School of Computer Science and Engineering

Faculty of Engineering

The University of New South Wales

seL4 API/ABI emulation approaches in Linux

by

Jiawei Gao

Thesis submitted as a requirement for the degree of

Master's of Information Technology

Submitted:	April 2021	Student ID:	z5242283
Supervisor:	Prof. Gernot Heiser	Topic ID:	

Abstract

This thesis examines different methodologies and designs that aim to realize the idea of emulating seL4 in Linux. This report will mainly focus on reviewing the related projects. The first three are QEMU, Cygwin, and Wine, which provide solutions to run either foreign operating systems or programs targeting the foreign operating systems on a different host operating system. Lastly, the UML project gives an idea of porting the Linux kernel to the userspace. This report also proposes two approaches, the first one is to perform emulation at the seL4 syscall API layer, and the second one is to provide seL4 kernel ABI layer. They all share similar goals but tackle the problem from different perspectives. An analysis in this report will reveal similarities and differences between those techniques. Evaluation approaches and applications will also be proposed at the end to demonstrate the underlying technical solutions and outcomes of this project.

Acknowledgements

The work of this thesis has been inspired by Prof. Gernot Heiser and Axel Heider who encourage me to explore the scope of this project leading to an enhancement in research skills and understanding of emulation technologies. Further support and encouragement have been given by James Nakoda Nugraha who provides me the recommendation, advice, and information about Linux. At last, I would like to thank you, my families and friends, who always support me.

Abbreviations

ABI Application Binary Interface

API Application Programming Interface

ASLR Address Space Layout Randomization

DLL Dynamic-link library

IPC Inter Process Communication

IRQ Interrupt Request

ISA Instructions Set Architecture

KVM Kernel-based Virtual Machine

OS Operating System

PID Process Identifier

POSIX Portable Operating System Interface

RPC Remote Procedure Call

TCB Trusted Computing Base

UML User-mode Linux

Contents

1 Introduction 1

2 Background 2

2.1 Overview of seL4 2

2.1.1 Cpabilities 3

2.1.2 Communication 3

2.1.3 Memory Management 5

2.1.4 Scheduling 6

2.2 Challenge 7

2.3 Related Work 9

2.3.1 Full system emulation 9

2.3.2 API-level Emulation 11

2.3.3 ABI-level Emulation 14

3 Approaches 23

3.1 API Layer Emulation Approach 23

3.1.1 Design Model 24

3.1.2 Advantages and Disadvantages 25

3.2 ABI Layer Emulation Approach 26

3.2.1 Design Model 26

3.2.2 Advantages and Disadvantages 28

3.3 Comparison Between API and ABI Layer Emulation 29

4 Evaluation 30

4.1 Qualitative Evaluation 30

4.1.1 seL4 Tutorials 30

4.1.2 seL4Test 31

4.1.3 Simple operating system project 31

4.2 Quantitative Evaluation 31

4.2.1 Microbenchmark 32

4.2.2 Macrobenchmark 33

5 Project Plan 34

5.0.1 Timeline 34

Bibliography 36

List of Figures

2.1 Linux based OS model vs seL4 based OS model. 7
2.2 The architecture of Cygwin 12
2.3 The control flow of handling system call in UML 17
2.4 The control flow of handling signal in UML 19
2.5 The control flow of handling signal in UML 20

3.1 The model of API layer emulation approach 24
3.2 The model of ABI layer emulation approach 27

List of Tables

3.1 Comparison between API and ABI layer emulation 29

4.1 Expected success criteria 32

Chapter 1

Introduction

seL4 provides a very secure environment to run any untrusted applications as separate components on top of it. What's more, it maintains a good performance as well as having a strong isolation guarantee of the applications and the kernel to keep the entire system robust and safe. However, developing seL4 systems is not trivial because of the development ecosystems of seL4. On the other hand, Linux has a much larger and more complete ecosystem than seL4 does, therefore, in this project, we are going to explore some approaches to emulate the entire the seL4 system in the Linux by both executing seL4 user-level applications and emulating the seL4 kernel functionalities. The outcome of this project will let us leverage helpful tools such as debuggers in Linux to make developing seL4 systems much easier and faster.

Chapter 2 explains the related projects that have been done before, followed by a detailed explanation of those projects. Chapter 3 states the rough design models and implementation ideas of this project. Chapter 4 explains the success criteria defined for this project and the methods that will be used for the evaluation. Chapter 5 states the plan for the future research project and the timeline for the next semester.

Chapter 2

Background

In the first section, we are going to introduce some basics of the design in seL4, then move on to the next section where defines the challenge that we are going to solve in this thesis. Besides, to clarify the term, we are going to call anything that runs at seL4 user-level as "seL4 applications". Also, we will classify those applications that request OS services as "client applications" and those applications that provide OS services as "server applications". In the final section, we will introduce some related projects that provide emulation at ISA, ABI, and API layers and then discuss their trade-offs. We will first introduce an existing solution that uses QEMU. Then we will introduce other three related projects from the API layer emulation approach using Cygwin down to ABI layer emulation approaches provided by Wine and UML.

2.1 Overview of seL4

seL4 has been formally verified for correctness ([KEH+09]) and is the fastest microkernel in the world. seL4 adheres to the principle of *minimality*, only provides the critical kernel functionalities that can't be achieved in the user-level implementation. Such design minimizes the TCB in seL4, making it possible to be formally verified. seL4 was born for safety. The componentized system architecture that seL4 implements enforce the isolation between those untrusted components and other trusted components running on top of the kernel. With the fine-grained capability-based access control model seL4 can carefully manage the access to the hardware resources from the software components. In this section, we will present the current state of relevant seL4 features to motivate this thesis. First, we will introduce the powerful and robust capability system in seL4. Then we will introduce the system calls in seL4 and the resource management. Finally, we will introduce the scheduling system in seL4.

2.1.1 Cpabilities

In seL4, the access rights to any resources are represented by capabilities. When the kernel launches the initial task, which is the *root task*, the capabilities to all the resources are also handed to it. The *root task* is the first seL4 user-level thread which will take care of the resource allocation in the future. Capabilities reside in the seL4 capability space which is also called *cspace*. A *cspace* consists of several capability nodes called *cnodes*, each *cnode* is a table that has several slots. Those slots can contain either a capability to the next *cnode* or a capability to a kernel resource or be empty. Hence, *cnodes* form a directed graph in the *cspace*. The *cspace* maps the access rights to kernel object identifiers. Resolving the capability requires resolving the *cspace* address to find the final slot. According to the seL4 semantics, there are several operations to manipulate capabilities for different purposes, such as copy, mint with a badge, move, mutate with a badge, delete or revoke, etc. When the capability is copied or minted with a specific unforgeable token called a badge, it's said to be derived. The seL4 kernel will keep track of the capability derivation via the capability derivation tree to delete a capability (only remove such capability from the *cnode*) or revoke (also remove all the capabilities derived from such capability) it later.

2.1.2 Communication

The seL4 provides both synchronous communication methods through IPC and asynchronous methods using notifications.

IPC

In the seL4, two threads can communicate with each other via either a blocking fashion or a non-blocking fashion over an endpoint. It can be a one-way message passing from sender to receiver, or a two-way fashion that implements a *rendezvous* model, where the sender will wait for a reply from the receiver. In the design of seL4, to send the message via IPC, the sender must specify the length of the message payload. If the message payload is relatively large, then it will be stored in each thread's IPC buffer, and the kernel will copy the payload from the sender's buffer to the receiver's buffer. However, If it is small, no memory copy is required, the message payload will be stored in the registers instead to improve performance. In seL4 the system calls are mainly for communication, they are *seL4_Send*, *seL4_NBSend*, *seL4_Call*, *seL4_Reply*, *seL4_Recv*, *seL4_NBRecv*, and *seL4_Yield*. The first four are sending system calls, the middle two are receiving system calls and the last one is the scheduling system call. Most other methods for manipulating the kernel objects are implemented using those critical system calls. Since the IPC mechanisms are critical to the performance in seL4, seL4 also implements a fast code path to further optimize the IPC performance when several conditions are satisfied:

1. the sender is using $seL4_{C}all$ or the receiver is using $seL4_{R}eplyRecv$,
2. No capbilities should be transferred,
3. The data in the message must fit into the $seL4_{F}astMessageRegisters$ registers without using IPC buffer,
4. the receiver has a valid address space without any memory faults,
5. the receiver has the highest priority among all the threads in the scheduler,

Besides the system calls, the seL4 also utilities IPC between teh kernel and the receiver to handle the thread faults. In the seL4, when a thread generates a thread fault, the kernel will suspend the execution of the faulting thread and constructs a message, which contains the information about the faulting thread, then send the message to the specific endpoint where a fault handler thread is waiting on. After the fault handler corrects the anomaly, it can either explicitly tell the kernel to resume the faulting thread or invoke the reply object to inform the kernel.

Notification

The notification mechanism provides an asynchronous method for seL4 threads to send signals to each other. It is primarily used for handling the interrupt and maintain synchronisation between threads. The notification object can have three states: waiting, active or idle, and it consists of a word of data, where the badge will be stored and accumulated if no threads are waiting on it. By using notification objects, a thread can poll or block on an interrupt. On the other hand, a notification object can also be bound to the thread. Hence, the thread can receive the notification issued while blocking on the endpoint waiting for IPC. Such design avoids using one dedicated thread for receiving IPC and the other one for receiving notifications.

2.1.3 Memory Management

Since the design seL4 adheres to the *minimality principle*, the seL4 kernel itself doesn't handle the memory management. Such OS services are implemented at the user-level. When the kernel boots, it will first allocate a small, static amount of memory for itself. Those memories have already been mapped so that the kernel can directly access them. However, the size of such memory is platform-specific. Then it will start the *root task*, and pass the untyped physical memory along with the capabilities and all the resources to the *root task*. Those untyped memories can be retyped into different kernel objects such as *cnodes*, *frames*, *page tables*, etc. With such a design, it's very flexible for the system developer to decide the specific memory layout. However, no fixed-size data structure can be used in the kernel as the kernel can't assume the memory layout.

2.1.4 Scheduling

In the seL4's implementation, the kernel will choose the head of the list queued at the scheduler with the highest priority as the next thread to run. To lookup for the highest priority efficiently, seL4 uses a two-level bit field to track the priority lists consisting of runnable threads in the scheduler. The first level of the bit field divides the priority levels into several sets, each bit represents a set of the priority levels. Also, it is the index to the second-level bit field array. Each bit in the entry of the second-level bit filed array maps a priority level in the current set. Such data structure ensures only two words need to be resolved to get the highest priority.

Traditional Scheduling

In seL4, there are two main scheduling policies, the first policy is the traditional one that implements a round-robin scheduler with 256 priority levels. Each thread has two attributes, a maximum controlled priority and an effective priority which must be less than the maximum controlled priority. The priority levels can be changed through the thread's capabilities. A Thread can run until it exhausts its time slice or gets preempted by the other threads with higher priorities, or it can give up the execution manually by calling *seL4_yield*.

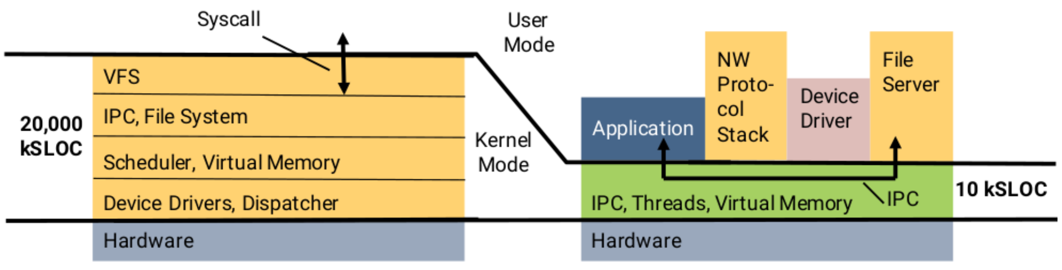
MCS scheduling

In the traditional seL4, the schedular prioritizes the thread with the highest priority, which means if the thread happens to be the only one with the highest priority, it can monopolise the CPU unless a thread with higher priority occurs or it finishes its task. To overcome such a problem, seL4 introduces its own MCS scheduling. The seL4's MCS scheduling can be enabled by changing the configuration when building the kernel. In the implementation of the seL4's MCS, each thread has a scheduling context that contains a budget that determines how long the thread can run and a period that determines how often the thread can run. Those two attributes form an upper bound of the thread's execution time and prevent the high-priority thread from monopolising the CPU. The seL4's MCS scheduling mechanism allows it to provide strong time resources isolation to the components with different criticalities besides the space resource isolation which is guaranteed by capabilities ([LMAH18]). Therefore, seL4 offers great support for real-time systems.

2.2 Challenge

The secure and well design of the seL4 microkernel together with the formal verification ensure the kernel itself is robust and is the ideal foundation to build a secure system on top of it ([KAE$^+$10]). What's more, seL4 is not only secure but also fast. With its performant IPC mechanisms and most advanced mixed critical real-time systems, seL4 is capable of handling a wide range of real-world scenarios ([HKA20]). However, seL4 is relatively young compared to other mainstream kernels such as Linux, and its ecosystem is still growing. For those seL4 developers, there are limited tools that can be used to develop seL4-based systems or applications. On the other hand, Linux provides us a powerful development environment with many useful tools. Therefore, the main motivation of this project is to explore some ways to leverage the Linux development tools to make developing seL4 systems much easier and faster.

But developing applications targeted for seL4 in Linux is challenging because seL4 as a microkernel has different semantics and OS model compared to Linux (in figure 2.1).



Source: "The seL4 Microkernel – An Introduction" (p. 3), by Gernot Heiser, 2020, the seL4 foundation. Copyright under the Creative Commons Attribution-ShareAlike 4.0 International (CC BY-SA 4.0) License.

Figure 2.1: Linux based OS model vs seL4 based OS model.

Linux is a monolithic kernel that provides a wide range of OS services. The drivers and many OS components are implemented inside of the kernel. Many those OS services execute in kernel-mode. In contrast, seL4 is just a thin wrapper of the CPU, providing minimal critical services of the hardware resources, and thus it guarantees strong isolation of different OS components. The OS services are implemented at the user-level as separate components. Such design minimizes the TCB.

In a monolithic kernel, the user-level applications request the OS services through the system call interfaces. A system call will cause a context switch from the user-mode to the kernel-mode, and the kernel will serve the OS requests by itself. For example, an I/O request from the user-level will trigger the system call and the kernel will dispatch such request to the corresponding handler. In Linux, this might be a handler in the VFS layer. Then the VFS layer handler will find the particular file system handler to perform the I/O. However, in the seL4 based OS model, it works differently. When the seL4 client applications request OS services, it will invoke IPC mechanisms provided by the seL4 kernel. The seL4 IPC is a synchronous or asynchronous method for transmitting small amounts of data and capabilities between seL4 threads. seL4 applications can invoke seL4 IPC mechanism by calling a seL4 system call, which triggers the context switch to the kernel-mode, and the seL4 microkernel will then transport the input to the designated seL4 server application via the endpoint. Those requests are served by the seL4 server application at the user-level. The seL4 client applications and seL4 server applications are isolated from one another, which ensures the security of the whole system.

As we can see, the main problem here is that seL4 and Linux have different semantics and program flow of requesting OS services, therefore to achieve the goal of this project, which is to run seL4 applications on Linux, we have to develop several methods to provide seL4 semantics on top of Linux.

2.3 Related Work

In this section, we are going to introduce several related projects which provide emulation at different levels from ISA layer to API layer. First, we will present an existing solution to the challenge we stated before, which uses QEMU to provide a full system emulation. Then we will present an API-level emulation project called Cygwin and two ABI-level emulation approaches called Wine and UML.

2.3.1 Full system emulation

The full system emulation provides a way to run an unmodified target binary code, such as applications or even operating systems like Linux or Windows on a host machine, even if the emulated target is compiled for different ISA from the host's.

QEMU

QEMU is a full system emulator supporting a wide range of ISA, and QEMU can also act as a virtualisation hypervisor if the guest's ISA is the same as host's ISA and the host supports Linux KVM ([Wik21b]). By using a full system emulator such as QEMU, we can readily run the seL4 microkernel and its applications on top of Linux, and from the seL4 user-level applications' view, they are still running on top of the seL4 microkernel.

(TODO: new QEMU uses TCG doesn't use GCC at all) QEMU is a machine emulator, which means it can use portable dynamic translation to emulate an entire machine, including its processors and peripherals. To convert the guest instruction set into the host instruction set, QEMU will perform two phases. The first phase is the translation and the second phase is the code generation. In the first phase, QEMU splits each target CPU instruction into a few instructions called micro-operations. Each micro-operation is implemented using a small piece of C code that will be compiled by GCC to an object file later. A novel design in QEMU is that QEMU uses dummy code relocations generated by GCC as a place holder with a prefix *__op_param* in the micro-operations, and patches it later when the code generator runs. Because some constants parameters in the target instructions are only known in the runtime. Those relocations can also be references to static data or functions. After that, the *dyngen* will be invoked to generate the dynamic code generator. It will parse the previously generated object file depends on the host operating system's object file format, such as ELF in Linux, to get the symbol table, relocations, and code sections and use that information as the input to generate a dynamic code generator. In the second phase, the dynamic code generator will copy the stream of micro-operations into the host code section and patch the relocations with the runtime parameters. Those relocation patches are host-specific. For optimization, QEMU will maintain a fixed size cache to hold the recently translated blocks of instructions.

Since QEMU can emulate CPU using dynamic binary translation, this means that QEMU has great flexibility, such as allowing binary code, such as running ARM binary on x86. However, emulation using dynamic binary translation introduces a massive performance overhead, therefore, recent versions of QEMU can leverage hardware virtualisation, such as Intel VT-x or AMD-V, which are supported by modern processors to allow full system virtualisation at near-native speed. These technologies may directly leverage the physical CPUs as the virtual CPUs. Hence, the instructions of the virtual CPU can directly execute on the physical CPU slice. In Linux, QEMU leverages KVM to access CPU virtualisation extensions. However, since KVM is a driver for the physical CPU capabilities, this approach is very tightly associated with the CPU architecture and/or ISA. It means that the benefits of hardware acceleration will be available only if the target system is targetting the same ISA as the host's and the host processor supports hardware virtualization feature. Moreover, though QEMU provides its debugging interfaces to debug the guest system in a single stepping mode.

Although QEMU is an existing solution to emulate the seL4 system in Linux at the ISA layer, it's not an optimal option for emulating a high-performance microkernel like seL4 because of the overhead of hardware virtualization or machine emulation. In the former case, if the guest system such as our seL4 performs a privileged instruction, then it will trigger a trigger a world switch to the host (e.g. VMEXIT) to the host, and such overhead is not negligible. In the latter case, if QEMU resorts to dynamic binary translation, then it will have to perform a full machine emulation and translate the ISA into the host's ISA, which introduces significant overhead. However, the design goal of QEMU focuses on low-level machine emulation. It's helpful to inspect each instruction's execution of the CPU, but it's hard to use from a high-level seL4 developers' perspective. Because the debugger is not OS aware, the guest OS's context switch might confuse the debugger and lead to a breakpoint missing or a wrong variable value issue. In the following sections, we will introduce other approaches performing emulation at different layers, and then propose approaches used in this thesis to tackle the problem that we stated before in chapter 3.

2.3.2 API-level Emulation

As we mentioned before, the ISA-level emulation involves emulating the emtire machine or using virtualization technologies, which introduces non-negligible overhead. In this section, we will introduce a project which performs emulation approach at the API layer to provide compatibility between two different systems.

Cygwin

Cygwin is a compatibility layer that allows Unix-like applications to run on top of Windows ([Wik21a]). This is achieved by introducing a DLL called cygwin1.dll which acts as an emulation layer providing substantial POSIX system call functionalities providing a Unix semantics (figure 2.2). Cygwin also provides several standard Unix utilities such as bash, etc.

Cygwin began development in 1995 at Cygnus Solutions. In the project, developers provided interfaces called Cygwin API to add the missing Unix-like functionalities in Win32 API, such as fork, signals, select, etc ([Cyg]). Cygwin will not magically make any POSIX application's binary code runnable on top Windows directly. Executing

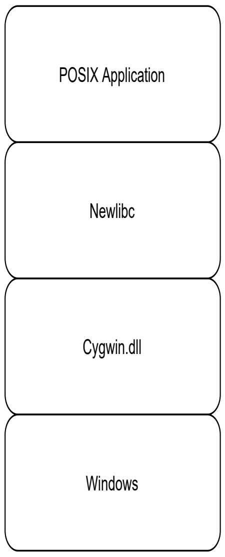


Figure 2.2: The architecture of Cygwin

those POSIX applications require the recompilation of the application. source code of the POSIX application can be recompiled using Cygwin and linked with DLL which implements the POSIX system call semantics using the Win32 APIs and native NT APIs. After executing the application, the Cygwin DLL will be loaded into the application's test region so that it has full access to the whole process. Next, shared memory is created containing the instances of the resources that the shared library can access. Therefore, the OS resources such as file descriptors can be tracked. Besides such shared memory regions, each process also has its resource bookkeeping structures, such as signal masks, PID, and so on.

Cygwin has several nice designs to implement the POSIX features in Windows. For example, it handles signals by starting a separate thread from the library for only signal handling purposes. This thread waits for the Windows event used to pass signals to the process, and scan through its signal bitmask to handle the signals appropriately. However such thread resides in the same address space as the executing program, the signal sending function for sending a signal to other processes is wrapped with a mutex, and the signal sending function which sends the signal to itself is wrapped by separate semaphore or event pair to avoid them being interrupted.

Another example is the implementation of sockets. They are mapped on top of the Winsock. However, Unix domain socket is not provided in Windows, so Cygwin implements it with the local IPv4 as the address family. Besides, Cygwin provides the Winsock initialization on the fly, as the Winsock requires to be initialized before the socket function is called. For implementing the POSIX *select*, Cygwin implements the polling of file handles besides socket type handles by sorting the file descriptors into different types and creating a thread for each type of the file descriptors present to poll those file descriptors with Win32 API. Such a design is because the Winsock only works on socket type file descriptors.

The API layer emulation requires to utilize the host systems' functionalities, but it's difficult to find an appropriate functions provided by the host systems sometimes. For example, in Unix-like systems, the $fork()$ system call provides a way to create a child process that copies the parent's address space, but there are no appropriate process creation functions in Windows which can be mapped on top of it, so the implementation of $fork()$ in Cygwin is relatively complex.

Implementing $fork()$ semantics in Windows requiring to copy all of the executable binary and all the DLLs loaded statically or dynamically to be identical as to when the parent process has started or loaded a DLL. This can be problematic as Windows allows the binaries to be renamed to even removed to the recycle bin while the binary is executing, which means they can reside in a different directory or have a different file name. However, an executing process has to access to the binary files via the original filenames to fork its child processes. The solution to this problem is that Cygwin will try to create a private directory that contains the hardlink to the original files and remove it when no process is using it. When the parent process wants to fork a child process, it will first initialize a space in Cygwin process table for the child and create a suspended child process using Win32 CreateProcess API.

After that, the parent process will use setjump to save the context and set a pointer to the current context in the Cygwin shared memory, and fill the child process's .data and .bss section with its own address space. Next, the parent process will block on a mutex and the child process will run and use the longjump to jump to the saved jump buffer. Then child process will release the mutex that the parent is blocked on and waits for another mutex. The parent process will copy its stack and heap into the child process space then release the mutex that the child is blocking on and return from the fork call. Finally, the child process will wake up and recreates any memory-mapped areas that passed to it and return from the fork call. However, such implementation is not perfectly reliable as in Windows, Windows implements the ASLR starting from Vista, which means the stack, heap, text, and other regions may be placed in different places in each process. This behavior interferes with the POSIX fork's semantic that is the child process has the same address space as the parent process. In that case, Cygwin will try to compensate the movable memory regions at the wrong place but can't do anything with those unmovable regions such as the memory heap.

In summary, adding compatibility at the API layer can solve most of the problems when attempting to run applications from foreign systems. While the downside is that, apparently this will only work when the user of Cygwin can obtain the source code of the application. Meanwhile, it's very difficult to implement every POSIX system call in Windows correctly due to the huge difference in the internal design between the two systems.

Even though Cygwin can be used to run POSIX applications on Windows NT environment, we can still leverage the nice idea that Cygwin uses. To emulate seL4 in Linux we can link applications to a specific library that remaps the libseL4 APIs with the underlying host's system calls.

2.3.3 ABI-level Emulation

In the previous section, we have introduced Cygwin, which is an API-compatible solution for providing compatibility between Windows and Linux. However, the downside is that it's not binary compatible as we have to obtain the POSIX application's source code first and recompile it. In that case, if we can't obtain the source code or the application that we want to emulate directly invokes system calls instead of calling them via the standard libraries, then it's impossible to perform the emulation. Hence, in this section, we will introduce two ABI-level emulations which are binary compatible.

Wine

In the previous section, we have introduced an approach to run POSIX applications on top of Windows using Cygwin. In this section, we will introduce a way to run Windows applications on a Unix-like OS such as Linux. The challenge is that Windows and Linux use different ABIs. An ABI is a low-level hardware-dependent format that defines how the computational routines or data structures are accessed by machine code. Considering the difference between the Window's ABIs and the Linux's ABIs, such as calling conventions, binary format of executable files, system call semantics, and so on, it's impossible to execute a Windows application on top of Linux directly. However, Wine provides such binary compatibility between those two systems ([Wik21c]).

Unlike QEMU, Wine doesn't emulate the internal Windows logic like a virtual machine or an emulator. Instead, it directly translates the Windows ABI into POSIX-compliant calls. It implements Windows's ABI entirely in the user space, rather than a kernel module. Wine supports several executable formats and provides custom binary loaders for those fromats, such as DOS executable that is an old format, Windows NE executable that uses the 16-bit format, Windows PE executable that becomes the native format after Windows 95, and the Winelib executable that is written using Windows API but compiled as a Unix executable. Because Wine can support old Windows executable format, its internal implementation works differently depending on which executable format is executed. For example, Wine will launch a separate Unix process for each Win32 process with the PE format. However, it will only create Unix threads for Win16 tasks in a dedicated Wine process called Windows on Windows process. To allow the Win16 process run in a separate address space, Wine also implements a process called *winevdm*.

Implementation-wise, Wine leverages the fact that most Windows applications usually invoke some particular DLL libraries, which invoke the user-mode gdi32/user32 libraries, and then finally invoke the system calls via Windows subsystems such as Win32 instead of invoking a system call directly. This is somehow similar to how seL4 applications request OS services through invoking APIs provided by the seL4 system call libraries.

Wine's architecture is close to the Windows NT architecture (figure **??**). In Windows NT architecture, the gdi/gdi32, user/user32, and the kernel/kernel32 are the core system DLLs as well as the NTDLL allows implementing different subsystems layered on top of it such as Win32 in the kernel32 DLL, therefore, Wine completely implements those four core system DLLs to provide compatibility. Wine implements those DLLs similar to the APIs in Windows platforms as much as possible so that Win32 applications can be linked with those Wine DLLs and thus can invoke Wine's internal functions. Those functions can be translated into POSIX equivalent by Wine.

The other backbone of Wine's architecture is the Wine server which mainly implements the communication, synchronisation, and management mechanisms of Wine's processes. The communication between Wine's processes and the Wine server uses UDS and the shared memory called request buffer. Each thread of the process in Wine shares a request buffer with the Wine server. Hence, when a thread wants to communicate with other threads or processes, it can fill the request buffer and informs the Wine to handle the request via the socket.

Even though Wine gives a wise way to provide binary compatibility at the ABI layer, we can't imitate the whole implementation of Wine, as in Wine's architecture most of the libraries are dynamically loaded, but libraries are statically linked in seL4. In the next section, we will introduce another approach to achieve binary compatibility.

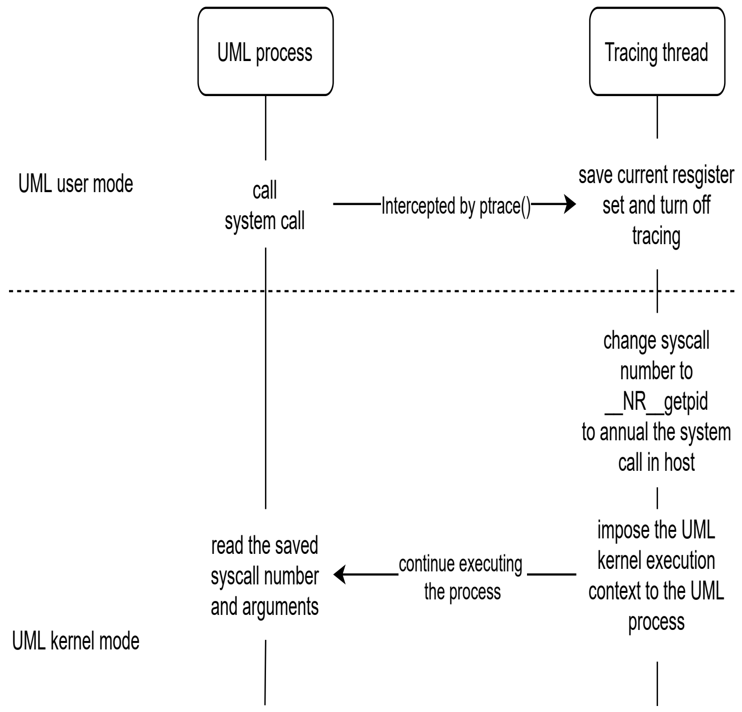

UML process
Tracing thread
UML user mode
call system call
Intercepted by ptrace()
save current resgister set and turn off tracing
change syscall number to __NR__getpid to annual the system call in host
impose the UML kernel execution context to the UML process
continue executing the process
read the saved syscall number and arguments
UML kernel mode

Figure 2.3: The control flow of handling system call in UML

User-mode Linux

In this section, we will introduce how does UML provide a way to simulate the Linux kernel in the user space using its own functions and intercept the system calls using *ptrace*(). The similar idea will be applied to the approach in the next chapter.

UML is an old project used to port a Linux kernel to the userspace. It enables multiple virtual Linux kernel-based operating systems to run as a regular Linux process in the host Linux's user sapce. This can be helpful to the system developers as it provides a nice way to develop and debug the kernel as well as to make several interesting applications possible for Linux ([Dik06]).

Implementation-wise, UML treats Linux as a platform to which the kernel can be ported. All the user-level code can run natively on the processors without any instruction emulation overhead, and UML implements critical OS services by leveraging the

Linux system calls without any modification of the host kernel.

System Call

System calls are the critical interfaces for the user-mode processes to request OS services, and the invocation of the system call will lead to a context switch from the user-mode to the kernel-mode. However, hardware platforms instead of Linux provide the mechanism of switching between the user-mode and the kernel-mode and enforing the lack of privileges in the user-mode. Hence, UML constructs such mechanism using the *ptrace* system call in Linux. It also implements a separate thread for tracing all the other threads via *ptrace*. This thread is notified when a thread is entering or leaving a system call, and has the ability to arbitrarily modify the system call and its return value. This capability is used to read out the system call and its arguments, annull the system call, and divert the process into the UML user space kernel code to execute it.

With such design, UML can distinguish whether a process is in the UML kernel-mode or in the UML user-mode. Those two modes are emulated in the UML. If the process is the UML user-mode, then it is beting traced by the tracing thread, and if it is the UML kernel-mode, it will not be traced. The tracing thread will also intercept all the signals and system calls that the running process issues, and then it will suspend the execution of the running process and switch the state of the running process into the UML kernel-mode by not tracing it. After that, the tracing thread will replace the register which contains the system call number with the system call number of *getpid* to anull the original system call in the host kernel. Next the tracing thread will save the process's registers into a data structure and load a new state to it so that it can invoke the system call handler. Once this is done, the ssytem call handler will read the system call numbers and the arguments, and then execute the such system call and let it be handled by the host kernel. After the system call finishes, the process will save the return value in tis register and the tracing thread will resume the process's routine by restoring the other saved registers of the process and continue tracing its system calls. At this satge, the process is back to the UML user-mode again (figure 2.3).

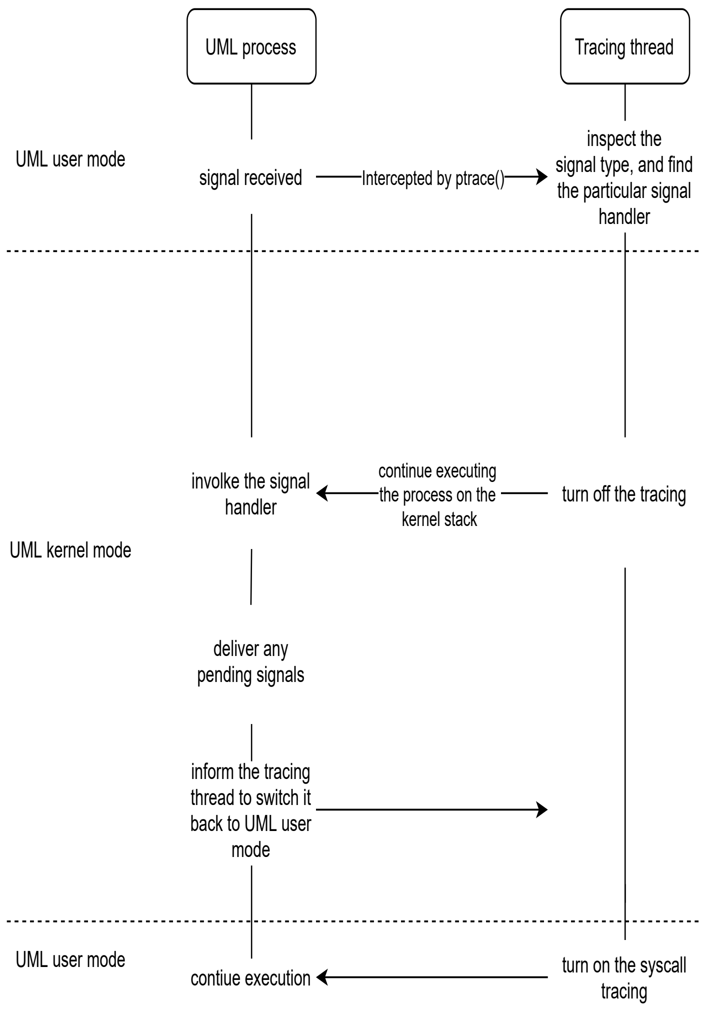

UML process
Tracing thread
UML user mode
signal received
Intercepted by ptrace()
inspect the signal type, and find the particular signal handler
invoke the signal handler
continue executing the process on the kernel stack
turn off the tracing
UML kernel mode
deliver any pending signals
inform the tracing thread to switch it back to UML user mode
UML user mode
contiue execution
turn on the syscall tracing

Figure 2.4: The control flow of handling signal in UML

Trap and Fault

UML not only virtualizes system calls but also implements several other Linux OS services and functionalities. For example, on physical machines, traps are caused by some piece of hardware like the clock, a device, or the memory management hardware forcing the CPU into the appropriate trap handler in the kernel. UML implements trap with Linux signals. The timer interrupt is implemented with the $SIGALRM$ and $SIGVTALRM$, I/O device interrupts with $SIGIO$ and memory faults with $SIGSEGV$.

The UML kernel declares its handlers for these signals. These handlers must run in the UNL kernel-mode with system call interception off (figure 2.5). The first is done by registering the handler to run on an alternate stack, the UNL process kernel stack, rather than the process stack. The second is accomplished by the handler requesting that the tracing thread turn off system call tracing until it is ready to re-enter the UML user-mode.

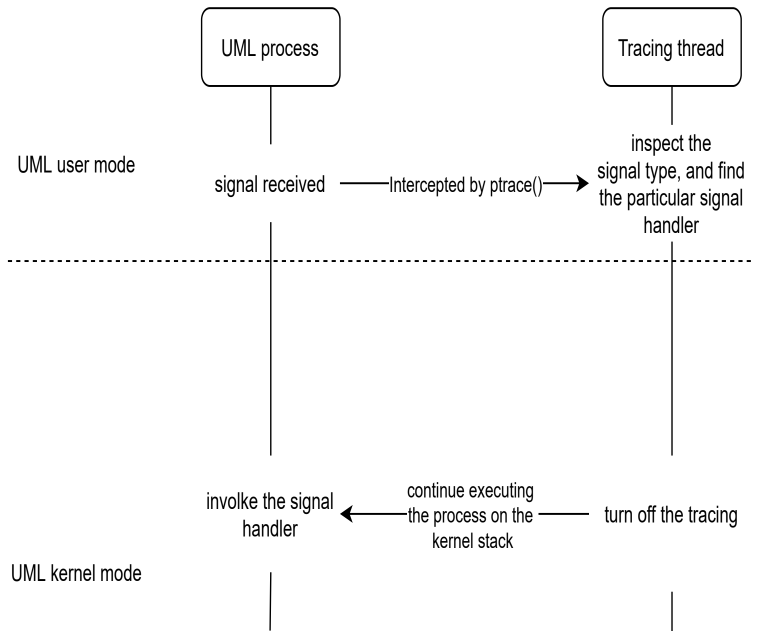

UML process
Tracing thread
UML user mode
signal received
Intercepted by ptrace()
inspect the signal type, and find the particular signal handler
involke the signal handler
continue executing the process on the kernel stack
turn off the tracing
UML kernel mode

Figure 2.5: The control flow of handling signal in UML

Interrupt

Since the interrupt happens asynchronously, UML implements external interrupt and timer interrupt using signals. More specifically, UML implements the timer interrupt using $SIGALARM$ or $SIGVTALARM$ depending on whether the interrupted process is idling or not. Both two signals will be sent to the process when the time limit specified in a call to a preceding *alarm* setting function (such as *setitimer*) elapses. However, *SIGALRM* is sent when real or clock time elapses, and the *SIGVTALRM* is sent when CPU time is used by the process elapses. Hence, if the process is idling, the *SIGALARM* will be sent. Otherwise, *SIGVTALARM* will be sent. On the other hand, the *SIGIO* is used to implement the external interrupt which is generated by devices because the *SIGIO* will be generated as long as a device receives data. UML uses *select* to figure out which file descriptor is expecting an I/O event. IRQ handlers will be invoked to handle the timer interrupt and the external interrupt once associated signals are received by the process.

Context Switching

A context switch in UML involves suspending the current running UML process and executing the other UML process. Giving each UML process its own address space avoids a total remapping of the process's address space when the context switch happens. However, it's necessary to remap those pages of the switching-in process that have been unmapped when it was switched out to ensure consistency. UML marks those pages using specific bits in the page table entry when a modification in the page table entry happens and clears those bits when the host updates the mapping of the pages. Besides handling mapping pages UML also handles the pending I/O events when the context switch happens. UML implements the device interrupt with *SIGIO* which can be queued to a UML process. Hence, UML will update the *SIGIO* registration from the switch-out process to the switch-in process to ensure the system is consistent.

Virtual memory

UML emulates the virtual memory by the kernel, and the process's virtual memory is implemented as a physical memory-sized file mapping into their address spaces. However, switching from the UNL user-mode to the UML kernel-mode doesn't automatically switch the address spaces as they are the same from the host's view. UML resolves the conflicts with process memory by placing the kernel text and data in areas that processes are not likely to use. In UML, a memory fault causes a *SIGSEGV* to be delivered to the process. The segmentation fault handler does the necessary checking to determine whether it was a UML kernel-mode or UML user-mode fault, and in the user-mode case, whether the page is supposed to be present or not. If the page is supposed to be present, then the kernel's page fault mechanism is called to allocate the page and read in its data if necessary. The segmentation fault handler then maps in the new page with the correct permissions and returns. On the other hand, if the fault was not a legitimate page fault, then the UML kernel either panics or sends a *SIGSEGV* to the process depending on whether the fault was in UML kernel-mode or UML user-mode.

Host filesystem access

UML also implements a virtual file system called *hostfs* which translates the directly into the functions in the host's *libc*. Hence, it provides direct access to the host filesystem.

UML was a nice project showing a novel way to use Linux interfaces to implement itself, also it demonstrated that porting a kernel in the userspace can be beneficial to both kernel and application development. From this thesis's perspective, UML gives us the hint of how to leverage *ptrace* to intercept the system call from the process being traced as well as how to simulate the kernel in the userspace.

Chapter 3

Approaches

At this stage, we are going to propose a software development approach to tackle the problem. Running seL4 applications in Linux requires introducing an extra compatibility layer between the application and the Linux OS. Hence we are going to propose two approaches to achieve this: the API layer emulation approach and the ABI layer emulation approach.

3.1 API Layer Emulation Approach

In the original seL4 system, a userland seL4 client application can request OS services from another seL4 server application which also runs in the userland through IPC mechanisms provided by the seL4 kernel. The seL4 system call library provides abstractions to those IPC interfaces.

However, if those seL4 userland applications run on top of Linux OS, then directly using the original seL4 system call libraries will not work due to the different semantics between seL4 and Linux. To solve that problem, Cygwin gives a good idea. In Cygwin, it replaces the original system call libraries with a custom implementation to achieve compatibility between POSIX and Windows. Hence, we can provide a custom system call library that exports the same interfaces as the original seL4 system call libraries to seL4 userland applications, but we will use Linux IPC mechanisms as a replacement instead of invoking the seL4 IPC directly.

3.1.1 Design Model

Figure 3.1 shows the model of the API layer emulation approach.

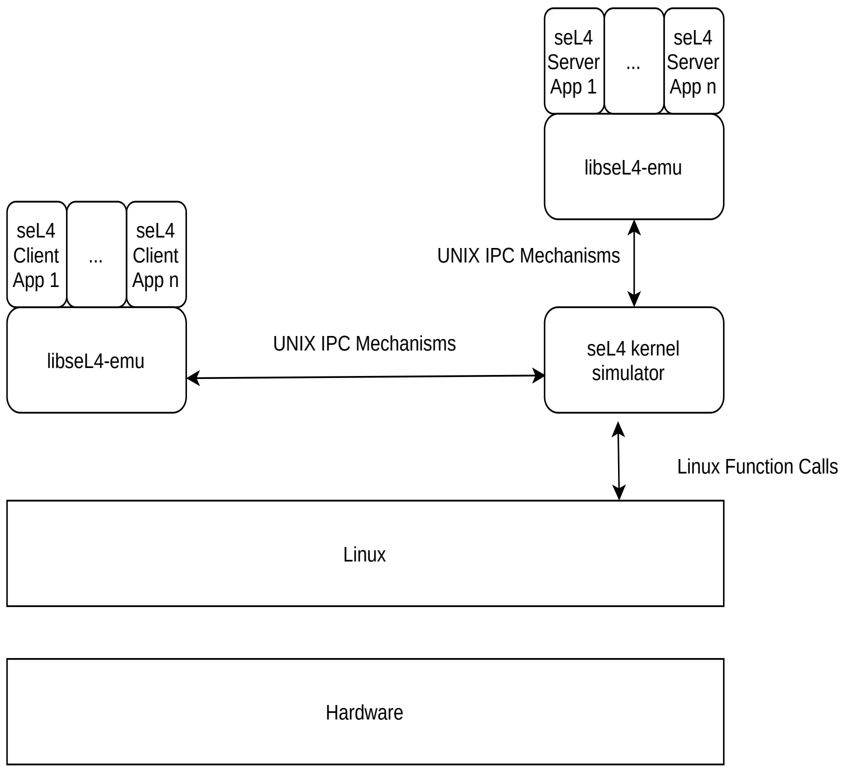


Figure 3.1: The model of API layer emulation approach

In general we need to develop two components in this model.

1. An seL4 emulation library
2. An seL4 kernel simulator

The seL4 emulation library is a compatibility layer named libseL4-emu in the figure which can be linked to other seL4 applications. Therefore, the seL4 applications will invoke the emulated seL4 system calls instead of the real original seL4 system calls. And the libseL4-emu library will dispatch those system call requests to the seL4 kernel simulator.

The seL4 kernel simulator is a regular user-level Linux application. Hence the emulated seL4 applications can communicate with the seL4 kernel simulator using Linux IPC, such as UDS or shared memory. The seL4 kernel simulator mainly provides the seL4 semantics and management of capability space, IPC mechanisms, scheduling context management, etc.

In this model, all the seL4 applications and the seL4 kernel simulator are just regular Linux processes running at the user-level. The seL4 applications act as clients, communicating with the server which is our seL4 kernel simulator.

3.1.2 Advantages and Disadvantages

The API layer emulation approach has several advantages. First of all, it is relatively easy to implement regarding the complexity of implementation.

Secondly, since simulating the seL4 IPC mechanisms uses Linux IPC mechanisms won't introduce too much overhead, the performance should be relatively good compared to fully emulating a machine using QEMU. But it's hard to give an intuitive result of comparing this approach with the hardware virtualization approach using QEMU without further detailed measurement. Although the hardware virtualization will introduce the overhead due to the word switching, the API layer emulation approach will require at least four context switches between the seL4 client application, the seL4 kernel simulator, and the seL4 server application.

Whatsmore, this approach is ISA independent. In the model, all the seL4 applications and the seL4 kernel simulator run at the user-level, and the seL4 system call emulation layer provides the interfaces. From the perspective of seL4 applications, everything below the system call emulation library is abstracted away. Ideally, those applications shouldn't notice that they are no longer running on top of the real seL4 kernel and communicate with the seL4 simulator server through the emulated seL4 system calls. Linux provides hardware abstractions, hence, this approach focuses on implementing the correct semantics of the real seL4 kernel.

However, the main disadvantage of the API layer emulation is that it is not binary compatible. Furthermore, the emulation will work only if the seL4 applications are are programmed to use seL4 system call emulation library instead of invoking seL4 system calls directly. Hence, we have to obtain the source code of the application first and recompile the source with the emulation library every time we want to use the emulation.

3.2 ABI Layer Emulation Approach

In the former approach, we have introduced the API layer emulation, which is not binary compatible, while in this approach, we are going propose a method to achieve binary compatibility. However, the main challenge is redirecting the system calls issued by the seL4 applications without modifying the source code. Both Wine and UML projects introduce different approaches of providing compatibility at the ABI layer to achieve such a goal. But in Wine's architecture, the system call libraries are implemented as a shared library, while the seL4 build system uses a statically linked method. Therefore, we will implement the ABI layer emulation approach using the idea given by the UML project, which is using the *ptrace*() system call in Linux.

3.2.1 Design Model

Figure 3.2 shows the model of the ABI layer emulation approach.

In the figure, mainly we have two components:

1. An seL4 application monitor

2. An seL4 application binary loader

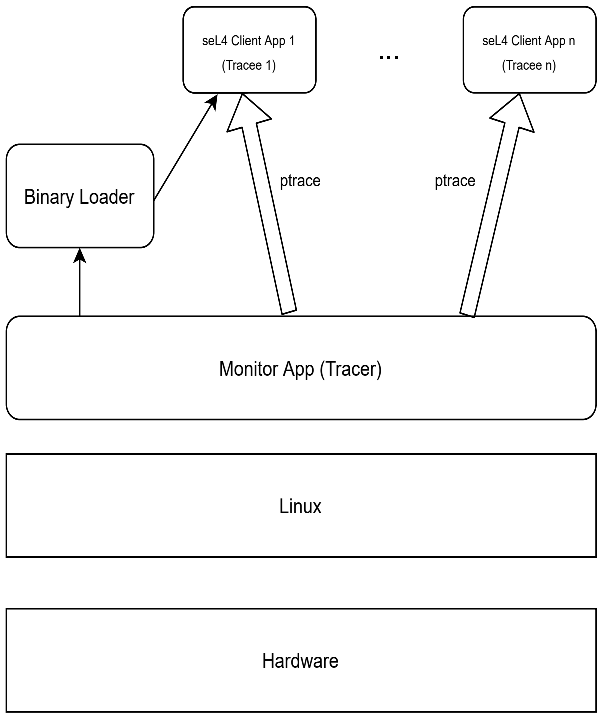


Figure 3.2: The model of ABI layer emulation approach

In the rest of the article, we will call the seL4 application monitor the *monitor*, and the seL4 application binary loader the *binary loader* for simplicity. The main challenge in this approach is that since the seL4 application is an unmodified binary code, it will invoke the real seL4 system calls. In that case, we have to intercept the system calls and observe what is the seL4 application requesting then serve the system call by ourselves according to the real seL4 semantics. To achieve this goal, we are going to use the *ptrace*() system call in Linux.

ptrace() allows a tracer process to observe and control the execution of the other process. The ”tracee” process can be attached to the ”tracer” process, so that the ”tracer” can modify the ”tracee” process’s memory, registers, etc. Also, the *ptrace*() semantics allow us to establish a one-to-many relationship, which means we can have several ”tracees”, but can only have one ”tracer”. In this approach, we can leverage such a feature to intercept the system call invocation from the seL4 applications.

(TODO: Why the loader is requireed and what does the loader do?)

In the model described above, the monitor acts as our ”tracer”, and the seL4 applications are our ”tracees”. The monitor is also a seL4 kernel simulator providing the same seL4 semantics and kernel mechanisms as the real seL4 kernel does. The emulation starts from the monitor forks a binary loader process that can load the seL4 application’s ELF file into the memory and set up the appropriate memory regions for it. Next, the binary loader will block and notify the monitor that the initialization has finished. The monitor can use *ptrace*() to control the seL4 application. Finally, the seL4 application can continue executing after the *ptrace*() process finishes. Now, every time the seL4 application invokes a seL4 system call, the monitor will be notified and suspend the execution of the seL4 application, then the monitor will inspect the system calls requests and do appropriate actions to serve them.

3.2.2 Advantages and Disadvantages

The main advantage of this approach is that the kernel ABI approach is binary compatible. We can directly run an unmodified binary code of seL4 applications without recompilation. However, using the ABI layer emulation will introduce overhead. This is because every time the seL4 application invokes a seL4 system call. The monitor will use *ptrace*() to intercept it and modify the arguments, return value if necessary, and the *ptrace*() itself is a system call, which requires trapping into the kernel. The trade-off here is leading to a penalty of the performance. Besides, *ptrace*() is usually used to implement the Linux debuggers, and it only allows on ”tracer” to exist at any time. Therefore, we have to provide our own debugging interfaces in some way, which can introduce implementation complexity.

3.3 Comparison Between API and ABI Layer Emulation

Table 3.1 listed the trade-offs of implementing the API and the ABI layer emulation approaches. The API layer approach is relatively easier to implement as we don't need to develop the binary loader and the debugging interfaces. The API layer emulation can fully leverage the native Linux debugging tools such as GDB and LLDB. In terms of portability, the API layer approach is ISA compatible as all the implementation is at the seL4 system call library level and doesn't involve the ISA dependent code. However, the API layer approach doesn't achieve binary compatibility as it requires recompiling the source code to work, while the ABI layer emulation approach is fully binary as we intercept the system calls using *ptrace*().

Last but not least, in terms of the performance, the API layer approach should be better than the ABI layer approach due to the internal implementation. As mentioned before, intercepting system calls and modifying the system calls' arguments or return values require extra context switches to the Linux kernel, which introduces overhead.

Table 3.1: Comparison between API and ABI layer emulation

Approach	Implementation Complexity	Portability	Binary Compatibility	Debugging Support	Kernel IPC Performance
Syscall Library API emulation	Intermediate	Good, as the solution is ISA portable	No	Can use native Linux debugging tools	First
Kernel ABI emulation	Difficult	No, if the ABI of the application is different from the host's, it won't work	Yes	Extra debugging interface required	Second

Chapter 4

Evaluation

In this chapter, we will propose some evaluation approaches regarding to both API layer emulation and ABI layer emulation approaches. Despite the fact that the real implementation and the benchmark haven't been finished yet, we can still set up some criteria in terms of the implementation in both a qualitative aspect and a quantitative aspect.

4.1 Qualitative Evaluation

One way to evaluate whether the implementation of the emulation approaches is successful or not is to test the emulation framework with some existing seL4 applications. In this project, we are going to use applications from the following source:

4.1.1 seL4 Tutorials

The seL4 tutorials contain several trivial seL4 applications used for demonstration and education purposes.

Those applications mainly focus on demonstrating seL4 features and API usage including printing "Hello World", simple IPC message passing, setting up runnable threads in seL4 userland and so on. Those applications can be used to use to test each of the emulated seL4 features separately.

4.1.2 seL4Test

The seL4Test contains several unit tests that focus on low-level kernel APIs. As those tests were used to prove the correctness of the implementation of the real seL4 kernel and some seL4 helper libraries, therefore, we can use those tests to evaluate the implementation of the seL4 kernel simulator. However, since the seL4 kernel simulator runs in the Linux userspace, it doesn't have the access to the hardware and performs privileged operations. We don't expect the kernel simulator can pass all of the tests. We will focus more on testing critical functionalities, such as the IPC mechanisms, cspace management, vspace management and scheduling and skip those tests that are not quite important to the emulation project or those are complex or even impossible to emulate correctly such as cache tests, multicore tests and so on.

4.1.3 Simple operating system project

The simple operating system is a simple multitasking operating system running on the seL4 kernel which has an interactive shell, and several subsystems including I/O, memory management, process management and so on. Since it has a relatively complex structure and more comprehensive functionalities implemented with seL4 semantics, emulating it using the emulation approach can be a nice way to both verify and demonstrate the emulation approaches function well.

4.2 Quantitative Evaluation

From the qualitative perspective, we are going to evaluate the two approaches via microbenchmark and macrobenchmark. Table 4.1 is the expected quantitative bench-

Table 4.1: Expected success criteria

	syscall API emulation (A1)	kernel ABI emulation (A2)
On the emulated hardware by QEMU (B1)	A1 should be significantly faster than B1 because no hardware emulation is involved	A2 should be significantly faster than B1 but slower than A1 due to syscall interception
Native hardware running seL4 (B2)	A1 should be less than an OoM slower than B2 because seL4 is highly optimized	A2 should be significantly slower than B2 due to syscall interception

marking outcome.

4.2.1 Microbenchmark

(TODO: I would have hoped for more analysis here. These costs can be estimated, at least with the help of some very simple micro benchmarking of Linux syscalls -¿ define the base line performance in Linux and seL4)

The microbenchmark mainly focuses on evaluating the performance of each emulated system call. We can measure the number of system calls that the seL4 application invokes per second. The higher the result, the better the performance. Besides, to make the result more representative, we should measure the result several times and make a statistical analysis.

The ideal expectation is listed in the table 4.1. In general, we expect the API emulation approach will not introduce too much overhead, and the result should be slower compared to the result of running the same application on the native seL4 kernel. In the worst case, the overhead's upper bound shouldn't exceed an order of magnitude. While comparing to the result of using ABI layer emulation, hardware virtualization, or emulation approach, the API layer emulation should be faster. This is because, in the API layer approach, all the real seL4 system calls are emulated, which can save some of the overhead of trapping into the kernel.

On the other hand, the ABI layer emulation approach should be faster than using any hardware emulation or virtualization, but slower than the API layer emulation approach, because intercepting system calls will introduce relatively high overhead.

4.2.2 Macrobenchmark

The macrobenchmark can be commenced under two circumstances and the result will be the measurement of the total system calls' elapsed time. On one hand, we can simulate a situation with a heavy system call workload. The result can emphasize the expectation that using either API or ABI layer emulation approach will not introduce unacceptable overhead. On the other hand, we are going to simulate another situation with a low system call workload, the result emphasizes that using API or ABI emulation approach is more performant than using hardware virtualization or emulation approaches such as QEMU.

Chapter 5

Project Plan

So far the work of this project has been done is the background research of the related projects as well as construct the basic solution model in a theoretical aspect. This report summarizes the related projects' ideas that can be borrowed in this project. QEMU is an existing solution for emulating the entire foreign system. While Cygwin and Wine project give the idea that we can avoid hardware virtualization of emulation to execute applications that target foreign systems by adding an extra compatibility layer at API or ABI layer. And the final project UML introduced an approach to port Linux kernel into Linux user space and intercept the system calls from the userspace by using *ptrace*(). With those ideas, we purposed the implementation of the project using the API layer emulation approach by modifying the system call libraries and using the ABI layer emulation approach by using *ptrace*() system calls in Linux. The design and implementation details are not yet finalized at this stage. It's only a rough prototype that could potentially work.

5.0.1 Timeline

Below is the timeline for future thesis research work.

1. implement the seL4 simulator server, and implement the basic framework for the

API. This will take roughly 2 weeks, and will be done during the break of the term 21T1

2. Refine the emulation framework, complete seL4's critical kernel functionalities, and pinpoint parts that are hard to emulate. At this stage, the emulation framework should be mostly functional. The plan is to achieve this stage before week3 in term 21T2.

3. Evaluate the API emulation framework using the methods listed in this report and modify the implementation if necessary. We should complete evaluating the API emulation approach two weeks before the week5 of the term 21T2.

4. Depending on the time left in the term 21T2. The next stage will be either implementing the ABI emulation approach or refining the theoretical part of the ABI emulation approach and adding details of implementation in the Thesis Report as future work.

5. Preparing for the final demonstration and the presentation.

Bibliography

[Cyg] A brief history of the cygwin project. `https://cygwin.com/cygwin-ug-net/brief-history.html`. Accessed: 2020-04-25.

[Dik06] Jeff Dike. *User Mode Linux.* Prentice Hall, 1rd edition, 2006.

[HKA20] Gernot Heiser, Gerwin Klein, and June Andronick. seL4 in Australia: From research to real-world trustworthy systems. *Communications of the ACM*, 63:72–75, April 2020.

[KAE$^+$10] Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an operating-system kernel. *Communications of the ACM*, 53(6):107–115, June 2010.

[KEH$^+$09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *ACM Symposium on Operating Systems Principles*, pages 207–220, Big Sky, MT, USA, October 2009. ACM.

[LMAH18] Anna Lyons, Kent McLeod, Hesham Almatary, and Gernot Heiser. Scheduling-context capabilities: A principled, light-weight OS mechanism for managing time. In *EuroSys Conference*, Porto, Portugal, April 2018. ACM.

[Wik21a] Wikipedia contributors. Cygwin — Wikipedia, the free encyclopedia, 2021. [Online; accessed 26-April-2021].

[Wik21b] Wikipedia contributors. Qemu — Wikipedia, the free encyclopedia, 2021. [Online; accessed 26-April-2021].

[Wik21c] Wikipedia contributors. Wine (software) — Wikipedia, the free encyclopedia, 2021. [Online; accessed 26-April-2021].