2021 T2 Thesis B seL4 API/ABI Emulation on Linux

(Design and implementation in the emulation framework)

Student: *Jiawei Gao*

Supervisor: *Gernot Heisor & Axel Heider*

Produced: 09 Aug 2021

Content Layout

Project Recap

What is the project about?

Implement a **compatibility layer** so that we can directly run seL4 applications on Linux without machine emulation or virtualization. (Here we define seL4 applications are applications that are linked to seL4 libraries and use seL4 functionalities. In the seL4 world, they are called seL4 threads) In the rest of the slides, term clients refer to seL4 applications aka seL4 threads. Server refers to the kernel emulator.

What are the benefits?

Developing:

- an easy way for rapid prototyping an application in Linux environment.
- use Linux tools directly. (e.g. GDB, valgrind, strace, etc.)
- Access to Linux system's rich input and output (e.g. Files, Networking etc.). Instead of developing a device-specific driver, we can implement a special seL4 sever application that has direct access to Linux I/O.

Self-learning:

- Learning a lot of things during the project

The expected outcome

With the emulation framework, we should achieve at least **source compatibility**, meaning that after recompiling the seL4 application source code. We can run the application successfully. Moreover, we can also achieve binary compatibility. However, this is not implemented at the moment and will be discussed in the later slides.

Analysing "Hello World" Roottask

Where to start with?

Let's start with inspecting a roottask printing "Hello World!". Notice that the following discussion will assume we use an x86_64 platform due to the current implementation. But we can extend the framework to be architecture independent without many changes.

What does a helloworld roottask need?

A runtime library

Set up the environment for an seL4 thread. (e.g. TLS region, POSIX syscall API redirection, IPC Buffer, etc.)

- seL4runtime

C library

Provide POSIX programming interfaces for the developer. (e.g. printf, open, read, etc.) Also the modified muslibc will redirect the POSIX syscalls to the handlers provided by seL4muslsys.

- musllibc
- seL4musllibcsys

seL4 system call library

Provide seL4 system call APIs.

- libseL4

seL4 kernel

Provide seL4 kernel functionalities and handle seL4 threads' requests.

- seL4 kernel

Analysing "Hello World" Roottask (cont)

How many seL4 syscalls needed by the helloworld roottask?

- **`seL4_SetTLSBase`** (set up the TLS region)
- **`seL4_DebugNameThread`** (needed if built with debug configuration)
- **`seL4_DebugPutChar`**

How to redirect those syscalls?

Dive deeper into how the above APIs are implemented. We can find they are actually wrappers around the raw seL4 syscalls. (e.g. x64_sys_send_recv() in syscall_syscalls.h) Those raw seL4 syscalls are wrapper around the ASM syscall instructions and they are **architecture dependent**.Other high level APIs provided by libseL4 eventually go here.

Hence we can redirect the system calls with the **minimal** modification in the original code by modifying the following raw syscall wrappers (Take x86_64 as an example).

`x64_sys_send`	invokes	**`seL4emu_sys_send`**
`x64_sys_reply`	invokes	**`seL4emu_sys_reply`**
`x64_sys_send_null`	invokes	**`seL4emu_sys_send_null`**
`x64_sys_recv`	invokes	**`seL4emu_sys_recv`**
`x64_sys_send_recv`	invokes	**`seL4emu_sys_send_recv`**
`x64_sys_nbsend_recv`	invokes	**`seL4emu_sys_nbsend_recv`**
`x64_sys_null`	invokes	**`seL4emu_sys_null`**

Implementation Overview at Client Side

Steps to develop client side emulation library

- Build system
  Reuse the powerful seL4 build system to integrate our emulation library and generate headers based on the user configurations.
- Another C library
  Develop a library that will be used by the emulation library and provide minimal Linux syscall wrapper functions and other utility functions.
- Client runtime library
  Modify the current sel4runtime to do the initial setup for the emulation library
- Emulation IPC library
  Use UNIX domain socket to passing the seL4 syscall paramters, such as syscall number, message registers, etc.

Minimusllibc Implementation

A C library used by the emulation library

The emulation library requires using Linux system call wrappers as well as other functionalities to make developing easier. However, the current musllibc is modified and used by seL4 system, so we can't use those APIs provided by musllibc. To do that I tried:

- Link glibc on the host: can't link glibc on the host as the seL4 application are statically compiled and linked. ✗
- Load glibc at runtime: the dynamic library provides **`dlopen`**, **`dlsym`** and so on depends on glibc itself. And porting those functions are complicated. ✗
- Change symbol names: the current working approach is quite simple. We ported a part of the musllibc code with symbol names changed and linked with the emulation library. (named **minimusllibc** :-)) ✓

The current minimusllibc provides:

- socket related APIs such as **`mini_socket, mini_accept, etc.`**.
- I/O APIs such as **`mini_write, mini_read`**, etc.
- memory mapping APIs such as **`mini_mmap`**, etc.
- signal APIs such as **`mini_sigaction, mini_sigstack`**, etc.

The minimusllibc is **not thread safe** for the simplicity will explain in later sides.

seL4 Emulation Runtime Library Implementation

When the roottask started by the seL4 kernel, it will do some environment setup beforing entring the main function. In the original seL4runtime the most important functionalities are:

- Obtain the bootinfo: the seL4 kernel passes the bootinfo as an argument to the roottask, including the first usable slot in cnode, ipc buffer address, etc.
- Set up the TLS region: the TLS region is used for storing IPC buffer's pointer and per seL4 thread error, CAmkES uses TLS for bookkeeping as well.

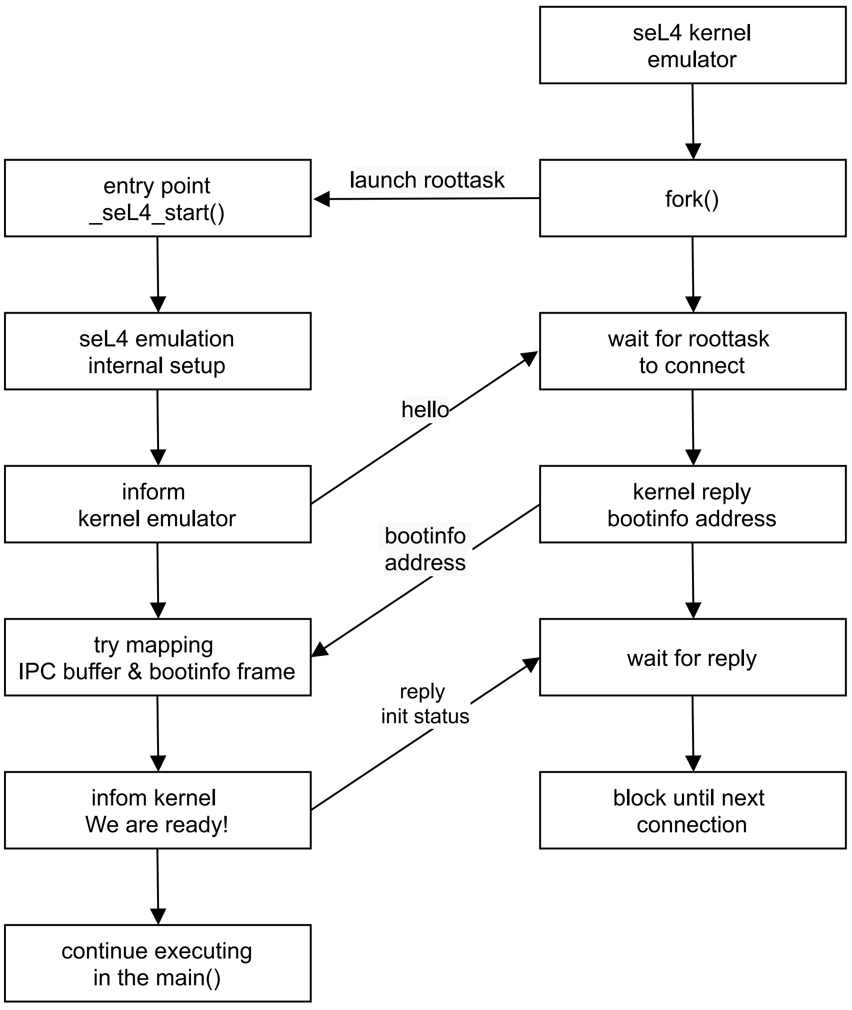
Reimplementation:

- Obtain the bootinfo: instead of passing as an function argument, we use map a share memory to access the bootinfo frame.
- Setup TLS: instead of kernel set up the TLS region for us, we setup the TLS region by ourselves. (e.g. using **`FSGSBASE`** instruction family on x86 if avalaible, otherwise use Linux process control syscalls)
- Emulation library internal setup: specific routine of the emulation library, including handshaking with the kernel emulator, mapping the IPC buffer frame and bootinfo frame using share memory, as well as setup the **`SIGSEGV`** signal handlers. (Explain in the later slides)

Emulation Runtime Library Implementation
(cont)

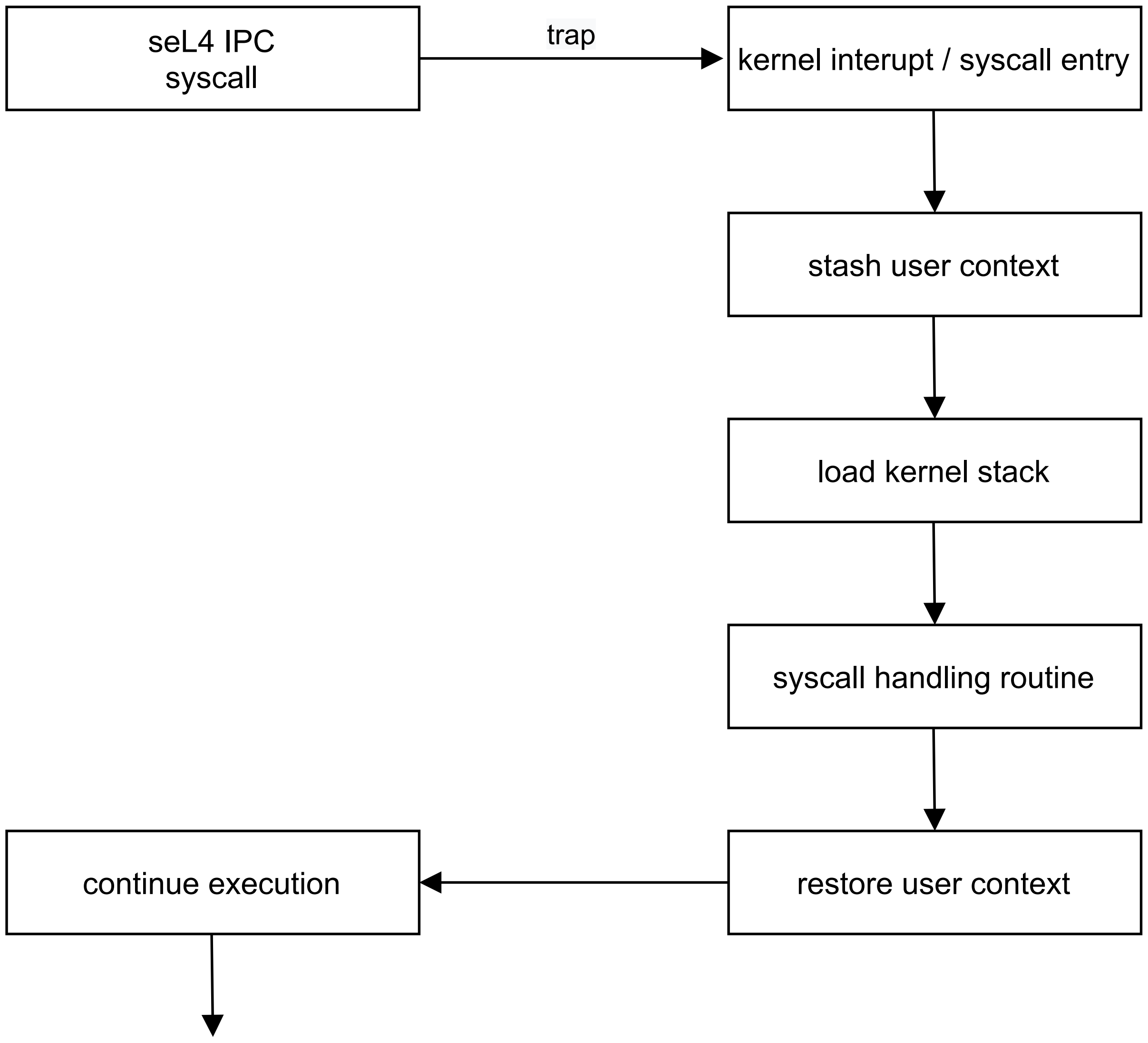
Since the emulation framework needs to do some intial setup before entring the main funtion, hence, most of the work will be done in the runtime library.

- the kernel emulator and the rootask will do a handshaking.
- the roottask will do an intial set up. (map IPC buffer, boot info, set up signal handler)
- roottask will inform the kernel emulator about the initialization status.


seL4 kernel emulator
fork()
launch roottask
entry point _seL4_start()
seL4 emulation internal setup
wait for roottask to connect
hello
inform kernel emulator
kernel reply bootinfo address
bootinfo address
try mapping IPC buffer & bootinfo frame
wait for reply
reply init status
infom kernel We are ready!
block until next connection
continue executing in the main()

seL4 IPC System Call Flow

The following diagram shows the execution flow when the seL4 IPC happens. The seL4 thread invokes the syscall instruction and traps into the kernel, and the kernel will store the user context and invokes the syscall handlers. After the handler finishes, the kernel will load the user context and return control back to the userland.



seL4 IPC Internals

What message do we need to pass?

Assume we are still on x86_64. By looking at the original seL4 syscall implementation. For example **`x64_sys_send_recv`**:

```
x64_sys_recv(seL4_Word sys, seL4_Word src, seL4_Word *out_badge,
        seL4_Word *out_info, seL4_Word *out_mr0, seL4_Word *out_mr1,
        seL4_Word *out_mr2, seL4_Word *out_mr3, seL4_Word reply)
{
        register seL4_Word mr0 asm("r10");
        register seL4_Word mr1 asm("r8");
        register seL4_Word mr2 asm("r9");
        register seL4_Word mr3 asm("r15");
        MCS_REPLY_DECL;
        asm volatile(
                "movq   %%rsp, %%rbx    \n"
                "syscall                \n"
                "movq   %%rbx, %%rsp    \n"
                : "=D"(*out_badge),
                "=S"(*out_info),
                "=r"(mr0),
                "=r"(mr1),
                "=r"(mr2),
                "=r"(mr3)
                : "d"(sys),
                "D"(src)
                MCS_REPLY
                : "%rcx", "%rbx", "r11", "memory"
        );
}
```

seL4 IPC Internals (cont)

According to the x86 calling conventions and the seL4 semantics

`RDI`	stores	**`syscall number`**
`RSI`	stores	**`message info`**
`RDX`	stores	**`capability pointer`**
`R10`	stores	**`message reigister 0`**
`R8`	stores	**`message reigister 1`**
`R9`	stores	**`message reigister 2`**
`R15`	stores	**`message reigister 3`**
`R12`	stores	**`reply`** (only used in MCS configuration)
`IPC Buffer`	stores	**`Other message registers`**

According to seL4's design, the only **first four** message registers will be passed using **physical registers** and the rest will be placed in the IPC Buffer. We will strictly follow this semantic so we need to design the IPC emulation protocal. (next slide)

IPC Emulation Protocol

To emulate the seL4 IPC syscalls, we implemented two different protocols at the moment. One follows the seL4 IPC syscall semantic. The other one is used internally by the emulation framework. (e.g. Handshaking with the kernel in runtime setup routine). The IPC message layout is as follow:

Tag	ID	Reserved	Body
Word Size	Word Size	Word Size	Word Size * n_contextRegisters

Because in the real seL4 system, after traping into the kernel, all the user context registers are stashed in the tcb structure.
So we decided to pass an **`n * word size`** length message for each emulated IPC invocation, where **`n = n_contextRegisters`**. (**`n_contextRegisters`** is defined in the arch dependent **`registerset.h`**, on x86_64 it is **24**)

One challenge is how to deliver other message registers on the IPC Buffer? In the real seL4 system, as the kernel can accessing directly to the seL4 thread's IPC buffer. (explain in later slides). But for the emulation it's not quite easy to access the kernel emulator and the seL4 threads are running as different processes on Linux (explain in later slides), they have different address spaces. We need other process communication mechanisms.

- passing the entire IPC buffer on each emulated seL4 IPC invocation. (extra 8KB for a round trip of the IPC invocation, will be a horrible overhead for the IPC heavy applications!) ✗
- map the IPC Buffer as a shared memory. This approach is quite efficient and solved the problem quite well. As from the seL4 thread's view, it writes to the IPC buffer as usual,

however, the kernel emulator can access the seL4 thread's IPC Buffer directly as well. ✓

seL4 Thread Emulation Implementation

In the real seL4 system, the program execution are modeled as seL4 thread. For the emulation, we also have two implementation options, map each seL4 thread to a Linux user space **thread** or a Linux user space **process**.
Trade-offs of mapping to threads

- creation / deletion are relatively light weighted. ✓
- we can share the address spaces. ✓
- porting pthread from standard libc is complicated. ✗
- starting an entire program requires loading the ELF file into the memory first then jumping to the entry point. ✗
- dealing with concurrency issues. ✗
- handling signals can be complicated. ✗
- pthreads already use TLS, which conflicts the seL4 system. ✗

Trade-offs of mapping to process

- much fewer concurrency issues. ✓
- handling signals is easy. ✓
- execution of the entrie program is easy. ✓
- no need to port pthread from libc. ✓
- process creation / deletion is relatively heavy. ✗
- different address spaces, requires explicit process communication mechanisms for sharing data. ✗

Memory Mapping Emulation Challenge

seL4 provides APIs to **explicitly** map a page. However, it's possible for a user to implements a virtual memory management server for handling the demand mapping, meaning that the actual mapping will be done on the **vm fault** rather than always mapping on the initial request. The current emulation library can handle both of the situations.

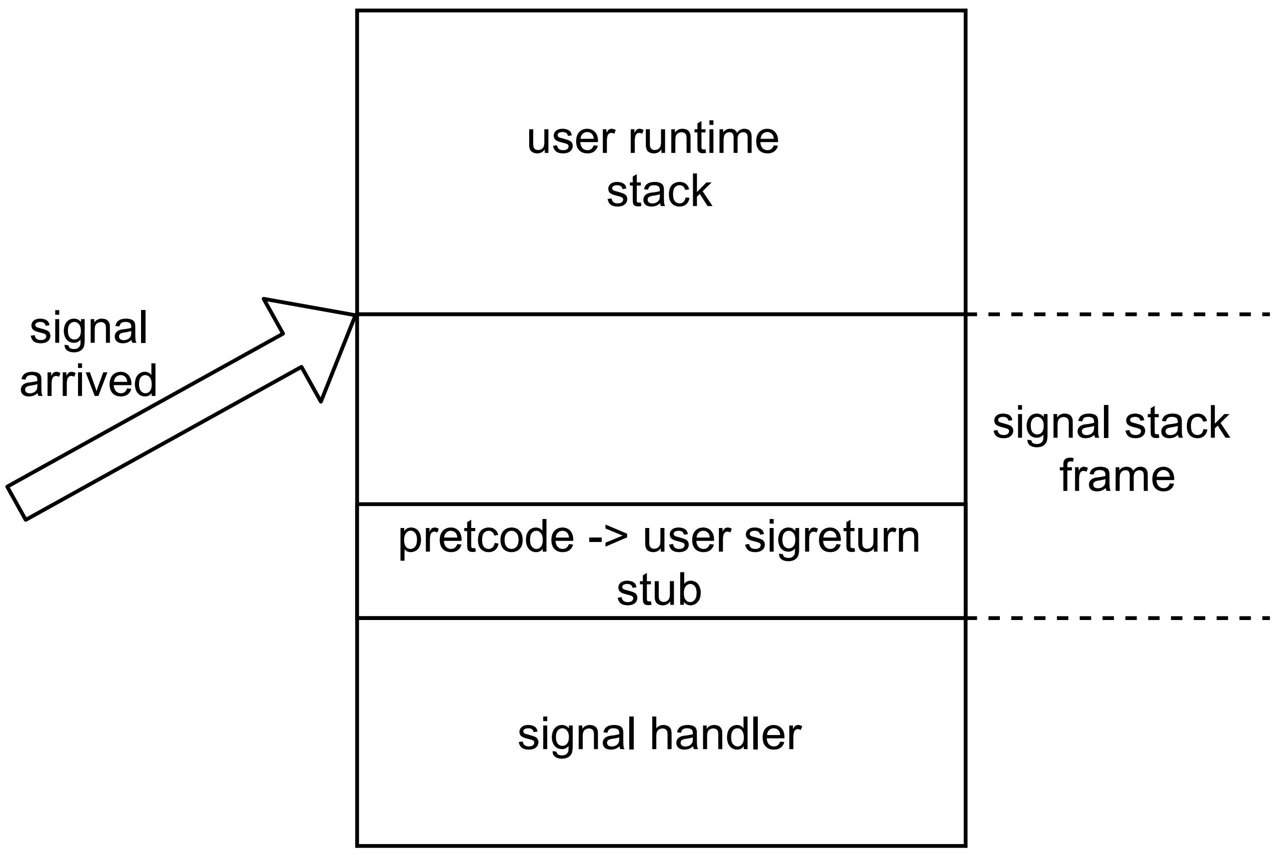
The first challenge is that the kernel emulator and the seL4 thread's running in different address spaces, hence, the kernel emulatior **can't** do mapping for the seL4 thread.

The second challenge is emulating the lazy mapping, so we don't map a page until we actually touch the page.

Remember that the invalid access of memory will generate an exception on the hardware and the exception handler will handle that. In case, we can model interrupts on seL4 using signals because siganls are software interrupts. Also, we have our old friend segmentation fault. (**`SIGSEGV`**). And we can set up a signal handler to emulate the exception handler. (more discussion on next slide)

Linux Signal Handler Recap

- When the signal is delivered to the current process, Linux already set up a signal stack for us. The default stack is placed below the current stack pointer's location.
- Linux kernel has assumptions about the layout of the signal handler frame. The lowest 8 byte points to the user signal restorer stub.
- Below the stack frame is the signal handler. Hence when the signal handler returns, it will automatically invoke the user signal restore stub and trap it back to the kernel again to ask the kernel to clean up and restore the original context when the signal happened.



Memory Mapping Emulation Implementation

What do we want to achieve

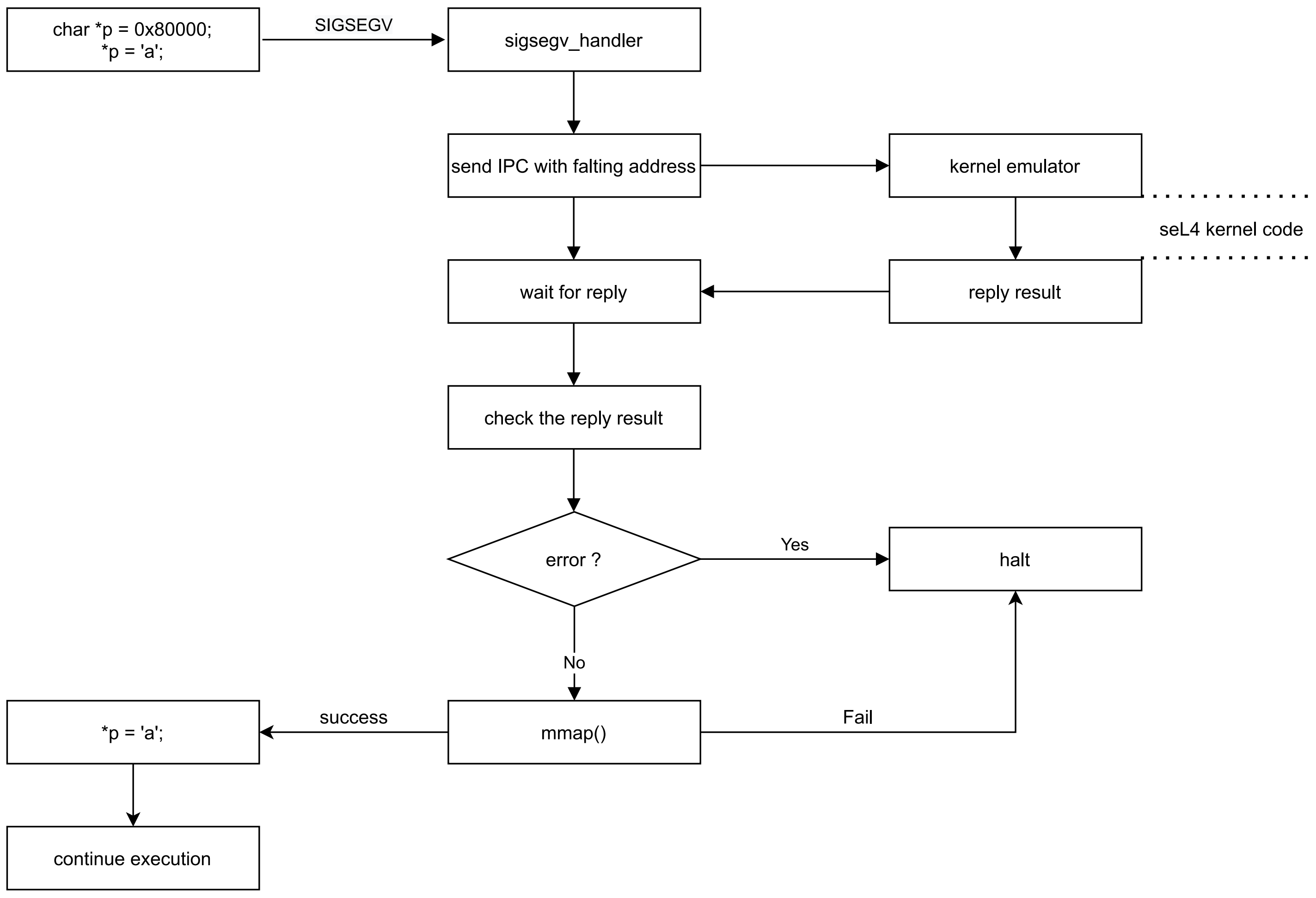
- The program execution will stop if the memory access is invalid and without any mapping.
- The program will continue execution after an explicit seL4 mapping invocation.
- Assume we implement a lazy mapping mechanism. The program will stop upon accessing the memory address and the vm fault handler will update the paging structure. Eventually, the faulting program restarts at the faulting instruction and continues.

How to implement?

- set up a signal handler during the runtime initial stage.
- the seL4 application tries to access an unmapped memory address.
- **`SIGSEGV`** triggered and invokes the siganl handler routine.
- the signal handler sends an IPC to the kernel emulator.
- the kernel emulator will verify the mapping and reply.
- the signal handler will check the reply. It will either map the faulting address by using **`MAP_FIXED`** then return back to the previous context or halt the program execution.

Memory Mapping Emulation Implementation (cont)

The following diagram illustrates the program flow of the memory mapping emulation. Here we assume *0x80000* is already mapped by the virtual memory management server. And the paging structure is updated already.


char *p = 0x80000;
*p = 'a';
SIGSEGV
sigsegv_handler
send IPC with falting address
kernel emulator
seL4 kernel code
wait for reply
reply result
check the reply result
error ?
Yes
halt
No
success
*p = 'a';
mmap()
Fail
continue execution

Client Side Emulation Library Summary

At this stage, we have solved most of the problems regarding the client side emulation library

- emulation runtime library sets up the TLS, IPC Buffer.
- seL4 IPC emulation implementation using UNIX Domain Sockets to pass the message registers as well as the emulated register set.
- the IPC emulation library distinguishes the seL4 IPCs and the emulation internal IPCs using two different protocols.
- memory mapping emulation implementation uses signal handlers to capture the vm fault and might do lazy mapping on-demand or halt the execution.
- one seL4 thread is mapped to one Linux process

However, we haven't discussed the rest of the problems in detail.

- How does the kernel emulator run?
- How does the kernel emulator start the roottask?
- How do we emulate physical memory?
- How does the kernel emulator start the roottask?
- How can kernel distinguish different seL4 applications?
- How to emulate the scheduling?

The following slides will focus on discussing those questions.

Kernel Emulator Implementation

To implement the kernel emulator, we want to reuse the kernel code as much as possible and modify the kernel code as less as possible.

Keeping those in mind, we wrapped the kernel code and run as a Linux process with the following modification:

- Provide a new kernel entry point, but we can reuse the boot code to emulate the system booting stage. (Collect the boot info and set up the kernel objects for the roottask, etc.)
- After the kernel boots and entring the userland, we will only trap into the kernel routine via interrupt or exceptions. (In seL4, interrupts and exceptions are all handled in one routine and system calls which are entered using syscall instructions will enter a fast syscall routine) So we need to provide wrapper functions for the kernel interrupt handling routine and syscall handling routine.
- We need to emulate privilege instructions or bypass them.

Kernel Emulator Implementation (cont)

First of all, we need to decide how to modify the kernel code. The current implementation is to duplicate parts of the kernel code and reuse the seL4 build system to generate headers we need and use configuration to control the building of the original seL4 kernel and the kernel emulator.

- the goal of doing this is because it's quite simple, we don't need to implement our own build systems.
- don't need to worry about making a lot of modifications to the original kernel code base and making it hard to read.

seL4 Kernel Recap

Recap on seL4 kernel functionalities

For emulating a simple hello world roottask, the minimal requirements of the kernel objects are **cnode**, **vspace**, **endpoints**, and **tcb**.

- the **capability** is the most critical part of the seL4 system. As far as I understand it's **`access rights + reference`**. With the capability, we can retrieve the virtual address of this kernel object then access the object.
- the **vspace** implements the architecture dependent paging structures so that we can map virtual addresses to the physical addresses.
- the **endpoint** facilitates the message-passing communication between threads and implements the rendezvous IPC model. The endpoint structure internally is implemented as a queue which only has a head and a tail. They both point to a **tcb** structure.
- the **tcb** represents the seL4 thread. It contains all information about the seL4 thread, such as the thread state, capability pointer to the IPB buffer, fault, scheduling priority, etc.

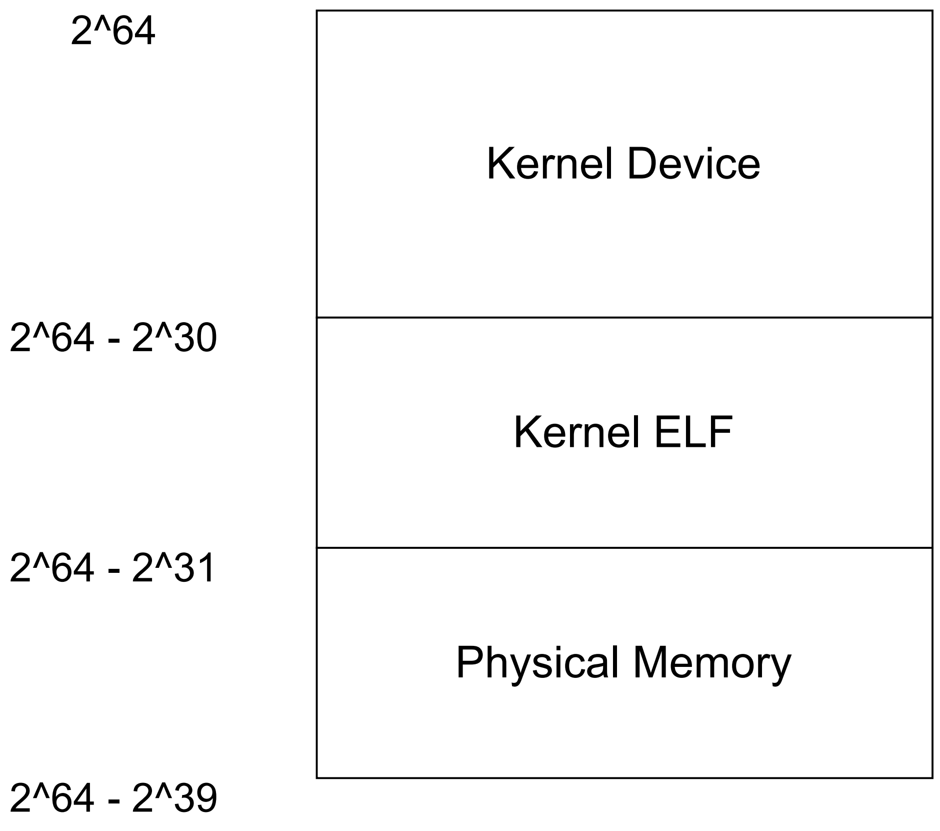
  However, we can see the most important part is how to resolve the pointer referencing so that the kernel emulator can access any seL4 thread's kernel objects. We explore how the original kernel does that. The answer is the seL4 kernel map the entire physical memory into the kernel window.(discuss in next slide)

seL4 Kernel Window Mapping

Kernel window mapping

The kernel has 1:1 physical memory mapped in the virtual memory. And this mapping is at the base of the kernel window which is 2^64 - 2^39. (assume we are using the x86_64 architecture)

Therefore, we can easily access any data on the physical memory by add an offset to the physical address.



Physical Memory Emulation Implementation

With the mapping method discussed in the previous slide, the kernel can easily access any content on the physical memory by calculating the offset.

However, for the emulation, there is one challenge. Since the kernel needs to map the physical memory to a very high address, it uses a custom linker script to assign the predetermined address at linking time. Hence, our **text**, **bss** as well as the **data** section will also be assigned those high virtual addresses. But this will not work on Linux as the Linux kernel doesn't allow us to use those high virtual memory areas.

To solve this we are going to map a shared memory in the kernel emulator address space, and use this as a 1:1 mapping to the emulated physical memory.

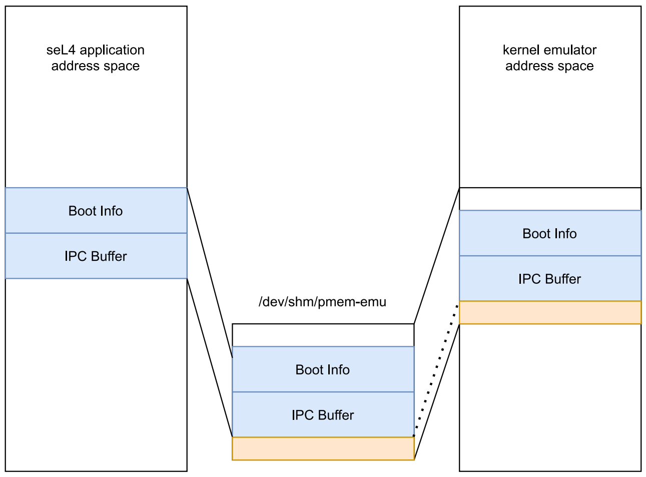
Physical Memory Emulation Implementation
(cont)

With this approach, we only need to modify a few codes, then most of the kernel code will work automatically as they are manipulating data structures.

To implement this we are going to use the **`shm_open`** syscall provided by Linux. This will create a shared memory file in **`/dev/shm/`** and returns us a file handle. This Implementation also solved the problem of passing boot info as well as setting up the IPC buffer for the seL4 application on the client side.

- to pass the boot info, we first locate the boot info frame by calculating the offset.
- pass the offset using emulation internal IPC to the seL4 application.
- seL4 application maps a page for the part of the file based on the offset.
- the same method makes mapping IPC buffer work as well.

Physical Memory Emulation Diagram



Kernel Emulator Implementation (cont)

Since we are using **UNIX Domain Socket** to emulate the seL4 IPCs, hence we need to determine the calling seL4 application. In real seL4, this is easy, as there is a global variable named **`ksCurThread`** which always points to the current running thread. For the emulation, to achieve this we use **process ID** as well as an internal bookkeeping structure to determine the calling seL4 application.

To start the roottask is quite straightforward, we implement this using **`fork`** and **`execve`**

Future Work

Most of the rudimentary designing and developing work of this project has been done. However, due to the time limits (I've been blocked on some problems and bugs for quite a long time) and my capability. There are still lots of work that haven't been done yet.

Still working on it, might be finished soon

- The current platform info collection in the kernel booting stage is hardcoded as it's not quite important, but we can make it be configured by the user.
- The current implementation only supports running the roottask at the moment, should be extended to run multiple seL4 applications.
- The current implementation has only been tested on simple applications, should be tested on more complicated applications. (e.g. capdl-loader, seL4 test suites)
- About emulating scheduling, I've just come out with the idea recently, still implementing it.
  To emulate the round robin scheduling in master kernel, we can use blocking sockets, so each time kernel uses the scheduling algorithm to determine the next running seL4 thread and only replies to it. Hence all other seL4 applications will block on the socket except one that has been chosen.
  To emulate the preemption, we can set a timer in the kernel emulator, and signal our seL4 application, then the signal handler routine is invoked and block on the socket until the next IPC message is passed from the kernel emulator telling it to resume.

Future Work (cont)

Definitely future works

- modify the simple operating system and compile it using x86_64 version and test it with the emulation framework.
- study the MCS kernel and see if we can emulate the MCS scheduling.
- extend the minimusllibc as well as the IPC emulation library (emulated register set, calling conventions) to support more architecture, so make the framework architecture-independent.
- achieve binary compatibility vis **`ptrace`** or **Linux User Syscall Dispatch feature**. (currently only supported on x86 and kernel version ≥ 5.11)

Code Demo