2021 T2 Thesis B
seL4 API/ABI Emulation on Linux

Design and implementation in the emulation framework

Student: *Jiawei Gao*

Supervisor: *Gernot Heiser & Axel Heider*

Produced: 09 Aug 2021

Content Layout

- Project Recap
- Client Side Emulation Framework Implementation
- Kernel Emulator Implementation
- Future Work
- Code Demo

Project Recap

What is the project about?

- A **compatibility layer** for the seL4 system (seL4 threads + seL4 kernel) to run on Linux.
    - clients = seL4 applications = seL4 threads
    - server = kernel emulator
- No **machine emulation** (e.g. Qemu, etc.) or **virtualization**. (e.g. KVM, etc.)

What are the benefits?

Developing seL4 userland applications:

- Rapid prototyping an application for seL4 system in Linux environment.
- Use Linux tools directly. (e.g. GDB, valgrind, strace, etc.)
- Directly access to Linux system's I/O (e.g. Files, Networking ,etc.).

Project Recap (cont)

Current Outcome

- Source code compatibility.
- A fundamental emulation framework.
- Run some simple seL4 roottasks.
- Invoke most seL4 kernel functionalities. (cnode, vspace, tcb, etc.)

Analysing "Hello World" Roottask

What does a "Hello World" roottask need?

A runtime library

- seL4runtime
    - Set up the environment for an seL4 thread. (e.g. Thread Local Storage (TLS) region, POSIX syscall API redirection, IPC buffer, etc.)

C library and its backend

- musllibc
    - Provide POSIX programming interfaces for the developer. (e.g. **`printf`**, **`open`**, **`read`**, etc.)
- seL4musllibcsys
    - Provide backend handlers for seL4 musllibc.

seL4 system call library

- libseL4
    - Provide seL4 system call APIs.

seL4 kernel

- Provide critical seL4 kernel functionalities.

Analysing "Hello World" Roottask (cont)

How many seL4 syscalls does the "Hello world" roottask invoke?

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

How to redirect those syscalls?

Modify the libseL4 to integrate our emulation library.

At which level?

Modify the raw syscall ASM wrappers (**`syscall_syscalls.h`**) to achieve **minimal** modification in the original code. Take x86_64 as an example:

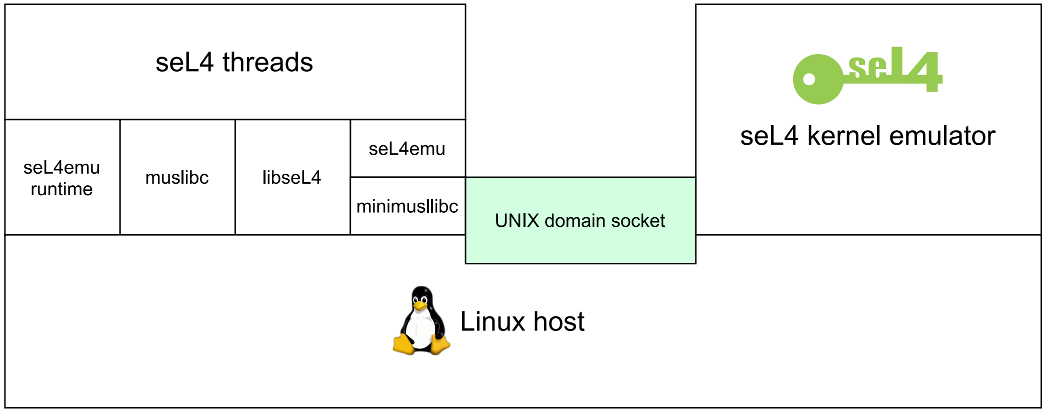
`x64_sys_send`	invokes	**`seL4emu_sys_send`**
`x64_sys_reply`	invokes	**`seL4emu_sys_reply`**
`x64_sys_send_null`	invokes	**`seL4emu_sys_send_null`**
`x64_sys_recv`	invokes	**`seL4emu_sys_recv`**
`x64_sys_send_recv`	invokes	**`seL4emu_sys_send_recv`**
`x64_sys_nbsend_recv`	invokes	**`seL4emu_sys_nbsend_recv`**
`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.
- Linux syscall wrapper library
  Develop a library that **connects** emulation framework and Linux host.
- Client runtime library
  Modify the current sel4runtime to **initialize** the emulation library.
- Emulation IPC library
  Develop a library that **connects** seL4 threads and kernel emulator.

Emulation Framework Architecture



Minimusllibc Implementation

Minimusllibc is a C library **only** used by the emulation library.

Current implementation, simple but works :)

- Port musllibc (glibc is too complex, porting it is overkill) and change symbol names. (named **minimusllibc**) ✓

Why?

- Symbol name confliction with host glibc and seL4 musllibc. ✗
- Static linking introduces limitations. ✗
- Dynamic loading library depends on glibc, porting one is quite overkill.✗

The current minimusllibc provides

- Socket related APIs (e.g. **`mini_socket`**, **`mini_accept`**, etc.)
- I/O APIs (e.g. **`mini_write`**, **`mini_read`**, etc.)
- Memory mapping APIs (e.g. **`mini_mmap`**, etc.)
- Signal APIs (e.g. **`mini_sigaction`**, **`mini_sigstack`**, etc.)
- **But not thread safe** (unnecessary at the moment).

seL4 Emulation Runtime Library Implementation

Modification on seL4runtime

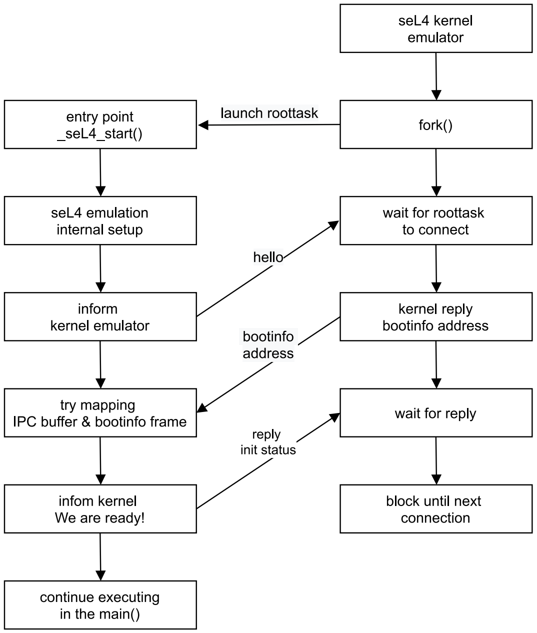
Emulation library **environment setup** before entering the **`main`** function.

- **Obtain the bootinfo**
  - Before kernel passes bootinfo as function arguments.
  - Now map bootinfo frame as shared memory to access.
- **Setup TLS**
  - Before kernel sets up the TLS region.
  - Now we set up by ourselves implicitly. (e.g. using **`FSGSBASE`** instruction family on x86 if available, otherwise use Linux process control syscalls)
- **Emulation library internal setup**
  - Before no such routine.
  - Now we initialize the emulation library. (Explain in the next slide)

seL4 Emulation Runtime Library Implementation (cont)

The initialization of the emulation library will be done **before** the **`main`** function starts.

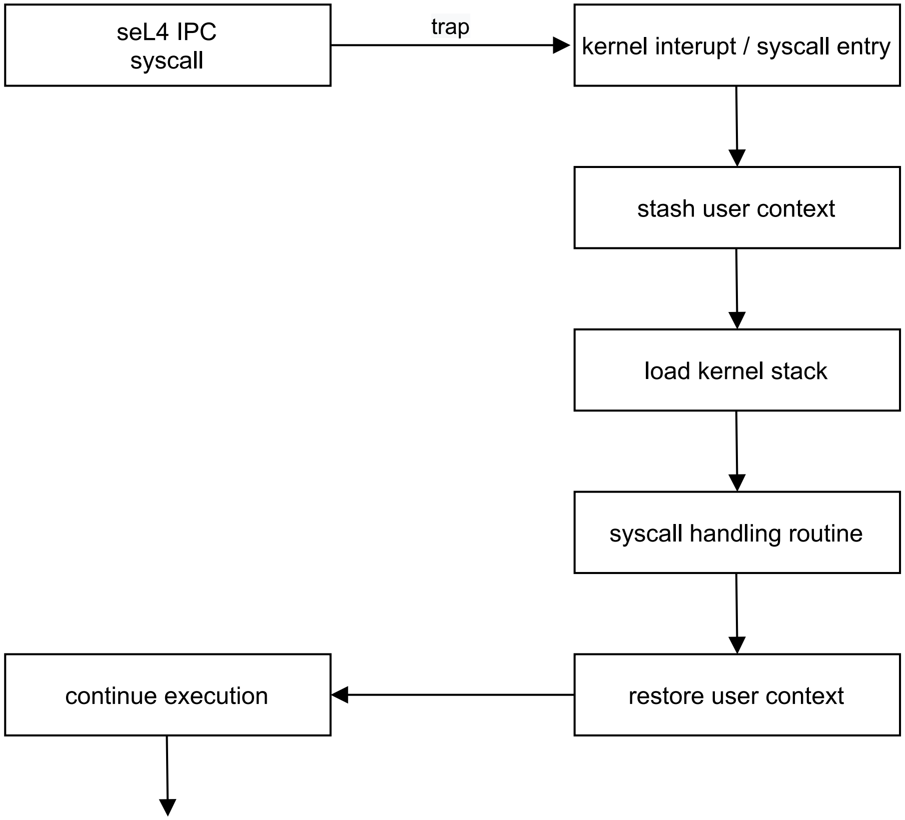
- Handshaking between the kernel emulator and the roottask.
- Initial setup for the roottask. (map IPC buffer, bootinfo, set up signal handler)
- Report about the initialization status.


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

To emulate the seL4 IPC invocation:

- Use UNIX Domain Sockets to transport IPC messages.
- **`Send`** and **`receive`** emulates **context switching** between the seL4 kernel and the userland.



seL4 IPC Emulation Internals

To implement the seL4 IPC emulation, we need to:

- Pass the **emulated register set values** + the message registers.
- Follow the architecture specific calling conventions and the seL4 semantics.

For example to emulate **`x64_sys_send_recv`**: on x86_64:

```
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"
        );
}
```

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**

IPC Emulation Protocol

The IPC message layout is as follow:

Tag	ID	Body
Word Size	Word Size	Word Size * n_contextRegisters

The UNIX Domain Sockets message passing has two protocols:

- Use **Tag** to distinguish:
  - seL4 IPC emulation message.
  - Emulation library internal message.
- The kernel emulator uses **ID** to distinguish the calling seL4 thread.
- The body section can be either:
  - The user context register set.
  - The internal IPC message content. (**n_contextRegisters** is defined in the arch dependent **registerset.h**, on x86_64 it is **24**)

seL4 IPC Buffer Emulation

How to pass our IPC Buffer?

- The seL4 thread and the kernel emulator are **different Linux processes** (explain in later slides), they have different address spaces.
- Writing to the IPC buffer should **immediately** appear on the kernel emulator's view.
- IPC buffer is **4KB size**, passing as UDS message can be a significant overhead. ✗

The current implementation uses a mapped shared memory to emulate the IPC buffer.

- Efficiently pass the content on the IPC buffer. ✓
- **`Write`** and **`Read`** won't have any concurrency issues. ✓
- Kernel emulators can access the seL4 thread's IPC buffer directly even they are different processes. ✓

seL4 Thread Emulation Implementation

To emulate the seL4 threads, we have two implementation options:

- Map each one to a Linux user space **thread**.
- Map each one to a Linux user space **process**.

Trade-offs of mapping to threads

- By default, we share the address spaces. ✓ , but we can't isolate them. (can't emulate different vspaces)✗
- Creation/Deletion are relatively light weighted. ✓
- Introduce implementation complexity. (concurrency issues, signal handlings, etc.)✗
- pthread library already uses TLS, which seL4 is using as well. ✗

Trade-offs of mapping to process

- Can emulate different vspaces. ✓
- Simpler to implement. ✓
- Process creation/deletion is relatively heavy. ✗

Memory Mapping Emulation Implementation

Implementation challenges:

- The emulation is all in the user space, we can't modify the paging structures.
- The program should continue execution normally after a seL4 mapping invocation.
- The emulation of lazy mapping should also work.
  - Need to emulate the virtual memory fault.

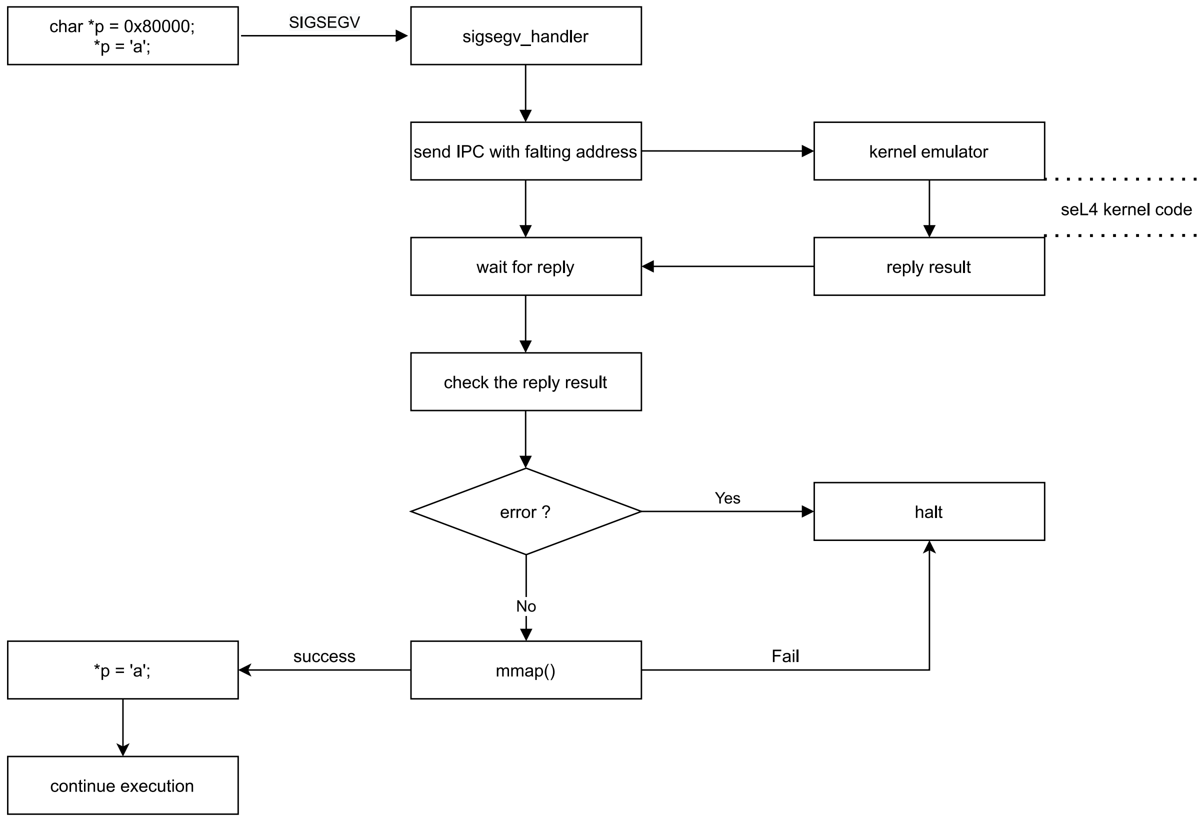
Use **`signal`** to solve the challenges:

- **Software interrupts** emulate hardware interrupts such as seL4 VM fault.
- Signal handlers **implicitly** do mapping jobs.
- The program execution **continues normally** after returning from signal handlers.
- Emulating both explicit mapping calls and lazy page mapping will work.

Memory Mapping Emulation Implementation (cont)

The following diagram illustrates the program flow of the memory mapping emulation.

- Assume *0x80000* is already mapped by the virtual memory management server.
- Assume the paging structure is updated already.



Kernel Emulator Implementation

To implement the kernel emulator, we want:

- Reuse the kernel code **as much as** possible.
- Modify the kernel code **as less as** possible.

The current modifications in the kernel:

- Provide a new kernel entry point.
- Reuse the boot code to do the initialization work. (Collect the bootinfo and set up the kernel objects for the roottask, etc.)
- Dispatch the seL4 IPC message into the **kernel interrupts or syscalls handling routines**.
    - (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**)
- Emulate privilege instructions or bypass them.
- Emulate the **kernel window mapping**.

seL4 Kernel Recap

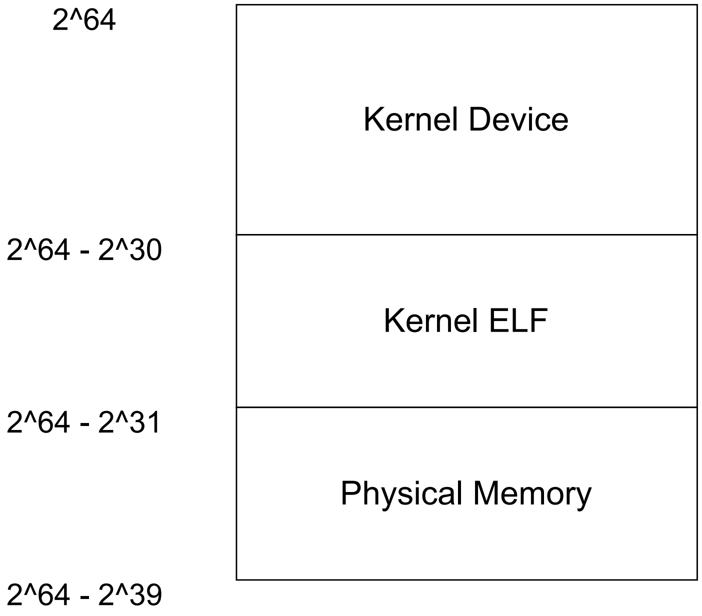
How does kernel access kernel objects?

use capability (**access rights + reference**) ⇒ memory reference ⇒ any kernel objects on the memory.

Kernel window mapping

The kernel has 1:1 physical memory mapped in the virtual memory.

- At the **fixed base** of the kernel window which is 2^64 - 2^39. (assume we are using the x86_64 architecture)
- Easily access any data on the physical memory by adding an **offset** to the physical address.



Physical Memory Emulation Implementation

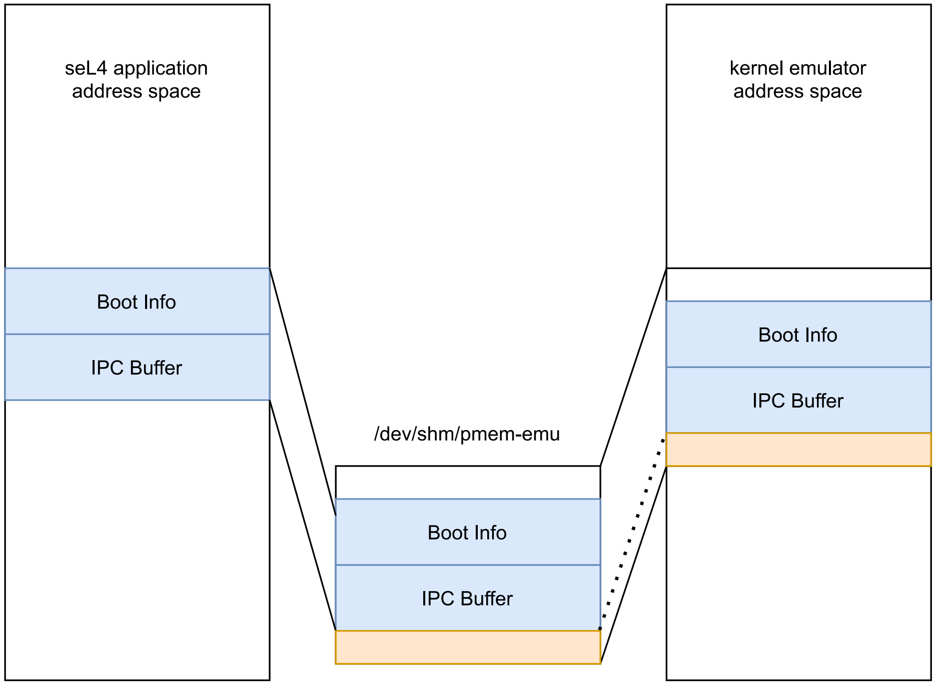
The physical memory emulation is also implemented using a 1:1 mapping method.

- Use shared memory to emulate the physical memory.
- Efficiently solved sharing IPC buffers.
- Efficiently solved passing bootinfo to the roottask.
- Kernel code modification only involves address the translation part.

Physical Memory Emulation Example

Stages of seL4 runtime setup using emulated physical memory:

- Pass the bootinfo via physical offset and the internal IPC message.
- A roottask maps a page for the part of the file based on the offset.
- The same method to map the IPC buffer.



Future Work

The current implementation solved most of the fundamental problems. And can emulate simple roottask and most of the kernel functionalities. (cnodes, vspace, tcb, etc.)

Partially finished:

- User-configurable kernel booting emulation. (roottask image, physical memory size, platform information, etc.)
- Running multiple seL4 threads so that we can use CAmkES on the emulation framework.
- Scheduling emulation, I just came up with some ideas recently, due to the time limits of the presentation, this part resides on last slide as extra content.

Future Work (cont)

Non-started work

- extend emulation framework to be 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

The roottask (seL4 tutorial capability roottask) does the following things in the **`main`**:

- Retrieve the bootinfo to get the **first free CSlot** and **Last free CSlot**.
- **Copy** cnode to both first free and last free slots.
- Use the **copied capability** to **set thread priority**.
- **Revoke** both of the copied capabilities.
- Test if **revoke invocation** successful.
  - Try moving capabilities in the empty cnode slots.
  - Expect **`seL4_FailedLookup`** as result.
- Suspend the thread execution.

Code Demo (cont)

The demo will be three parts.

- Run roottask both on qemu and the emulation framework.
    - Compare and show results are the same.
- Run kernel emulator independently and debug roottask.
    - Show the seL4 emulation runtime library setup routine.
    - Show the emulation of seL4 IPC calls.
- Debug roottask and kernel emulator simultaneously.
    - Show how we emulate the **context switching**.
    - Show **smooth switch** between kernel emulator and the roottask.

Code Demo-Part 1

Compare output on qemu and seL4 emulation framework

Code Demo-Part 2

Debugging seL4 emulation runtime library + IPC emulation

Code Demo-Part 3

Debugging seL4 roottask and kernel emulator simultaneously

An extra discussion on the scheduling emulation

About emulating scheduling, I've just come up with the idea recently, still implementing it.

- To emulate thread suspension, 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 threads will block on the socket except one that has been chosen.
- To emulate the preemption, we can send signal to our seL4 application, then the signal handler routine is invoked and will be blocked on the socket until the next time when the kernel emulator sends a message and tells it to resume.

THANKS :)