$$
\begin{array}{lll}
(\mathit{Register}) & r ::= & r_0 \mid r_1 \mid \ldots \mid r_{31} \\
(\mathit{Word}) & w ::= & 0 \mid 1 \mid \ldots \mid 2^{32}-1 \\
(\mathit{Program\ counter}) & pc ::= & 0 \mid 1 \mid \ldots \mid 2^{32}-1 \\
(\mathit{Privilege\ level}) & p ::= & \mathit{user} \mid \mathit{kernel} \\
(\mathit{Code\ memory}) & C ::= & \left\{0 \mapsto \iota_0,\ \ldots,\ 2^{30}-1 \mapsto \iota_{2^{30}-1}\right\} \\
 & & (\mathit{Dom}(C) = \{n \mid 0 \leq n \leq 2^{30}-1\}) \\
(\mathit{Data\ memory}) & D ::= & \left\{2^{30} \mapsto w_{2^{30}},\ \ldots,\ 2^{31}-1 \mapsto w_{2^{31}-1}\right\} \\
 & & (\mathit{Dom}(D) = \{n \mid 2^{30} \leq n \leq 2^{31}-1\}) \\
(\mathit{Register\ file}) & R ::= & \{r_0 \mapsto w_0,\ \ldots,\ r_{31} \mapsto w_{31}\} \\
(\mathit{Kernel}) & K ::= & \{w_0 \mapsto \mathit{MetaFunc}_0,\ \ldots\} \\
 & & (\mathit{Dom}(K) \subseteq \{n \mid 2^{31} \leq n \leq 2^{32}-1\}) \\
(\mathit{Instruction}) & \iota ::= & \texttt{add}\ r_{s_1}, r_{s_2}, r_d \mid \texttt{movi}\ w, r_d \mid \texttt{mov}\ r_s, r_d \mid \texttt{jmp}\ r_s \\
 & & \mid \texttt{blt}\ r_{s_1}, r_{s_2}, r_{s_3} \mid \texttt{ld}\ w[r_s], r_d \mid \texttt{st}\ r_s, w[r_d] \mid \texttt{illegal} \\
(\mathit{State}) & S ::= & (C, D, R, pc, p, K) \mid \texttt{user\_error} \mid \texttt{kernel\_error}
\end{array}
$$

Fig. 1. Syntax of the abstract machine states

for data. The register file R is defined as a map from registers to word values. The program counter pc is a word value that points to an instruction that is about to be executed. The privilege level p consists of two values: user and kernel. The value user represents that the abstract machine runs in the user mode and the value kernel represents that it runs in the kernel mode. The kernel K represents an operating system kernel. It is defined as a map from addresses represented as word values to meta functions ($\mathit{MetaFunc}$) that translate a machine state to another. The meta functions can be viewed as inner functions of the kernel that implement system calls. In a real machine, the meta functions are only sequences of instructions. In this paper, however, we do not care the real representation because they are regarded as a trusted computing base in practice and there are not significant points in the representation. In addition, there are two special machine states that represent an error state: `user_error` and `kernel_error`. `user_error` represents an error only for user programs that does not affect safety or integrity of the whole system. On the other hand, `kernel_error` represents a fatal error that may crash the whole system.

Figure 2 defines the operational semantics of the abstract machine. They are defined as a conventional small-step function that translates a machine state S to another S'. If the program counter pc points into the domain of the code memory C, then an instruction, $C(pc)$, is executed as usual. The branch instruction `jmp` and `blt` can be viewed as special instructions for invocation of system calls if their target addresses are in the domain of the kernel. If the memory access instructions (`ld` and `st`) access the illegal memory, that is, the code memory or the kernel, then a machine state evaluates to `user_error` (if $p = \mathit{user}$) or `kernel_error` (if $p = \mathit{kernel}$). If the program counter pc points into the domain of the kernel K, a system call is executed, that is, S' becomes a machine state that is obtained by applying S to a meta function which is pointed by pc.

$$(C, D, R, pc, p, K) \;\mapsto\; S'$$

$$\frac{\text{If } pc \notin Dom(K) \cup Dom(C)}{} \qquad S' = error(p)$$

$$\frac{\text{If } pc \in Dom(K)}{} \qquad S' = K(pc)(S)$$

$$\frac{\text{If } pc \in Dom(C)\text{:}}{}$$

if $C(pc) =$	then $S' =$	
`add` r_{s_1}, r_{s_2}, r_d	$(D, R', pc+1)$	$R' = R\{r_d \mapsto R(r_{s_1}) + R(r_{s_2})\}$
`movi` w, r_d	$(D, R', pc+1)$	$R' = R\{r_d \mapsto w\}$
`mov` r_s, r_d	$(D, R', pc+1)$	$R' = R\{r_d \mapsto R(r_s)\}$
`jmp` r_s	$(D, R, R(r_s))$	
`blt` $r_{s_1}, r_{s_2}, r_{s_3}$	$(D, R, R(r_{s_3}))$	when $R(r_{s_1}) < R(r_{s_2})$
	$(D, R, pc+1)$	when $R(r_{s_1}) \geq R(r_{s_2})$
`ld` $w[r_s], r_d$	$(D, R', pc+1)$	$R' = R\{r_d \mapsto D(R(r_s) + w)\}$ when $R(r_s) + w \in Dom(D)$
	$error(p)$	when $R(r_s) + w \notin Dom(D)$
`st` $r_s, w[r_d]$	$(D', R, pc+1)$	$D' = D\{R(r_d) + w \mapsto R(r_s)\}$ when $R(r_d) + w \in Dom(D)$
	$error(p)$	when $R(r_d) + w \notin Dom(D)$
`illegal`	`user_error`	

where $error(p) =$ `kernel_error` when $p = kernel$
`user_error` when $p = user$

C, p and K are omitted in the above table because they never change.

Fig. 2. Operational semantics of the abstract machine

2.2 Safety of the Traditional Protection-by-Hardware Approach

The traditional protection-by-hardware approach can be expressed simply in the abstract machine. It only sets the privilege levels of the machine states to *user* to prevent `kernel_error` from occurring.

Theorem 1 (Safety of the Traditional Protection-by-Hardware). *If meta functions of K never alter privilege levels and never translate machine states to* `kernel_error`*, then any machine state of the form $(C, D, R, pc, user, K)$ never evaluates to* `kernel_error`*.*

Proof. Straightforward from the operational semantics of the abstract machine. □

2.3 Typed Assembly Language

Now, we define a TAL that prevents `kernel_error` from occurring. Figure 3 shows the syntax of the typed assembly language for the abstract machine. (It only shows the difference from the syntax of the abstract machine.)

Our TAL has 6 kinds of basic types: α (type variable), *int* (word value), $\langle \tau_1, \ldots, \tau_n \rangle$ (tuple), $\forall[\Delta].\Gamma$, $sizeof(\alpha)$, and $sizeof(\langle \tau_1, \ldots, \tau_n \rangle)$. $\forall[\Delta].\Gamma$ is a type of instruction sequences. For example, if an instruction sequence I has

$$
\begin{array}{lr}
(\textit{type}) & \tau ::= \alpha \mid \mathit{int} \mid \forall[\Delta].\Gamma \mid \langle \tau_1, \ldots, \tau_n \rangle \\
 & \quad \mid \mathit{sizeof}(\langle \tau_1, \ldots, \tau_n \rangle) \mid \mathit{sizeof}(\alpha) \\
(\textit{type variables}) & \Delta ::= \alpha_1, \ldots, \alpha_n \\
(\textit{code memory type}) & \Psi_C ::= \{w_0 : \forall[\Delta_0].\Gamma_0, \ldots, w_n : \forall[\Delta_n].\Gamma_n\} \\
(\textit{data memory type}) & \Psi_D ::= \{w_0 : \langle \tau_0, \ldots \rangle, \ldots, w_n : \langle \tau_n, \ldots \rangle\} \\
(\textit{kernel type}) & \Psi_K ::= \{w_0 : \forall[\Delta_0].\Gamma_0, \ldots, w_n : \forall[\Delta_n].\Gamma_n\} \\
(\textit{register file type}) & \Gamma ::= \{r_0 : \tau_0, \ldots, r_{31} : \tau_{31}\} \\
(\textit{tuple}) & T ::= \langle w_1, \ldots, w_n \rangle \\
(\textit{instruction sequence}) & I ::= \iota_1; \ldots; \iota_n
\end{array}
$$

Fig. 3. Syntax of the typed assembly language for the abstract machine. Only the difference from the syntax of the abstract machine is shown

a type $\forall[\Delta]\Gamma$, then the register file of the abstract machine must have a register file type (explained below) represented by Γ, to jump to and execute the instruction sequence. $\mathit{sizeof}(\langle \tau_1, \ldots, \tau_n \rangle)$ is a type of a word value that represents a size of a tuple of the type $\langle \tau_1, \ldots, \tau_n \rangle$. For example, if a word value w has a type $\mathit{sizeof}(\langle \mathit{int}, \mathit{int}, \mathit{int} \rangle)$, then we know that $w = 3$. $\mathit{sizeof}(\alpha)$ is a type of a word value that represents a size of a tuple of the type α. Our TAL also has 4 kinds of types for the components of the abstract machine. Ψ_C is a type of code memory, Ψ_D is a type of data memory, Γ is a type of a register file, and Ψ_K is a type of a kernel and corresponds to the interface of system calls of the kernel. Although our TAL does not have rich types, we can extend it with them, such as existential types [12], recursive types [5], array types [10] and stack types [11], in theory.

The dynamic semantics of our TAL are unchanged from the abstract machine. This indicates that we can erase type information of programs and no type-checking is required at runtime.

Figure 4 presents static judgments of our TAL that assert the well-formedness of the components of the TAL. To have a well-formed machine state, the code memory, the data memory, the kernel, the register file and the instruction sequence that starts from the address of the program counter must be well-formed. The static semantics for the well-formedness are presented in Appendix A. Because of space limitations, the rules for word values, tuples and instructions are omitted. They are basically the same line of the rules of FTAL [5] etc.

The well-formedness of the kernel is defined as follows. The basic idea is that system calls never break the well-formedness of machine states.

Definition 1 (The Well-formedness of the Kernel). *The kernel K is well-formed, denoted as $\Psi_K \vdash K$, if:*

1. *Ψ_K is well-formed, that is, $\vdash \Psi_K$.*
2. *The domain of K is equal to the domain of Ψ_K.*
3. *For all w in the domain of K, if $\Psi_K \vdash S$ where $S = (C, D, R, w, p, K)$, then $K(w)(S) \neq$ `user_error` or `kernel_error` and $\Psi_K \vdash K(w)(S)$.*

Judgment	Meaning
$\Delta \vdash \tau$	τ is a well-formed type
$\vdash \Psi_C$	Ψ_C is a well-formed code memory type
$\vdash \Psi_D$	Ψ_D is a well-formed data memory type
$\vdash \Psi_K$	Ψ_K is a well-formed kernel type
$\Delta \vdash \Gamma$	Γ is a well-formed register file type
$\vdash (C, D) : \Psi$	C is well-formed code memory of type Ψ_C
	D is well-formed data memory of type Ψ_D
$\Psi \vdash R : \Gamma$	R is a well-formed register file of type Γ
$\Psi, \Delta \vdash w : \tau$	w is a well-formed word value of type τ
$\Psi \vdash T : \tau$	T is a well-formed tuple of type τ
$\Psi, \Delta, \Gamma \vdash I$	I is a well-formed instruction sequence
$\Psi_K \vdash K$	K is a well-formed kernel
$\Psi_K \vdash S$	S is a well-formed machine state

where $\Psi = (\Psi_c, \Psi_d, \Psi_k)$

Fig. 4. Static judgments of the typed assembly language

2.4 Safety of the Typed Assembly Language

Now, we show that, if a machine state S and a kernel K are well-formed, then S never evaluates to `kernel_error`. For that purpose, we first show that our typed assembly language satisfies the following usual Preservation and Progress lemmas.

Lemma 1 (Preservation). *If $\Psi_K \vdash K$, $\Psi_K \vdash S$ and $S \mapsto S'$, then $\Psi_K \vdash S'$.*

Lemma 2 (Progress). *If $\Psi_K \vdash K$ and $\Psi_K \vdash S$, then there exists S' such that $S \mapsto S'$.*

Proof. By case analysis on $C(pc)$ (if $pc \in Dom(C)$) and $K(pc)$ (if $pc \in Dom(K)$), and induction on the typing derivations. Here, we only show the proof for the case that $pc \in Dom(K)$.

If $pc \in K$, there exists $S' = K(pc)(S)$ such that $S \mapsto S'$. Thus, the progress lemma is satisfied. In addition, from the well-formedness of the kernel $\Psi_K \vdash K$, we have $\Psi_K \vdash S'$. Thus, the preservation lemma is also satisfied. □

From these two lemmas, we can prove that, if a user program can be type-checked, that is, if a machine state that represents the user program is well-formed, then the program never violates safety of the kernel from the fact that the machine state never evaluates to `kernel_error`, as long as the kernel is correctly implemented. Thus, we can safely replace the hardware protection facilities (the privilege levels) with the static type-checking of our TAL.

Theorem 2 (Safety of the Typed Assembly Language). *If $\Psi_K \vdash K$ and $\Psi_K \vdash S$, then S never evaluates to `kernel_error`.*

Proof. Straightforward from Lemma 1 and Lemma 2.

2.5 Dynamic Memory Allocation

In our framework, a dynamic memory allocation mechanism can be realized by having the kernel include it as a system call, instead of having a special macro instruction as Morrisett's TAL [12]. For example, we can use the following kernel.

Example 1 (The Kernel with the Malloc System Call).

$$\Psi_K = \{\ w_{malloc}\ :\ \forall[\alpha].\{r_0 : sizeof(\alpha), r_1 : \alpha, \\ r_{31} : \forall[\beta].\{r_0 : \alpha, r_1 : \alpha, r_{31} : \beta\}\}\}$$
$$K\ = \{\ w_{malloc} \mapsto Malloc\ \}$$

(Registers other than r_0, r_1 and r_{31} are omitted here for the sake of brevity.)

The meta function $Malloc$ allocates unused memory of the size specified by r_0 from the data memory D. Then, it initializes the allocated memory with the contents of the memory specified by r_1 and sets the register r_0 to the address of the allocated memory. Finally, it jumps to the return address specified in the register r_{31}.

3 Kernel Mode Linux

Based on the argument of the previous section, we implemented Kernel Mode Linux (KML), a modified Linux kernel for IA-32 CPUs which can execute user programs in the kernel mode. In this section, we describe the implementation of Kernel Mode Linux (KML).

3.1 How to Execute User Processes in the Kernel Mode

In IA-32 CPUs, the privilege level of a running program is determined by the privilege level of the code segment in which the program is executed. A program counter of IA-32 CPUs consists of the CS segment register, which specifies a code segment, and the EIP register, which specifies an offset into the code segment.

To execute a user process in the kernel mode, the only thing KML does is to set the CS register of the process to the kernel code segment, the most-privileged segment, instead of the user code segment, the least-privileged segment. Then the process is executed in the kernel mode. We call such processes as "kernel-mode user processes".

Because of this simple approach of KML, a kernel-mode user process can be an ordinary user process (except for its privilege level). Therefore, even if a kernel-mode user process consumes huge amount of memory and/or enters an infinite loop, the kernel can reclaim the memory through its paging facility and suspend the process through its process-scheduling facility, because KML does not modify any code or data of the facilities.

3.2 How to Invoke System Calls from Kernel-Mode User Processes

To ensure safety of the kernel, user programs that will be executed in the kernel mode must be written in TALx86 [10], a notable typed assembly language implementation for IA-32 CPUs, or Popcorn [10], a safe dialect of the C programming language which can be translated to TALx86.

In the current KML, the interface of the system calls (that is, a kernel type Ψ_K mentioned in Sect. 2) is exported as TAL's interface file. The interface file contains pairs of a name of a system call and its TAL type. The actual address of a system call is looked up from the name of the system call at link time, as usual. Then, programmers write their programs in TALx86 or Popcorn according to the interface file. Invocations of system calls are written as usual function calls, as expected. Though almost all system calls can be exported to user programs safely, there are a few exceptions that cannot be exported directly because they may violate the well-formedness of the kernel (e.g., `mmap`/`munmap` system calls).

Actually, KML must type-check user programs just before executing them for ensuring safety. However, the current KML itself does not perform type-checking, because the current TALx86 implementation cannot type-check executable binaries (though intermediate relocatable binary objects can be checked). Thus, the current KML needs to trust the external TALx86 assembler. We plan to develop our own TAL type-checker for checking the executable binaries to solve this problem.

3.3 Executing Existing Applications in the Kernel Mode

The current KML has a loophole mechanism to execute existing applications in the kernel mode and eliminate the overhead of system calls without modifying them. Though safety is not ensured, the mechanism is useful for measuring the performance improvement of applications due to the elimination of the overhead of system calls.

To implement the loophole mechanism, we exploit the facility of the recent original Linux kernel for multiplexing a system call invocation into a traditional software interruption (*int* 0x80) or *sysenter*/*sysexit* instructions depending on a kind of CPU. We implemented a third branch for the multiplexer that invokes system calls with direct function calls. Because the multiplexer executes additional instructions to invoke system calls, a system call invocation with this mechanism is not optimal. However, it is sufficiently fast compared to the software interruptions and the *sysenter*/*sysexit* instructions.

3.4 The Stack Starvation Problem

As described in Section 3.1, the basic approach of KML is quite simple. However, there is one problem we call *stack starvation*. In the original Linux kernel, interrupts are handled by interrupt handling routines specified in the Interrupt Descriptor Table (IDT). When an interrupt occurs, an IA-32 CPU stops execution of the running program, saves its execution context and executes the interrupt handling routine.

How the IA-32 CPU saves the execution context of the running program at interrupts depends on the privilege level of the program. If the program is executed in the user mode, the IA-32 CPU automatically switches its memory stack to a kernel stack. Then, it saves the execution context (EIP, CS, EFLAGS, ESP and SS registers) to the kernel stack. On the other hand, if the program is executed in the kernel mode, the IA-32 CPU does not switch its memory stack and saves the context (EIP, CS and EFLAGS registers) to the memory stack of the running program.

What happens if a user process that is executed in the kernel mode on KML accesses its memory stack, which is not mapped by page tables of a CPU? First, a page fault occurs, and the CPU tries to interrupt the user process and jump to a page fault handler specified in the IDT. However, the CPU cannot accomplish this work, because there is no stack for saving the execution context. Because the process is executed in the kernel mode, the CPU never switches the memory stack to the kernel stack. To signal this fatal situation, the CPU tries to generate a special exception, a double fault. However, again, the CPU cannot generate the double fault because there is no stack for saving the execution context of the running process. Finally, the CPU gives up and resets itself. We call this problem *stack starvation*.

To solve the *stack starvation* problem, KML exploits the task management facility of IA-32 CPUs. The IA-32 task management facility is provided to support process management for kernels. Using this facility, a kernel can switch processes with only one instruction. However, today's kernels do not use this facility because it is slower than software-only approaches. Thus the facility is almost forgotten by all.

The strength of this task management facility is that it can be used to handle exceptions and interrupts. Tasks managed by an IA-32 CPU can be set to the IDT. If an interrupt occurs and a task is assigned to handle the interrupt, the CPU first saves the execution context of the interrupted program to a special memory region (called a task state segment, or TSS) of the running task, instead of to the memory stacks. Then, the CPU switches to the task specified in the IDT to handle the interrupt. The most important point is that there is no need to switch a memory stack if the task management facility is used to handle interrupts. That is, if we handle page fault exceptions with the facility, a user process executed in the kernel mode can access its memory stack safely.

However, if we handle all page faults with the task management facility, the performance of the whole system degrades because the task-based interrupt handling is slower than the ordinary interrupt handling. Therefore, in KML, only double fault exceptions are handled with the task management facility. That is, only page faults caused by memory stack absence are handled by the facility. Thus, the performance degradation is very small and negligible because memory stacks rarely cause page faults.

4 Benchmarks

To measure the degree of performance improvement by executing user programs in the kernel mode, we conducted two benchmarks that compared performance of the original Linux kernel with that of KML. In each benchmark, we executed exactly the same benchmark program both on the original Linux kernel and KML with the system call multiplexing mechanism described in the previous section. We also compared KML with the *sysenter/sysexit* mechanism. The experimental environment is shown in Table 1.

Table 1. Experimental environment

CPU	Pentium 4 3.000GHz (L2 cache 512KB)
Memory	1GB (PC3200 DDR SDRAM)
Hard disk	120GB
OS	Linux kernel 2.5.72 (KML_2.5.72_001)

4.1 First Benchmark: Latencies of System Calls

The first benchmark measured latencies of 5 system calls by using LMbench [9] (version 2.0). `getpid` is the simplest system call that only obtains a process ID. Therefore, the overhead of system calls becomes very large in it. `read` and `write` are basic I/O system calls. In the benchmark, `read` and `write` were performed on the null device (i.e., /dev/null). `stat` and `fstat` are system calls for obtaining file statistics. The result is presented in Table 2.

Table 2. Latencies of system calls. "Original" means the original Linux kernel and "sysenter" means the original Linux kernel using *sysenter/sysexit*

	`getpid`	`read`	`write`	`stat`	`fstat`
Original	371.3	439.3	402.3	1157.3	608.7
sysenter	135.1	201.1	164.9	896.5	383.5
KML	16.9	91.0	53.4	756.2	204.1

(Unit: nanoseconds)

The result shows that the `getpid` system call was about 22 times faster in KML than in the original Linux kernel. It also shows that it was about 8 times faster in KML than using *sysenter/sysexit*. The latencies of the other system calls were also improved in KML.

4.2 Second Benchmark: Throughputs of File I/O Operations

The second benchmark examined how performance of file I/O is improved in KML by using IOzone [13] (version 3.172). In the benchmark, we measured throughputs of 4 I/O operations in 4 tests: `Write`, `Re-write`, `Read` and `Re-read`. The `Write` test measures the throughput of writing a new file. The `Re-write` test measures the throughput of writing a file that already exists. The `Read` test measures the throughput of reading an existing file. The `Re-read` test measures the performance of reading a file that is recently read. In all of the 4 tests, we measured the throughput of the I/O operations on files whose size is from 16 Kbytes to 512 Kbytes. We fixed the buffer size to 16 Kbytes. The benchmark was performed on a file system of Ext3fs. The result is shown in Figure 5.

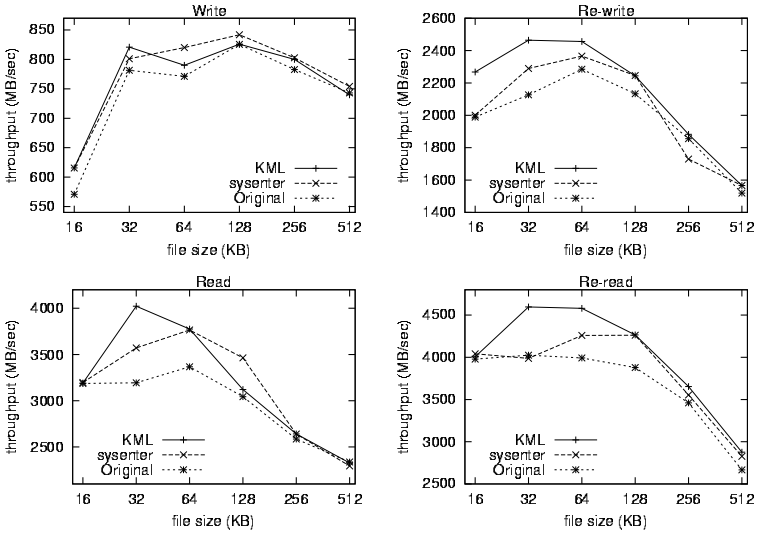

Fig. 5. Throughputs of I/O operations. "Original" means the original Linux kernel and "sysenter" means the original Linux kernel using *sysenter/sysexit*

The result indicates that KML can improve various I/O operations for files of small size. The result shows that, compared to the original Linux kernel, the throughputs of `Write`, `Re-write`, `Read` and `Re-read` were improved up to 8 %, 15.9 %, 25.9 % and 14.7 % respectively. Compared to the *sysenter/sysexit* mechanism, the throughputs of `Write`, `Re-write`, `Read` and `Re-read` were improved up to 2 %, 13.3 %, 12.6 % and 15.3 % respectively. There is performance degradation in some cases (especially, in the `Write` test). This is mainly due to CPU cache effects.

5 Related Work

In the field of operating system and programming language research, there are several works related to safe execution of user programs in a kernel. An interesting difference between our research and the previous work lies in their objective. Our objective is to execute ordinary user programs in the kernel mode safely while the work below is concentrated on how to extend a kernel safely.

5.1 SPIN Operating System

SPIN [2] is an extensible kernel that ensures safety by a language-based protection. In SPIN, kernel extensions are written in the Modula-3 programming language [6]. Safety of SPIN is ensured by the fact that Modula-3 is a type-safe language. That is, programmers cannot write malicious kernel extensions.

This approach of SPIN has two problems. The first problem is that its trusted computing base (TCB) becomes large because the kernel must trust external compilers of Modula-3. In SPIN, the kernel cannot check safety of binary codes. In our approach, on the other hand, TCB is smaller than SPIN because safety is checked at the machine language level and we need not trust external compilers. The second problem is that the kernel extensions must be written in Modula-3. In our approach, we can write user programs in various programming languages, if there exist compilers that translate the languages to TALs.

5.2 Software-Based Fault Isolation

Software-based Fault Isolation (SFI) [16] is a technique that modifies binary codes of applications to ensure memory and control flow safety. In the SFI approach, check codes are inserted before each memory access and jump instruction of untrusted programs to ensure safety.

The problem of the SFI approach is its large overhead of the inserted runtime safety check codes. In our approach, on the other hand, safety can be mostly ensured at load time through type-checking.

5.3 Foundational Proof-Carrying Code

In the Foundational Proof-Carrying Code (FPCC) [1] approach, a user program is attached a logical proof of its safety, and the proof is verified before executing the program.

There are two advantages in the FPCC approach, compared to the simple TAL [12]. First, TCB becomes very small. The TCB of FPCC consists of a proof-checker, a machine specification that represents a behavior of a CPU and memory, and a safety policy. On the other hand, the TCB of the simple TAL system becomes larger because it includes a TAL type-checker. To solve this problem, we can take the approach of Hamid et al [5]. They showed that their TAL that is carefully defined so that well-formed TAL programs are mapped

to valid machine states of FPCC can be syntactically translated to a FPCC program that does not violate memory safety and control flow safety. In their approach, the TCB can be as small as the FPCC approach.

Second, a safety policy can be flexible. In the FPCC approach, a safety policy is specified in logics of a proof-checker. Therefore, we can specify a safety policy that cannot be expressed in a simple TAL type system. For example, a limitation of memory or CPU usage can be ensured by the FPCC approach. However, there is a drawback of this flexibility of a safety policy: Proof generation may become very hard. In TALs, on the other hand, type-checking is simple and easy. In addition, to replace the hardware protection mechanisms, type safety suffices because they ensure only memory safety and control flow safety.

6 Conclusion

In this paper, we showed that the hardware protection mechanisms that traditional operating systems exploit can be safely replaced with static program analysis, mainly type-checking. By discarding the hardware protection mechanisms, the overhead of switching a privilege level of a CPU can be eliminated and efficiency of applications can be improved. Based on this approach, we developed KML, an operating system in which user programs can be executed in the kernel mode of a CPU (available at http://www.taplas.org/~tosh/kml). The result of several benchmarks shows effectiveness of KML.

7 Future Work

Although the current KML is effective for improving performance of programs, there should be a limitation because it only eliminates the overhead of system calls. To improve the performance further, we should modify the kernel and its interface and exploit the TAL type system more aggressively. For example, we think that a kernel can be modified to export network communication hardware to user programs as user-level communication technologies (e.g., [14, 15, 4]). The problem of the user-level communication is a tradeoff between performance and safety. To achieve high performance, the kernel must export network hardware to user programs directly and give up its safety because the user programs can access the kernel directly. To achieve safety, on the other hand, the kernel must encapsulate network hardware by system calls and give up high-performance communication. By using our approach, we can achieve both high-performance communication and safety because the overhead of system calls can be eliminated without losing safety.

As other directions, our approach can be applied to microkernels. Traditional microkernels have a problem of the overhead of communication between a kernel and user servers. By applying our approach, the overhead can be reduced largely. In addition, we think that the large part of a kernel itself can be written in a strongly typed low-level language as TALs. Of course, it is difficult to ensure total safety of the kernel. We think, however, that the simple memory and control flow

safety is still valuable. For example, consider synchronization primitives such as mutex locks and semaphores. It is difficult to ensure that these primitives are properly used for preventing deadlocks. However, it is very easy to ensure memory safety, because they only make a decent memory access.

References

1. A. W. Appel. Foundational proof-carrying code. In *In proc. of 16th Annual IEEE Symposium on Logic in Computer Science*, pages 247–258, June 2001.
2. B. N. Bershad, C. Chambers, S. J. Eggers, C. Maeda, D. McNamee, P. Pardyak, S. Savage, and E. G. Sirer. SPIN - an extensible microkernel for application-specific operating system services. In *In proc. of ACM SIGOPS European Workshop*, pages 68–71, September 1994.
3. K. Crary. Toward a foundational typed assembly language. In *Proc. of Symposium on Principles of Programming Languages*, pages 198–212, January 2003.
4. C. Dubnicki, A. Bilas, Y. Chen, S. Damianakis, and K. Li. VMMC-2: efficient support for reliable, connection-oriented communication. In *Proc. of Hot Interconnects*, pages 37–46, August 1997.
5. N. Hamid, Z. Shao, V. Trifonov, S. Monnier, and Z. Ni. A syntactic approach to foundational proof-carrying code. Technical Report YALEU/DCS/TR-1224, Dept. of Computer Science, Yale University, 2002.
6. S. P. Harbison. *Modula-3*. Prentice Hall, 1992.
7. Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual*.
8. The Linux kernel. http://www.kernel.org.
9. L. W. McVoy and C. Staelin. lmbench: Portable tools for performance analysis. In *Proc. of USENIX Annual Technical Conference*, pages 279–294, 1996.
10. G. Morrisett, K. Crary, N. Glew, D. Grossman, R. Samuels, F. Smith, D. Walker, S. Weirich, and S. Zdancewic. TALx86: A realistic typed assembly language. In *Proc. of ACM SIGPLAN Workshop on Compiler Support for System Software*, pages 25–35, 1999.
11. G. Morrisett, K. Crary, N. Glew, and D. Walker. Stack-based typed assembly language. In *In proc. of Types in Compilation*, pages 28–52, 1998.
12. G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, 1999.
13. W. D. Norcott. The IOzone file system benchmark. http://www.iozone.org.
14. L. Prylli and B. Tourancheau. BIP: a new protocol designed for high performance networking on Myrinet, March 1998.
15. H. Tezuka, A. Hori, and Y. Ishikawa. PM : a high-performance communication library for multi-user parallel environments. Technical Report TR-96015, Real World Computing Partnership, 1996.
16. R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. *ACM SIGOPS Operating Systems Review*, 27(5):203–216, December 1993.

A The Static Semantics of Our Typed Assembly Language (Rules for Types and Machine States Only)

$$\frac{FTV(\tau) \subseteq \Delta}{\Delta \vdash \tau} \quad \text{(TYPE)}$$

$$\frac{\begin{array}{c}\forall w_i \in Dom(\Psi_C).\ w_i \in [0, 2^{30}-1] \\ \text{and } \Psi_C(w_i) = \forall[\Delta].\Gamma \text{ and } \Delta \vdash \Gamma\end{array}}{\vdash \Psi_C} \quad \text{(CODE MEMORY TYPE)}$$

$$\frac{\begin{array}{c}\forall w_i \in Dom(\Psi_K).\ w_i \in [2^{31}, 2^{32}-1] \\ \text{and } \Psi_K(w_i) = \forall[\Delta].\Gamma \text{ and } \Delta \vdash \Gamma\end{array}}{\vdash \Psi_K} \quad \text{(KERNEL TYPE)}$$

$$\frac{\begin{array}{c}\forall w_i \in Dom(\Psi_D).\ TupleRange(\Psi_D, w) \subseteq [2^{30}, 2^{31}-1] \\ \text{and } \Psi_D(w_i) = \langle \tau_1, \ldots, \tau_n \rangle \text{ and } \cdot \vdash \Psi_D(w_i) \\ \text{and } \forall w_j \in Dom(\Psi_D) \text{ s.t. } w_j \neq w_i. \\ TupleRange(\Psi_D, w_i) \text{ and } TupleRange(\Psi_D, w_j) \text{ do not overlap}\end{array}}{\vdash \Psi_D} \quad \text{(DATA MEMORY TYPE)}$$

$$\frac{\forall r_i \in Dom(\Gamma).\Delta \vdash \Gamma(r_i)}{\Delta \vdash \Gamma} \quad \text{(REGISTER FILE TYPE)}$$

where
$TupleRange(\Psi_D, w) = [w, w+n-1]$ where $\Psi_D(w) = \langle \tau_1, \ldots, \tau_n \rangle$

Fig. 6. The static semantics of the typed assembly language: types

$$\frac{\begin{array}{c}\vdash (C, D) : \Psi \qquad \Psi \vdash R : \Gamma \\ \text{(1) If } pc \in C, \quad \Psi, \cdot, \Gamma \vdash InstrDec(C, pc) \\ \textit{or} \\ \text{(2) If } pc \in K, \quad \Psi_K(pc) = \forall[\Delta'].\Gamma' \qquad \cdot \vdash \tau_i \qquad [\tau_1, \ldots, \tau_n/\Delta']\Gamma' = \Gamma\end{array}}{\Psi_K \vdash (C, D, R, pc, p, K)} \quad \text{(STATE)}$$

$$\frac{\begin{array}{c}\vdash \Psi_C \qquad \vdash \Psi_D \\ \forall w_i \in Dom(\Psi_C).\Psi, \Delta_i, \Gamma_i \vdash InstrDec(C, w_i) \text{ where } \Psi_C(w_i) = \forall[\Delta_i].\Gamma_i \\ \forall w_i \in Dom(\Psi_D).\Psi \vdash TupleDec(D, w_i, n) : \Psi_D(w_i) \text{ where } \Psi_D(w_i) = \langle \tau_1, \ldots, \tau_n \rangle\end{array}}{\vdash (C, D) : \Psi} \quad \text{(MEMORY)}$$

$$\frac{\forall r_i \in Dom(\Gamma).\Psi, \cdot, \cdot \vdash R(r_i) : \Gamma(r_i)}{\Psi \vdash R : \Gamma} \quad \text{(REGISTER FILE)}$$

where
$InstrDec(C, w) = \iota_w, \ldots, \iota_{2^{30}-1}$ where $\iota_i = C(i)$
$TupleDec(D, w, n) = \langle w_0, \ldots, w_{n-1} \rangle$ where $w_i = D(w+i)$

Fig. 7. The static semantics of the typed assembly language: state, memory and register file