SAFE EXECUTION OF USER PROGRAMS IN KERNEL MODE USING TYPED ASSEMBLY LANGUAGE

by

Toshiyuki Maeda

A Master Thesis

Submitted to
The Graduate School of
The University of Tokyo
on February 5, 2002
in Partial Fulfillment of the Requirements
for the Degree of Master of Science
in Information Science

Thesis Supervisor: Akinori Yonezawa
Professor of Information Science

ABSTRACT

In traditional operating systems, user programs suffer from the overhead of system calls because of transitions between the user mode and the kernel mode across their protection boundary. However, this overhead can be eliminated if the user programs can be executed *safely* inside the kernel mode. We achieve this effect by developing a safe kernel mode execution mechanism using *TAL*, Typed Assembly Language.

TAL is an assembly language which ensures memory safety and control flow safety of machine code through a type system. Memory safety means that a program accesses only memory which the program is permitted to access, while control flow safety means that a program jumps to only valid code which the program is permitted to execute. This memory and control flow safety are verified through a type checker using type annotations attached to machine code by the assembler of TAL.

In our approach, user programs are written in TAL and their safety are verified through the type checker of TAL *before* they are executed in the kernel mode. Thus, user programs can be executed in the kernel mode both safely and efficiently, because their safety is verified before execution and there is little overhead of runtime checks. Moreover, unlike other approaches to safe kernel mode execution—such as the SPIN operating system and PCC (Proof-Carrying Code)—our approach neither depends on a specific high-level programming language and its compiler, nor requires expensive calculation of complex proofs.

We implemented a prototype system based on our approach by modifying the Linux Kernel. This prototype system uses original system call functions of the Linux kernel as its interface to user programs, and achieves the *same* degree of safety (e.g., about access control of files) while eliminating the overhead of system calls only. For the purpose of performance evaluation, a TAL version of the "find" program, which traverses directory trees of a file system, is implemented on our prototype system and found to run 14 % faster in the kernel mode than in the user mode. Also, a TAL version of the "echod" program, which receives data from a client and sends it back to the client, is executed and its latency is improved 4 μs in the kernel mode.

Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank Prof. Akinori Yonezawa for his useful comments and suggestions. I show my special thanks to Eijiro Sumii who gave me essential ideas of the research and encouraged me. Finally I wish to thank the members of Yonezawa Laboratory for their useful discussions and comments.

Chapter 1

Introduction

In traditional operating systems, a kernel is protected from user programs by a privilege level facility and an MMU of a CPU. First, user programs and a kernel are separated in different privilege levels of a CPU: the user programs are in the user mode, which is the least privileged level, and the kernel is in the kernel mode, which is the most privileged level. Then, the kernel is mapped in memory as a privileged region which only programs in the kernel mode can access, by a page table of an MMU in the CPU. Thus, the user programs cannot access the kernel.

A system call is a function that is called by user programs to invoke a service of the protected kernel. In traditional operating systems, the system call checks the arguments given by the user programs, builds a data structure to convey the arguments to the kernel, and executes a special instruction called a software interrupt. Then, the interrupt hardware of a CPU saves the state of the user programs and switches the privilege level to the kernel mode. Finally, it dispatches to the inner function that implements the system call. Thus, in traditional operating systems, user programs suffer from the overhead of system calls because they need the costly software interruptions and context switches.

If user programs are executed in the kernel mode, the overhead of system calls can be eliminated because it is unnecessary to transit between the kernel mode and the user mode. However, it is dangerous to simply let user programs execute in the kernel mode because programs can operate the whole system without restriction.

This paper shows that user programs can be executed in the kernel mode safely by using Typed Assembly Language(TAL) [MWCG98, MCG+99].

TAL is an assembly language which ensures memory safety and control flow safety of programs through its type check. The memory safety of a program means that the

program accesses only memory which is permitted to access. The control flow safety of a program means that the program executes only instructions which are permitted to execute. In traditional operating systems, the memory and control flow safety are ensured by using an MMU and a privilege level facility of a CPU.

In our approach, user programs are written in TAL and their safety is verified at load time, that is, before execution. Thus, the user programs are executed in the kernel mode safely and efficiently because runtime safety check is almost unnecessary.

A prototype system was implemented based on our approach by modifying the Linux kernel. In the system, user programs are executed in the kernel mode safely and the overhead of system calls is eliminated. Also, it provides the same facility as the original Linux kernel provides (e.g., an access control of a file system) by using inner functions of system calls of the original Linux kernel as an interface to user programs.

For performance measurement, we executed applications in the kernel mode on our prototype system. "find", which is a program that traverses directory trees of a file system, was executed 14 % faster in the kernel mode than in the user mode. Also, latency of "echod", which is a server program that receives data from a client and sends back the data to the client, was improved 4 μs in the kernel mode compared to the user mode.

The remainder of this paper is organized as follows: In chapter 2, we discuss related work. In chapter 3, we describe our approach and TAL. In chapter 4, we introduce our prototype implementation based on our approach. In chapter 5, we present experiments for performance measurement using our prototype implementation. In chapter 6, we conclude this paper and discuss future work.

Chapter 2

Related Work

In the field of operating system and programming language researches, there is several work related to safe execution of user programs in a kernel. Interesting difference of our research and the previous work is an objective. Our objective is to execute ordinal user programs in the kernel mode safely while the work below is concentrated on how to extend a kernel safely.

2.1 SPIN

SPIN [BSP+95] is an extensible kernel that ensures safety by a language based protection. In SPIN, as shown in Fig. 2.1, programmers write their kernel extensions in Modula-3 programming language and insert them into the kernel. Thus, the kernel extensions access the kernel resources and services with little overhead because there are no expensive transition between protection boundaries within the kernel. For example, the kernel exports interfaces that offer kernel extensions fine-grained control over a few fundamental system abstractions, such as processors, memory, and I/O. Therefore, the SPIN kernel enable kernel extensions to define customized kernel interfaces and implementation with which application-specific service can be built.

The SPIN kernel is protected from malicious kernel extensions because the kernel extensions are written in a type-safe programming language, Modula-3, and external trusted compilers compile the source programs to the binary programs. The type safety of Modula-3 helps guarantee that the kernel extensions don't violate the integrity of the kernel.

SPIN has two problems. The first problem is that its Trusted Computing Base(TCB) becomes larger because the kernel must trust external compilers of Modula-3. In SPIN, the kernel only checks integrity of an interface of kernel extensions and doesn't check

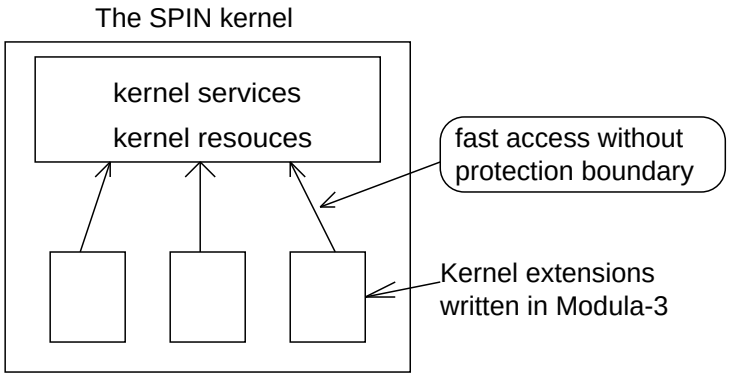

Figure 2.1: Overview of SPIN

the code of them. The second problem is that the kernel extensions must be written in Modula-3.

In our approach, on the other hand, TCB is smaller than SPIN because safety is checked at machine language level and we need not trust external compilers. Also, we can write user programs in various programming languages, if there exist compilers which translate the languages to TAL. In fact, there exist compilers which emit TAL from Popcorn(safe dialect of C) [Cor] or Scheme. (Note that we can compile non-typed languages, such as Scheme, to TAL without big performance degradation by encoding based on an universal type and soft typing [CF91].)

2.2 Software-based Fault Isolation

Software-based Fault Isolation (SFI) [WLAG93] is a technique that modifies an application binary to ensure memory and control flow safety. In SFI approach, an untrusted program is partitioned into a code segment and data segment (separation is needed to prevent the code from modifying itself). Each segment is a contiguous range of memory that all addresses in a segment are identical in their upper several bits (the bits are the segment ID). Then, check code is inserted before each memory access and jump to verify that the target address is valid, that is, the upper bits of the target address equal to the segment ID. Fig. 2.2 shows an example of a check code insertion of SFI. In the figure, the original memory access instruction (line 4) stores 0x5 to memory addressed by $Target$. The inserted codes (from line 1 to line 3) checks whether if the upper 16 bits of $Target$ equals to the segment ID, 0x12, and if not equal, that is, the original memory access is illegal, the code jumps to the error

handling routine (line 3).

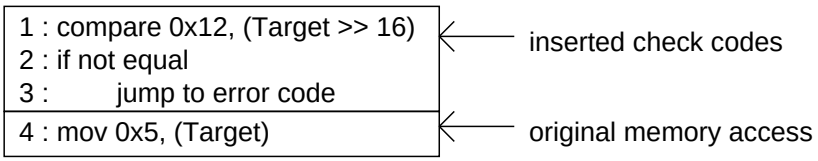

Figure 2.2: Example of check code insertion of SFI

The problem of SFI is a big overhead of runtime safety check.

In our approach, on the other hand, safety is mostly checked once at load time and there is no run time overhead (except for boundary checks of arrays).

2.3 Proof Carrying Code

In Proof Carrying Code (PCC) [NL96] approach, an user program is attached a logical proof of its safety, and the proof is verified before execution of the program. The proof is structured in such a way that makes it easy and foolproof for any agent to verify its validity without using cryptographic techniques or consulting with external trusted entities. There is also no need for any program analysis, code editing, compilation or interpretation to verify the proof. Besides being safe, PCC binaries can be executed fast because the safety check needs to be conducted only once, after which the kernel knows it can safely execute the program without any further run-time checking. Fig. 2.3 is an example PCC program as a flow chart, which calculates summation of an array A of size n. The annotations surrounded by "{" and "}" constitute a proof of the proposition that whenever a reference is made to array element $A[i]$, i is between 0 and $(n-1)$. For example, the assignment box $(i = i + 1)$ means that if i satisfies a condition: $0 <= i < n$, and the box is executed, i satisfies a condition: $1 <= i <= n$. To verify the proof, we need to check that the all propositions stated by the boxes are true.

The problem of PCC is that programmers or compilers are incurred a heavy burden of proof generation. Although PCC can ensure higher level safety than TAL such as memory and CPU usage limitation, it is very difficult to proof such high level safety.

On the other hand, in our approach, programmers and compilers are not incurred such a heavy burden. Also, memory and control flow safety suffice to ensure the same safety as traditional operating systems ensures because an MMU can only control read, write and execute of memory.

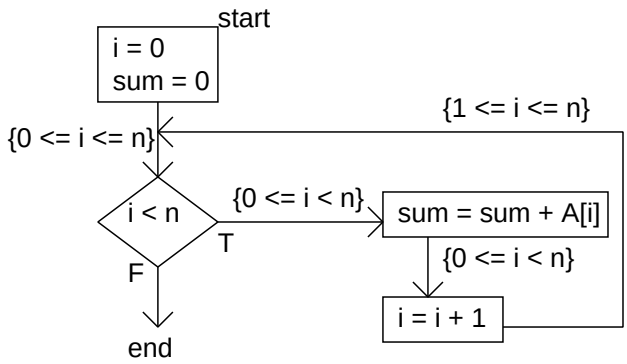

Figure 2.3: Example of Proof Carrying Code

TAL can be regarded as one kind of PCC which makes proof easy by limiting safety compared to the original PCC. For example, when compiling typed languages such as ML to TAL, safety proof (i.e., type information) can be automatically generated through type inference.

Chapter 3

Approach

3.1 Overview

Our approach consists of two stages: first stage is a program creation and second stage is a program execution.

Fig. 3.1 shows an overview of the program creation stage. In our approach, a programmer writes an application in TAL or other programming languages. If the programmer uses other language, a compiler of the language translates it to TAL. Then, a TAL assembler emits a machine code binary ant its type annotations. These type annotations are used in the program execution stage to check the safety of the program at machine code level.

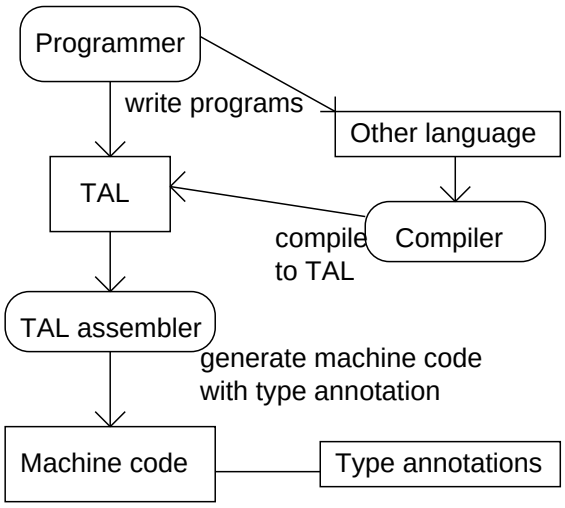

Figure 3.1: Overview of program creation in our approach

Fig. 3.2 shows an overview of the program execution stage. A kernel checks memory and control flow safety of an user program using a type checker with type annotations. If the type checker can verify the safety of the user program, the kernel can executes it in the kernel mode safely and efficiently.

In our approach, we need to add the type checker of TAL to TCB of the kernel. However, it doesn't become a problem because the type checker of TAL can be very simple and small. In fact, complexity of the type check of TAL is $O(n)$ while n is a size of a program.

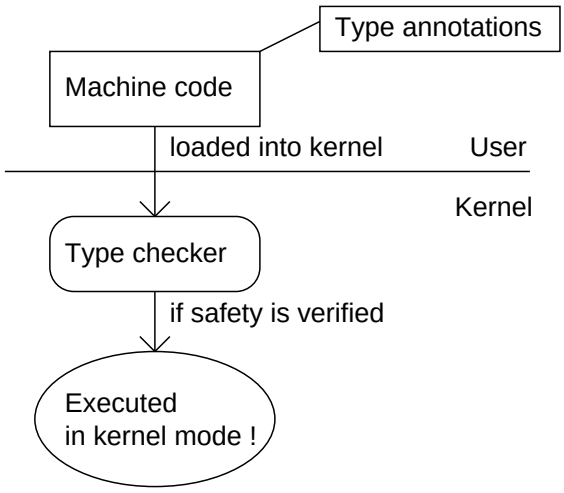

Figure 3.2: Overview of program execution in our approach

3.2 Typed Assembly Language

TAL is a "Typed Assembly Language" that ensures memory and control flow safety through its type system. Except for "typed", it is a completely usual assembly language.

Fig. 3.3 shows an example of an original IA-32 assembly code labeled "sum". The code adds value of the register EAX to value stored in memory which is pointed by the register EBX. Then the code jumps to a code which is pointed by the register EDX.

The program of Fig. 3.3 is not safe in a certain case. For example, see the program shown in Fig. 3.4. It stores a value which is not a valid memory address into the register EBX at line 7. and makes an illegal memory access at line 2. Thus, it is not memory safe.

```
1:  sum :
2:      addl %EAX, (%EBX)
3:      jmp  %EDX
```

Figure 3.3: An example of an original assembly code of IA-32

```
1:  sum :
2:      addl %EAX, (%EBX)
3:      jmp  %EDX
4:
5:  illegal :
6:      movl $0x1, %EAX
7:      movl $0x01234567, %EBX
8:      jmp sum
```

Figure 3.4: A program which makes an illegal memory access

For another example, see the program shown in Fig. 3.5. It stores a value which is not a valid instruction address into the register EDX at line 7 and makes an illegal code execution at line 3. Thus, it is not control-flow safe.

```
1:  sum :
2:      addl %EAX, (%EBX)
3:      jmp  %EDX
4:
5:  illegal :
6:      movl $0x1, %EAX
7:      movl $0x01234567, %EDX
8:      jmp sum
```

Figure 3.5: A program which makes an illegal code execution

TAL prevents such illegal memory accesses and code execution by typing register, memory and labels.

Fig. 3.6 shows a TAL program which is a typed version of the program shown in Fig. 3.3. Type annotations are added at line 1 and 2. Usually, the type annotations are added by a programmer when the programmer writes a program in TAL directly,

or by a compiler when a programmer writes a program in some programming language and uses the compiler which emits TAL program. In reality, TAL has more complex type annotations, such as types of a stack frame of functions, universal type and an existential type. We skip them in this paper for simplicity.

The meaning of the type annotation at line 1 and 2 in Fig. 3.6 is as follows: "<" and ">" declare that the annotation is a code label type. The code label type describes that a program can jump to the labeled code if and only if it satisfies conditions written between "<" and ">". For example, line 1 and 2 declare that a program can jump to the code labeled "sum" if and only if it satisfies three conditions:

- The EAX holds an integer value.
- The EBX holds a pointer to an integer value.
- The EDX holds a pointer to a labeled code to which a program can jump if and only if it satisfies two conditions:
  - The EAX holds an integer value.
  - The EBX holds a pointer to an integer value.

```
1: < %EAX : int, %EBX : int*,
2:    %EDX : <%EAX : int, %EBX : int*> >
3:   sum :
4:         addl %EAX, (%EBX)
5:         jmp  %EDX
```

Figure 3.6: An example of TAL program

These type annotations are preserved after a TAL assembler translates the TAL program to a machine code binary because the TAL assembler emits not only the machine code binary, but also the type annotations. Thus, safety of the machine code binary can be checked by using the generated type annotations.

Next section describes how to ensure memory and control flow safety through a type check of TAL.

3.2.1 Ensuring memory safety

Fig. 3.7 shows a TAL program which is a typed version of Fig. 3.4. It stores a value which is not a valid memory address into the register EBX at line 10 and tries to make an illegal memory access at line 4, as same as Fig. 3.4.

```
1: < %EAX : int, %EBX : int*,
2:    %EDX : <%EAX : int, %EBX : int*> >
3:   sum :
4:         addl %EAX, (%EBX)
5:         jmp  %EDX
6:
7: < %EDX : <%EAX : int, %EBX : int*>>
8:   illegal :
9:         movl $0x1, %EAX
10:        movl $0x01234567, %EBX
11:        jmp sum
```

Figure 3.7: A TAL program which tries to make an illegal memory access

The program cannot passes a type check of TAL. It tries to jump to the code labeled "sum" at line 11. However, because the type of the register EBX becomes integer type at line 10, the program doesn't satisfy the one of three conditions in order to jump to the code labeled "sum": "The EBX holds a pointer to an integer value." Like this, TAL programs can be ensured not to make an illegal memory access through a type check.

3.2.2 Ensuring control flow safety

Fig. 3.8 shows a TAL program which is a typed version of Fig. 3.5. It stores value which is not a valid instruction address into the register EDX at line 10 and tries to make an illegal code execution at line 5, as same as Fig. 3.5.

The program cannot passes a type check of TAL. It tries to jump to the code labeled "sum" at line 11. However, because the type of the register EDX becomes integer type at line 10, the program doesn't satisfy the one of three conditions in order to jump to the code labeled "sum": "The EDX holds a pointer to a code label to which a program can jump if and only if it satisfies two conditions":

- The EAX holds an integer value.
- The EBX holds a pointer to an integer value.

```
1: < %EAX : int, %EBX : int*,
2:    %EDX : <%EAX : int, %EBX : int*> >
3:   sum :
4:         addl %EAX, (%EBX)
5:         jmp  %EDX
6:
7: < %EBX : int* >
8:   illegal :
9:         movl $0x1, %EAX
10:        movl $0x01234567, %EDX
11:        jmp sum
```

Figure 3.8: A TAL program which tries to make an illegal code execution

Like this, TAL programs can be ensured not to make an illegal code execution through a type check.

3.2.3 Problems of current TAL

Current TAL has two problems from a practical point of view. However, researches for the problems are ongoing and we believe that they will be solved in near future at some level. In this section, we note the problems and the researches.

First problem is that current TAL relies on a garbage collector (GC). That is, current TAL assumes that GC is safe and safety of programs depends on safety of GC. To solve this problem, Crary et al. [CWM99], Smith et al. [SWM00] and Walker et al. [WM00] try to extends TAL in order to be able to manage memory directly by using a linear type and a region-based memory management.

Second problem of current TAL is that boundary checks of an array cannot be done before execution. That is, the checks are done at run time and an efficiency of programs may decrease. To solve the problem, Xi et al. [XH99] suggest DTAL which is a TAL variant extended by a dependent type. DTAL can eliminate unnecessary runtime boundary checks of arrays to some extent.

Chapter 4

Implementation

4.1 Overview

We implemented a prototype system base on our approach described in the previous chapter. The system was implemented by modifying the Linux Kernel on IA-32 CPUs.

Our prototype system provides three facilities. First, it provides a method to execute user programs in the kernel mode. Second, it provides a solution for the "stack starvation problem" which arises when executing user programs in the kernel mode. Third, it provides an interface to user programs in the kernel mode to call system calls.

The primary purpose of the implementation is to eliminate the overhead of system calls. Therefore, we decide to use the original inner functions of system calls as an interface to user programs. For the sake of this decision, our implementation can provide the same level safety(e.g., access control of file system) as the original Linux kernel provides.

Currently, our prototype system is not integrated with a type checker of TAL. Therefore, a safety mechanism of our prototype system is somewhat incomplete. However, there are no technical difficulties to integrate the type checker into our system and we can integrate it sooner or later.

4.2 How to execute user programs in the kernel mode

This section explains how to execute user programs in the kernel mode. First, we outline the segmentations of IA-32. Then we explain how IA-32 implements a privilege level facility. Finally, we describe our implementation.

4.2.1 Segmented memory model of IA-32

In IA-32, an address space is segmented and all memory addressing is "segmented addressing." That is, memory is addressed by a pair of a segment and an offset in the segment, as shown in Fig. 4.1. For example, the code shown in Fig. 4.2 addresses

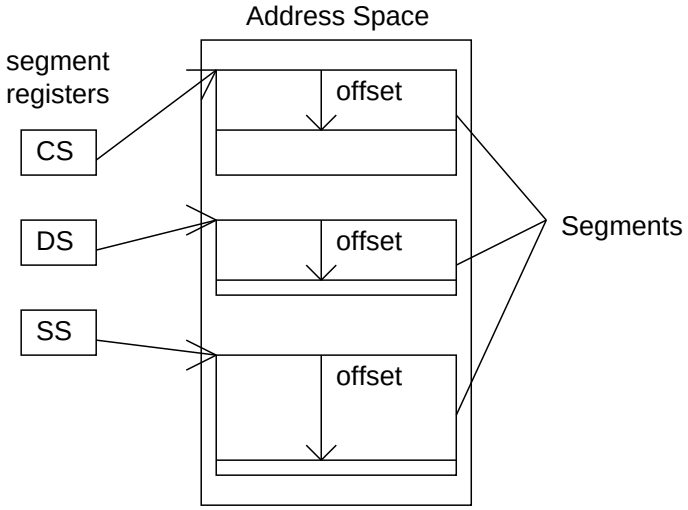

Figure 4.1: IA-32 segmented addressing model

the memory at offset 0x1234 in the segment pointed by the DS register. Although

```
movl    %DS:0x1234, %EAX
```

Figure 4.2: An example of explicitly segmented addressing

the example explicitly specifies a segment register, usually it is implicitly specified. For example, if the code accesses its stack through the ESP stack pointer register, it implicitly uses the segment pointed by the SS register. Also, a program counter of an IA-32 CPU is a pair of the segment pointed by the CS register and an offset represented by the EIP register. For example, if a program jumps to 0x1234, it implies that the program jumps to the offset 0x1234 in the segment pointed by the CS segment register.

The primary reason for using segmented memory model was to increase the reliability of programs and systems. For example, by placing a stack in a separate segment, we can prevent the stack from growing into the code or data space and overwriting them.

Nowadays, however, the segmented memory model loses its meaning because IA-32 has virtual memory mechanism. Therefore, current operating systems usually exploit the "segmented flat memory model", as shown in Fig. 4.3 and programs are protected by virtual memory mechanism. In the model, all segment registers point to segments which cover the whole virtual address space.

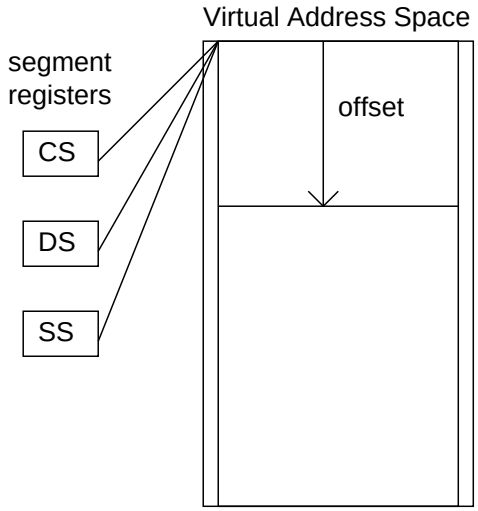

Figure 4.3: Segmented flat memory model of IA-32

4.2.2 Segments and Privilege Levels

As described in the previous section, segments of IA-32 lose its primary reason because of a virtual memory mechanism. However, they still have a very important function for traditional operating systems: Privilege levels.

In this section, we first outline privilege levels of IA-32, then describe their relation to the segments.

Privilege Levels of IA-32

As shown in Fig. 4.4, IA-32 has 4 privilege levels, numbered from 0 to 3. The greater numbers mean lesser privileges. In traditional operating systems, a kernel is executed at the privilege level 0 and user programs are executed at the privilege level 3. Thus, the kernel is protected from the user programs. For example, Fig. 4.5 shows how the Linux kernel protects itself from user programs. Because Linux uses only the two privilege levels, we denote the privilege level 0 as the kernel mode and the privilege

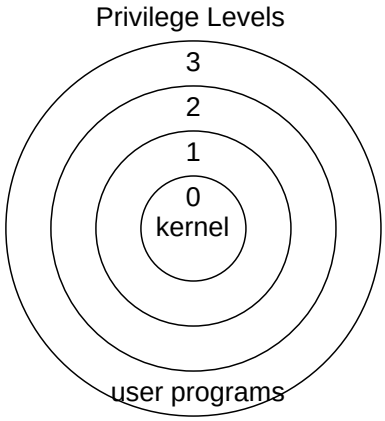

Figure 4.4: Privilege Levels of IA-32

level 3 as the user mode from now on. In Linux, the kernel is mapped in the bottom of an address space of user programs in order to reduce the overhead of context switches between the kernel and the user programs. To protect the kernel from user programs, Linux maps it in the kernel mode using its virtual memory mechanism and executes user programs in the user mode.

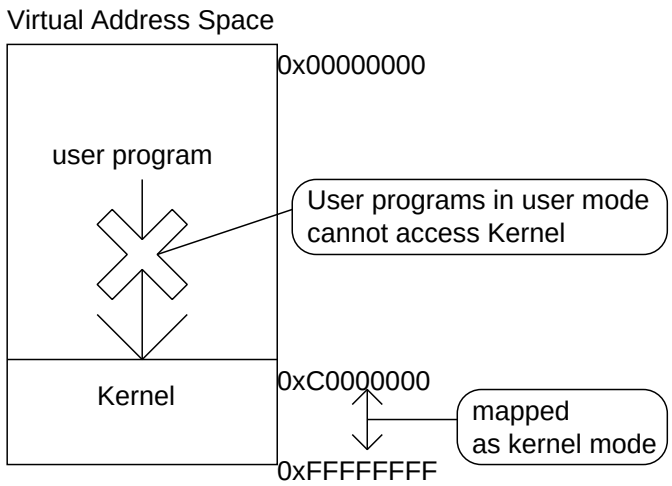

Figure 4.5: Protection of a kernel in Linux

What determines a privilege level of programs ?

The previous section describes the privilege levels of IA-32 and how to protect a kernel in the kernel mode from user programs in the user mode. Then, what determines the privilege levels of the programs ? The answer is "segments."

A privilege level of a running program is determined by the segment which is pointed by the CS segment register (Recall that a program counter of an IA-32 CPU is a pair of a segment and offset in the segment, that is, the CS segment register and the EIP register.) For example, if the segment pointed by the CS segment register is defined as the kernel mode, the running program is in the kernel mode.

In IA-32, a privilege level of a segment is defined in a segment descriptor, as shown in Fig. 4.6. It describes the size, location and privilege level of the segment and is stored in Global Descriptor Table (GDT) of a kernel. The segment register (i.e., CS, DS, SS and so on) hold an offset value to a segment descriptor in GDT and its least significant 2 bits are set to the privilege level of the segment. For security reason, a program can change its segment registers to a new segment only when the new segment has lower or equal privilege level than the privilege level of the running program. That is, if a program is executed in the user mode, its segment registers cannot point to a segment of the kernel mode. In the Fig. 4.6, the CS segment register points to the segment whose location is 0x1000, size is 0x0500 and privilege level is 3. Thus, the running program is in the user mode. (More accurately, the above explanation is the simplest case and there are many complex cases. For example, a segment descriptors can be defined not only in GDT, but also Local Descriptor Table (LDT). Also, the least significant 2 bits of a segment descriptor must not be equal to the privilege level of the segment. For simplicity, however, we ignore these complex cases in this paper.)

Fig. 4.7 shows the simplified version of GDT of the Linux kernel. The figure shows only privilege levels of segment descriptors because Linux uses the segmented flat memory model(see Fig. 4.3) and the location and size of the all segments are 0 and 4 giga bytes. The 0x10 segment descriptor is for a kernel code execution, the 0x18 segment descriptor is for a kernel data access, the 0x20 segment descriptor is for an user code execution, and the 0x28 segment descriptor is for an user data access.

In Linux, an user program is started in the state that its CS segment register is set to 0x23 and its DS, ES and SS segment registers are set to 0x2B. Thus, the user program is in the user mode.

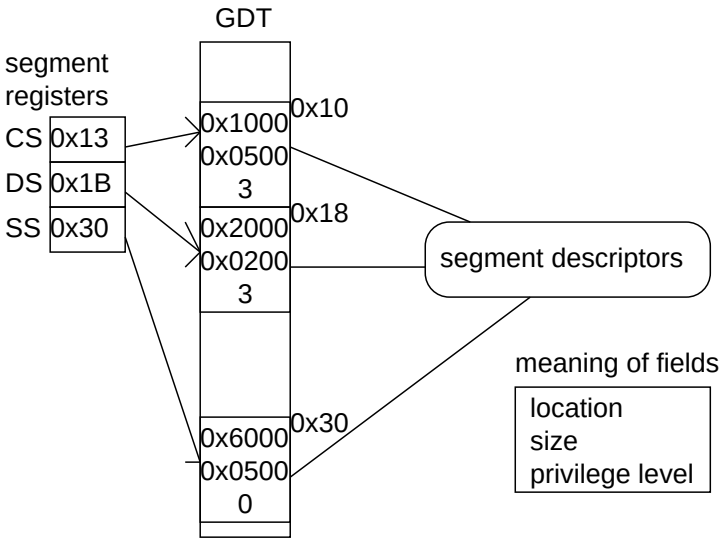

Figure 4.6: An example of segment descriptors

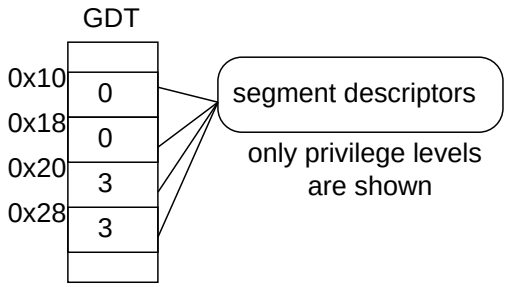

Figure 4.7: Segment descriptors in the Linux kernel (only privilege levels are shown)

4.2.3 Our method

Our method is very simple: start an user program in the state that its CS segment register is set to 0x10 and its DS, ES and SS segment register are set to 0x18. Then the user program is executed in the kernel mode.

The advantage of our method is that user programs are executed normally as in the original Linux kernel except for its privilege mode. For example, paging and preempting of user programs work successfully by a very subtle modification of a kernel.

4.3 Stack Starvation Problem

In this section, we describe the "stack starvation problem" which arises when user programs are executed in the kernel mode. First, we outline an interrupt handling mechanism of IA-32. Then, we describe the detail of the "stack starvation problem". Finally, we show our solution.

4.3.1 Interrupt handling with interrupt handlers in IA-32

Interrupts are forced transfers of execution from currently running program to a special procedure called an interrupt handler. For example, if a program accesses a memory page which doesn't have a valid mapping entry in page tables of an MMU, a page fault exception occurs and the running program is interrupted. Then, a page fault handler is executed and it takes an appropriate action. Finally, execution of the interrupted program is resumed.

In IA-32, interrupt handlers are associated with interrupts in Interrupt Descriptor Table (IDT). Fig. 4.8 shows a simplified version of IDT in the Linux kernel. Each entry of the IDT specifies an interrupt handler with a segment and an offset in the segment. For example, an interrupt handler for a page fault exception is at 0xC0106D4C in the kernel code segment (Recall that, in the Linux kernel, 0x10 means the kernel code segment). Thus, if a page fault occurs, the CS segment register is set to 0x10 and the EIP register is set to 0xC0106D4C.

4.3.2 Saving an execution context at interrupts

When an interrupt occurs, an IA-32 CPU not only jumps to an interrupt handler specified in IDT, but also saves the execution context of the running program in order to resume the interrupted program later.

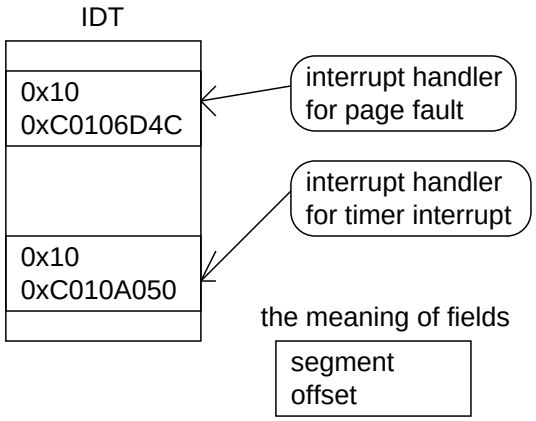

Figure 4.8: IDT in the Linux kernel

If an interrupt occurs in the kernel mode, an IA-32 CPU pushes the CS segment register, the EIP register and the EFLAGS flag register onto a stack, as shown in Fig. 4.9.

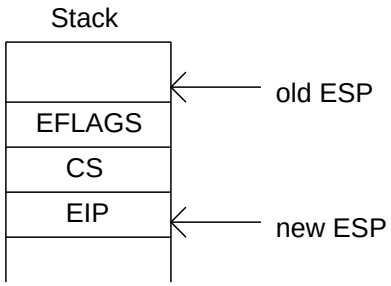

Figure 4.9: Saving execution context on a stack in the kernel mode

If an interrupt occurs in the user mode, first, an IA-32 CPU switches a stack from the user's one to the kernel's one because the user's stack may be not present because of, for example, a page fault. Then, the CPU pushes the CS segment register, the EIP register, the EFLAGS flag register, the SS segment register and the ESP stack pointer register onto the kernel's stack, as shown in Fig. 4.10.

4.3.3 Stack starvation in the kernel mode

There is a problem in the above interrupt handling mechanism in the case that user programs are executed in the kernel mode, as in our implementation. For example,

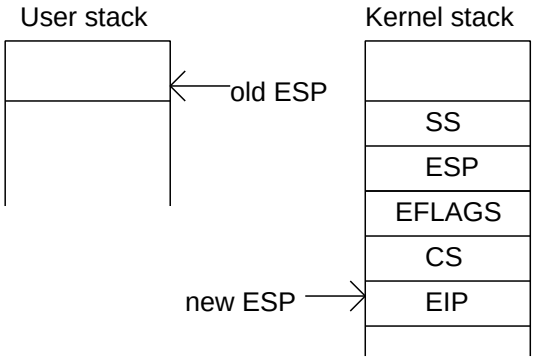

Figure 4.10: Saving execution context on a stack in the user mode

consider what happens if an user program in the kernel mode accesses its stack which is not mapped by page tables of an MMU. First, a page fault occurs and an IA-32 CPU tries to interrupt the running program and jump to a page fault handler. However, the CPU can't accomplish the work because the program is executed in the kernel mode and there is no stack to save the execution context. Then, to signal this fatal state, the CPU tries to generate a special interrupt called a double fault. However, again, the CPU can't generate a double fault, because there is no stack to save the execution context of the running program. Finally, the CPU gives up and resets itself. We named the above problem the "Stack Starvation"

4.3.4 Our solution

To solve the stack starvation problem, we exploit the task management facility of IA-32.

The task management facility of IA-32 is that a hardware of an IA-32 CPU switches a task (a thread of an execution and an address space) instead of an operating system software. Fig. 4.11 shows an example of the tasks. In the example, there are two tasks (task A and task B) and they have their own task state which is a per task memory storage that holds values of registers and a pointer to a page table. Also, a task pointer in an IA-32 CPU points to a task state of a current running task. Thus, the task A is running on the CPU in the example. Fig. 4.12 shows an example of a task switch from the task A to the task B by the task management facility of IA-32. First, an IA-32 CPU saves its registers and page table pointer to the task state of the current running task, the task A. Next, the CPU restores registers and a page table pointer from the task state of the next task to run, the task B. Finally, the task pointer points to the

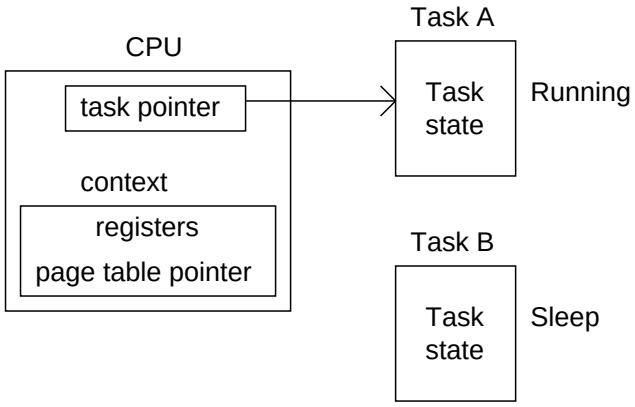

Figure 4.11: Example of tasks in IA-32

task state of the task B.

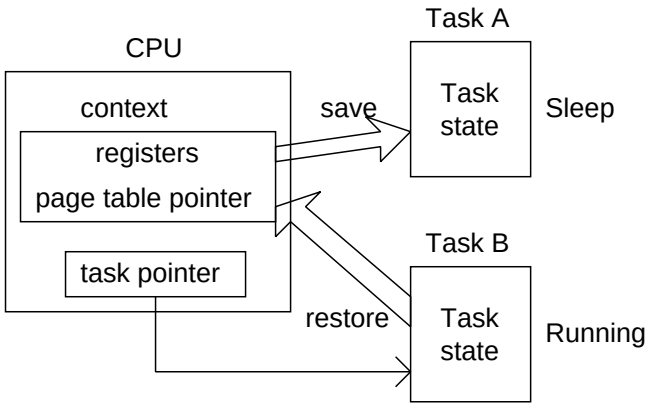

Figure 4.12: Example of a task switch by the task management facility of IA-32

An important feature of the task management facility of IA-32 is an interrupt handling using tasks. To handle an interrupt with a task, it must be set in IDT, as shown in Fig. 4.13. Then, if the interrupt occurs, an IA-32 CPU switches a running task to the task specified in IDT and the task takes an appropriate action.

A naive solution for the stack starvation problem is to handle a page fault with the task management facility because the facility switches tasks at a page fault and the CPU can save the execution context of the running program in a stack of a new running task. However, the solution has a big problem that the task management facility is very inefficient. Therefore, if we handle all page faults with a task, an efficiency of the

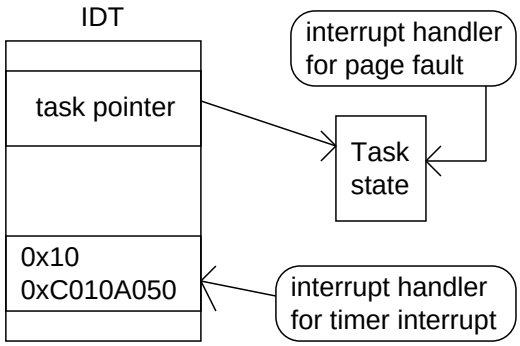

Figure 4.13: Set a task in IDT to handle an interrupt with the task

whole system may be degraded.

Our solution is that we handle not a page fault, but a double fault with the task management facility. The solution can minimize a degradation of an efficiency because it equals to handling a page fault with the facility only when an user program in the kernel mode tries to extend its stack. Appendix A shows the source code of the program which handles the stack starvation.

4.4 How to call system calls in the kernel mode

Basically, a system call by user programs in the kernel mode is a normal function call because the programs can access a kernel code directly in our implementation. The detail of system call facility in our implementation is as follows. First, a kernel exports information of its system calls. The information is a pair of the address of the system call and its TAL type which must be satisfied by an user program when it jumps to the address. Tab. 4.1 is an example of such information. (For simplicity, we use simple C-like type representation in the table.) Next, according to the exported

Type	Address
int getpid(void)	0xC02029C8
int socket(int, int, int)	0xC01186D8
int exit(int)	0xC011463C

Table 4.1: Example of exported system call information

system call information, programmers write their programs that call the system calls

as a normal function. Finally, the kernel checks the programs before executing them in the kernel mode whether if they conform the exported system call information. Thus, in our implementation, the kernel can safely let user programs call system calls as a normal function.

Also, in our implementation, we use the inner functions of system calls of the original Linux kernel almost "as is" and didn't change the interface of the kernel to user programs. Thus, our implementation ensures the same level safety as the original Linux kernel ensures. For example, to access files, user programs must use the inner functions of system calls such as open, read and write. Therefore, our implementation can ensure the safety of file accesses as long as the inner functions have no bugs, that is, the original Linux kernel ensures the safety. The bottom line is that our implementation replaces the safety check using privilege levels with a type check of TAL.

Although we mentioned that our implementation uses the inner functions of the system calls as is, in practice, they are wrapped with wrapper functions and the kernel exports the wrapper functions. The wrapper functions switch a stack to a non-fault stack because the inner functions of system calls require it. Also, they absorb the difference of representation of arrays and strings in C and TAL. In our implementation, an overhead of the wrapper functions is very small and negligible.

Chapter 5

Experiments

5.1 Experiments

We experimented our prototype system in order to see if user programs are executed in the kernel mode safely and the overhead of system calls is eliminated. For this purpose, we executed three programs in the kernel and user mode, and compared their efficiency. Also, we made a comparison with an approach that uses sysenter/sysexit [Int] instructions which are special instructions to accelerate system calls by switching between the kernel and user mode in a specialized way for system calls. IA-32 CPUs, after Pentium II, have the instructions and Windows XP uses them for fast system calls.

The compared three programs are **getpid**, **find** and **echod**.

getpid is a program which simply calls system call getpid. We compared **getpid** in order to investigate reduction of an overhead in a system call because system call getpid is very simple and short and the overhead of system calls becomes very large. We measured the time interval between the instant system call getpid is called and the instant it returns, with a time stamp counter of an CPU using rdtsc instruction.

find is a program which traverses directory trees as a find program of UNIX. We compared **find** in order to examine the effect of our approach on a realistic program. In the experiments, we traversed 35819 files and 2227 directories. Also, information of the directories are cached on a main memory because if disk accesses occur frequently, we cannot see the difference of efficiency because of the overhead of the disk accesses. We measured the time interval between the instant the program begins and the instant it exits, with a time command of bash.

echod is a server program which receives data from a client and sends back the data to the client on a TCP/IP network. We measured a latency of the **echod**, that is, the time interval between the instant a client begins to send data to the **echod** and the instant the client ends to receive the data, with a time stamp counter of CPU using rdtsc instruction. In the experiments, the data size is 8 bytes and the client was executed on the original Linux Kernel.

The above three programs are written in Popcorn programming language [Cor], which is a safe dialect of C. Tab. 5.1 shows the experimental environment.

	getpid,**find**, **echod** server	**echod** client
CPU	PentiumII 350MHz	PentiumIII 1GHz
Memory	256 Mbytes	512 Mbytes
OS	Linux Kernel 2.4.17	Linux Kernel 2.4.17

(Network : 100Mbps Ethernet)

Table 5.1: Experimental environment

5.2 Results

As shown in Tab. 5.2, **getpid** was executed 30 times faster and **find** was executed 14 % faster. Also, the latency of **echod** was improved 4 μs. Moreover, it shows that our approach improves efficiency larger than the sysenter/sysexit approach.

The reason for small improvement ratio of the **echod** latency is that TCP/IP communication has an overhead of not only system calls but also interruptions, context switches and protocol stacks, and it limits the effect of reduction of the overhead of system calls. In fact, 4 μs is not a "small" improvement because it equals to 1400 CPU cycles.

	getpid(ns)	**find**(ms)	**echod**(μs)
user mode	942	153	200
sysenter	411	141	200
kernel mode	31	131	196

Table 5.2: Results : Execution time of **getpid** and **find**, and Latency of **echod**(data size is 8 bytes)

Chapter 6

Conclusion and Future Work

This paper shows that user programs can be executed in the kernel mode safely by using TAL. In our approach, user programs are executed in the kernel mode and the overhead of system calls (e.g., interruptions and context switches) can be eliminated. We implemented a prototype system based on our approach and experimental results show that our prototype system eliminates the overhead of system calls successfully.

However, the results also show the limitation of our current system. For example, small improvement ratio of the latency of **echod** (see Tab. 5.2) shows that there exist applications such that efficiency cannot be improved largely by reduction of the overhead of system calls only.

The problem of our current prototype system is that redundant safety check is done at runtime because the system uses inner functions of system calls of the original Linux as is. For example, the inner functions check the validity of pointers passed from user programs. However, for the sake of type check of TAL, the checks are really unnecessary because the pointers never become invalid in our system.

The problem can be solved by modifying a kernel and exploiting the type system of TAL more aggressively. For example, we think that a kernel can be modified to export a network communication hardware to user programs as user-level communication technologies [PT98, THI96, DBC+97]. The user-level communication is a technology that achieves high-performance communication by exporting a network communication hardware to user programs directly and eliminating the overhead of system calls, buffer management and data copying. The problem of the user-level communication is a tradeoff between performance and safety. To achieve high performance, the kernel must export a 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 a network hardware by system calls and give up high-performance communication. However, by using our approach, the kernel can achieve both high-performance communication and safety because the overhead of system calls can be eliminated without losing safety.

6.1 Other directions

Although we applied our approach to a monolithic kernel (Linux), it also can be applied to a micro kernel. Traditional micro kernels have a problem of the overhead of communication between a kernel and user servers. By applying our approach, the overhead can be reduced largely.

Also, our approach can be applied not only to IA-32 but also other CPUs. Although current TAL is only for IA-32, it is not so difficult to build TAL for other CPUs. Moreover, our approach can be applied to embedded CPUs for PDAs or cellphones because our approach doesn't depend on memory protection facility of an MMU.

References

[BSP+95] B. Bershad, S. Savage, P. Pardyak, E. G. Sirer, D. Becker, M. Fiuczynski, C. Chambers, , and S. Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, pages 267–284, 1995.

[CF91] R. Cartwright and M. Fagan. Soft typing. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation (PLDI)*, pages 278–292, 1991.

[Cor] Cornell University. *Typed Assembly Language software distribution (Popcorn and Scheme--).* http://www.cs.cornell.edu/talc/releases.html.

[CWM99] Karl Crary, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. In *26th ACM SIGPLAN-SIGACT Symposium on Priciples of Programming Languages*, pages 262–275, January 1999.

[DBC+97] C. Dubnicki, A. Bilas, Y. Chen, S. Damianakis, and K. Li. VMMC-2: efficient support for reliable, connection-oriented communication. In *Hot Interconnects*, Aug 1997.

[Int] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual.* http://developer.intel.com/.

[MCG+99] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. TALx86: A realistic typed assembly language. In *ACM SIGPLAN Workshop on Compiler Support for System Software*, pages 25–35, May 1999.

[MWCG98] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to Typed Assembly Language. In *25th ACM SIGPLAN-SIGACT Sym-*

posium on Principles of Programming Languages, pages 85–97, January 1998.

[NL96] George C. Necula and Peter Lee. Safe kernel extensions without run-time checking. In USENIX, editor, *2nd Symposium on Operating Systems Design and Implementation (OSDI '96), October 28–31, 1996. Seattle, WA*, pages 229–243, Berkeley, CA, USA, 1996. USENIX.

[PT98] L. Prylli and B. Tourancheau. BIP: a new protocol designed for high performance. In PC-NOW Workshop, held in parallel with IPPS/SPDP98, Orlando, USA, 1998.

[SWM00] Frederick Smith, David Walker, and Greg Morrisett. Alias types. In *European Symposium on Programming*, pages 366–381, March 2000.

[THI96] H. Tezuka, A. Hori, and Y. Ishikawa. PM : a high-performance communication library for multi-user parallel environments. Technical report, Real World Computing Partnership, 1996.

[WLAG93] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. *ACM SIGOPS Operating Systems Review*, 27(5):203–216, December 1993.

[WM00] David Walker and Greg Morrisett. Alias types for recursive data structures. Technical report, Cornell University, March 2000.

[XH99] Hongwei Xi and Robert Harper. A dependently typed assembly language. Technical Report CSE-99-008, University of Cincinnati, 27, 1999.

Appendix A

Source Code of Stack Starvation Handling Routine

The following assembly code handles the stack starvation problem. It is set in the state of the task which is defined in IDT to handle the double fault exceptions.

```
ENTRY(double_fault_task)
/*
 * First,
 * we search the state of the task
 * in which the stack starvation problem may occurred.
 */

        /* store current task register to %eax */
        str %ax
        movzwl %ax, %eax

        /* load pointer to current tss(task state segment) into %edi */
        movl $gdt_table, %edx
        leal (%edx, %eax, 1), %edi
        movl 4(%edi), %eax
        movzwl 2(%edi), %edi
        movl %eax, %esi
        andl $0xff000000, %esi
        orl %esi, %edi
```

```
        andl $0x000000ff, %eax
        sall $16, %eax
        orl %eax, %edi
        /* %edi = current_tss */

        /* load pointer to previous tss into %ebx */
        movzwl (%edi), %eax
        leal (%edx, %eax, 1), %ebx
        movl 4(%ebx), %eax
        movzwl 2(%ebx), %ebx
        movl %eax, %esi
        andl $0xff000000, %esi
        orl %esi, %ebx
        andl $0x000000ff, %eax
        sall $16, %eax
        orl %eax, %ebx
        /* %ebx = prev_tss */

/*
 * Then,
 * we switch stack to non-faulting stack if needed,
 * by setting the stack pointer (%esp) to a kernel stack.
 */
        cmpw $__KERNEL_CS, TSS_CS(%ebx)
        jne 1f
        movl TSS_ESP(%ebx), %esi
        cmpl $TASK_SIZE, %esi
        ja 2f
1:
/* FIX_KERNEL_STACK_POINTER is a memory address where
         * a kernel stack address is stored. */
        movl (FIX_KERNEL_STACK_POINTER), %esi
2:
        movl %esi, %esp

        /* From now on, we can use stack. */
```

```
/*
 * Next,
 * we prepare the stack as if saved context at interrupt.
 * see Fig. 4.9 and 4.10.
 */
        cmpw $__KERNEL_CS, TSS_CS(%ebx)
        jne 3f
        movl TSS_ESP(%ebx), %esi
        cmpl $TASK_SIZE, %esi
        ja 4f
3:
        pushl TSS_SS(%ebx)
        pushl TSS_ESP(%ebx)

        movl TSS_ESP(%ebx), %esi
4:
        pushl TSS_EFLAGS(%ebx)
        pushl TSS_CS(%ebx)
        pushl TSS_EIP(%ebx)

        movw $0x0, 6(%esp)
        cmpw $__KERNEL_CS, TSS_CS(%ebx)
        jne 5f
        cmpl $TASK_SIZE, %esi
        ja 5f
        /* record stack switch in XCS */
        movw $0xffff, 6(%esp)
5:

/*
 * Then,
 * check whether if the stack starvation problem occured or not
 * If occured, jump to page fault handling routine.
 * If not, jump to real double fault handling routine.
 */
```

```
        /* %esi = prev_tss->esp */
        /* calling address_exists */
        addl $-4, %esi /* %esi = prev_tss->esp - 4 */
        addl $-12, %esp
        pushl %esi
        call address_exists
        addl $16, %esp

        testl %eax, %eax
        jne 7f
6:
        pushl $PAGE_FAULT_ERROR_CODE
        movl $page_fault_no_stack_switch, TSS_EIP(%ebx)
        andb $253, 37(%ebx) /* == andl $~IF_MASK, TSS_EFLAGS(%ebx) */
        movl %esi, %eax
        movl %eax, %cr2
        jmp 9f
7:
        addl $-12, %esi /* %esi = prev_tss->esp - 16 */
        addl $-12, %esp
        pushl %esi
        call address_exists
        addl $16, %esp

        testl %eax, %eax
        jne 8f
        jmp 6b
8:
        pushl $0
        movl $double_fault_no_stack_switch, TSS_EIP(%ebx)
9:
        /* some cleanups of EFLAGS register */
        andb $254, 37(%ebx) /* == andl $~TF_MASK, TSS_EFLAGS(%ebx) */
        movw $__KERNEL_CS, TSS_CS(%ebx)
        movl %esp, TSS_ESP(%ebx)
```

```
        movw $__KERNEL_DS, TSS_SS(%ebx)

        movl TSS_CR3(%edi), %eax
        movl %eax, TSS_CR3(%ebx)

        movl TSS_ESP(%edi), %esp

/*
 * OK. now jump to page fault handling routine,
 * (or real double fault handling routine)
 */
        iret
        jmp double_fault_task
```