Aarch64 Hypervisor from Scratch

Stephen Tong

CS 3210: Design of Operating Systems

May 4, 2020

Contents

Abstract

Hypervisor technology have gained significant influence in the last few years. Hypervisors, or virtual machine monitors, allow computer systems to virtualize computer software and run virtual machines. Virtualization allows entire operating systems to be migrated by abstracting away hardware implementations. However, hypervisor support for the hardware is required.

In this project, we discuss the design and implementation of a 64-bit ARM hypervisor for the Raspberry Pi. We will also cover the challenges we encountered in the process. We successfully virtualized the guest's physical memory and MMIO. We also leave room to support IRQ virtualization.

1 Introduction

In our project, we aim to develop an Aarch64 hypervisor suitable for hosting RustOS in a virtual machine on the Raspberry Pi, and virtualize various hardware components of the Pi. The Raspberry Pi has several idiosyncrasies which make other commodity hypervisors not quite suiable for the system. We aim to address these problems.

Our code is available here.

2 Background

The Raspberry Pi is a cheap, miniature computer suitable for embedded applications. Its specifications were designed to appeal to a broad audience, both hobbyists and professionals. Many embedded devices use ARM system-on-chip (SoC) components because they are very compact and power-efficient. The Raspberry Pi is no different. The Raspberry Pi offers a BCM2837 SoC. The BCM 2837 SoC offers the following peripherals:

- Programmable timer
- Interrupt controller
- UART
- Synposis USB controller
- 400MHz VideoCore
- Programmable DMA controller
- 1GB DRAM

The SoC, of course, includes an ARM CPU, clocked at 1.2GHz. Considering the low price and relatively powerful hardware, the Raspberry Pi is a very handy product. It would be really great if we could run multiple OS on the Raspberry Pi.

Fortunately, the ARMv8.0 ISA the BCM2837 SoC implements supports virtualization. In the ARM architecture, software privilege levels are implemented in hardware as exception levels (ELs). Figure 1 shows how ARM exception levels are laid out on Aarch64.

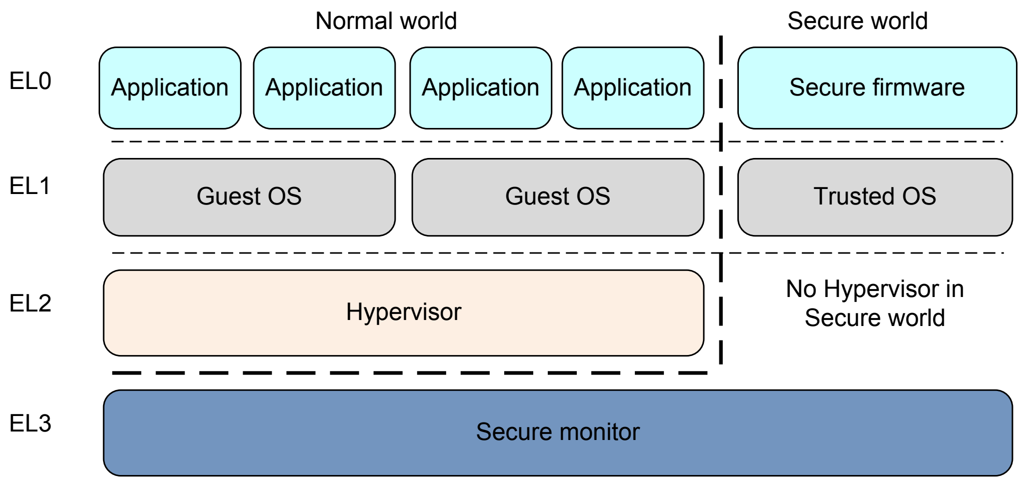
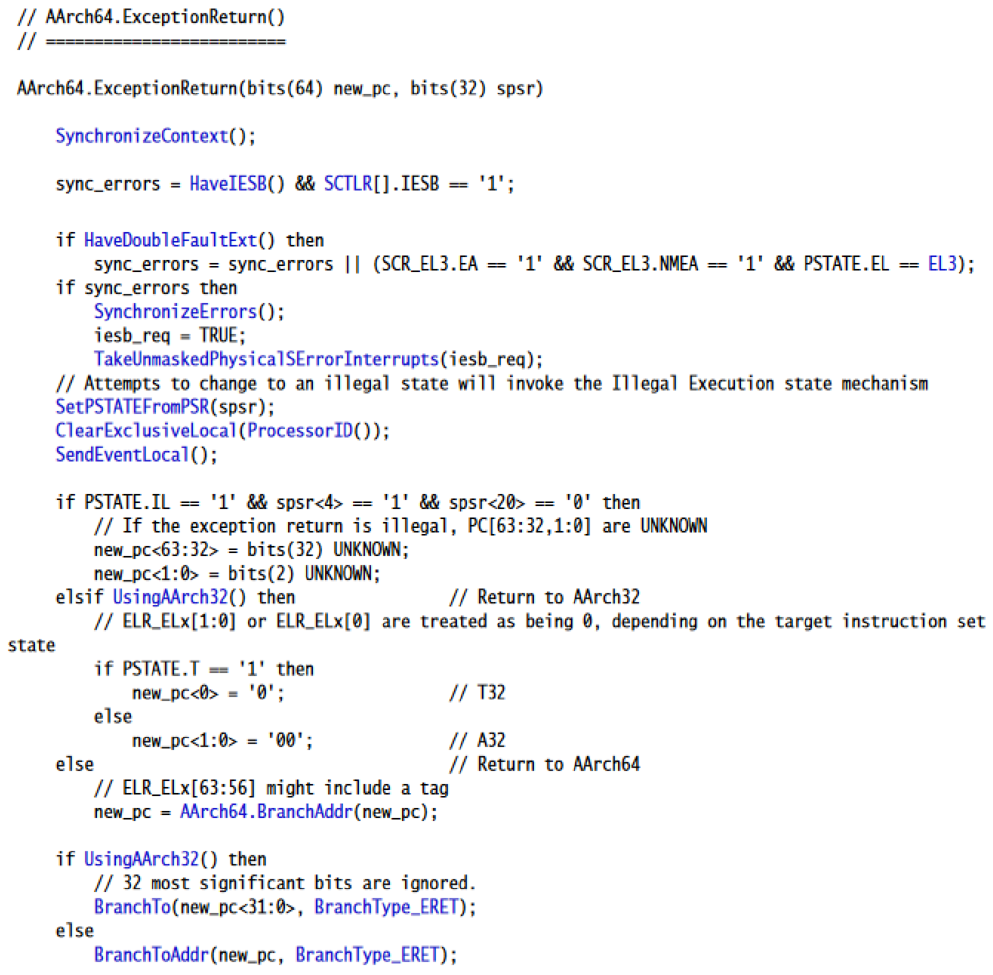


Figure 1: ARM exception levels. Hypervisors run at EL2, operating systems run at EL1, and user-space programs run at EL0.

In Aarch64, exceptions are the only way to change privilege level. To raise privilege level, code needs to trigger an exception. For example, user code can use the `svc` (supervisor call) to trap to the operating system, and the operating system can use `hvc` (hypervisor call) to trap to the hypervisor. To lower instruction level, the only possible way is to use the `eret` instruction. Figure 2 describes the operation of the ERET instruction.

As shown in the figure, ARM manages the system context using control registers, known as *system registers*. These registers include `ELR_ELx`, which stores the exception level to return to. Other control registers, like `HCR_EL2`, the hypervisor control register, which controls enables and disables hypervisor features. Another crucial control register for virtualization is `VTCR_EL2` and `TCR_EL2`, which controls *stage 2* memory translation. Memory translation without hypervisor enabled is considered `stage 1` memory translation, and stage 2 memory translation adds an additional layer of indirection. This additional indirection allows the hypervisor to effectively virtualize the guest's view of physical memory. The guest's view of physical memory is referred to as IPA or *Intermediate Physical Address*. Figure 3 and Figure 4 shows how hypervisors affect virtual memory translation.

Note that the hypervisor itself does not have stage 1 memory translation. The hypervisor's page tables map directly to physical addresses, wheras the operating system's page tables map to IPA.



```
// AArch64.ExceptionReturn()
// =========================

AArch64.ExceptionReturn(bits(64) new_pc, bits(32) spsr)

    SynchronizeContext();

    sync_errors = HaveIESB() && SCTLR[].IESB == '1';

    if HaveDoubleFaultExt() then
        sync_errors = sync_errors || (SCR_EL3.EA == '1' && SCR_EL3.NMEA == '1' && PSTATE.EL == EL3);
    if sync_errors then
        SynchronizeErrors();
        iesb_req = TRUE;
        TakeUnmaskedPhysicalSErrorInterrupts(iesb_req);
    // Attempts to change to an illegal state will invoke the Illegal Execution state mechanism
    SetPSTATEFromPSR(spsr);
    ClearExclusiveLocal(ProcessorID());
    SendEventLocal();

    if PSTATE.IL == '1' && spsr<4> == '1' && spsr<20> == '0' then
        // If the exception return is illegal, PC[63:32,1:0] are UNKNOWN
        new_pc<63:32> = bits(32) UNKNOWN;
        new_pc<1:0> = bits(2) UNKNOWN;
    elsif UsingAArch32() then                // Return to AArch32
        // ELR_ELx[1:0] or ELR_ELx[0] are treated as being 0, depending on the target instruction set
state
        if PSTATE.T == '1' then
            new_pc<0> = '0';                 // T32
        else
            new_pc<1:0> = '00';              // A32
    else                                     // Return to AArch64
        // ELR_ELx[63:56] might include a tag
        new_pc = AArch64.BranchAddr(new_pc);

    if UsingAArch32() then
        // 32 most significant bits are ignored.
        BranchTo(new_pc<31:0>, BranchType_ERET);
    else
        BranchToAddr(new_pc, BranchType_ERET);
```

Figure 2: Operation of the `eret` instruction on Aarch64.

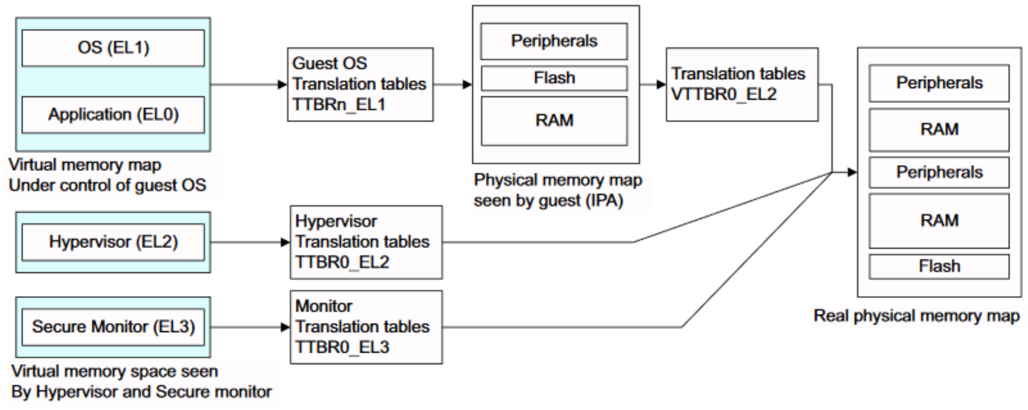


Figure 3: Operation of two-stage virtual memory translation on ARMv8.

Figure 4: ARM trying to help out system programmers with their addiction.

The EL3 secure monitor is also part of the ARMv8 architecture. However, we do not consider secure monitor for our project. The Raspberry Pi can actually be configured to boot directly into EL3 instead of EL2.

ARMv8 also support virtualizing interrupts for the guest. One way this can be done is using the `HCR_EL2.VI` bit to simulate a IRQ to the guest. Another way, which the ARM documentation recommends is better, is to use a General Interrupt Controller (GIC). However, the BCM2837 SoC does not have a GIC, so we must resort to the first method. To virtualize the IRQs, we will need to route all IRQs directly to the hypervisor, not the guest. This is done by setting the bit `HCR_EL2.IMO`. Then, in the hypervisor's interrupt handler, we emulate the IRQ and pass it to the guest. It is also necessary to trap all guest writes to system registers or MMIO relating to interrupt configuration so that the hypervisor knows exactly which IRQs should be simulated for the guest.

3 Overview

In section 4, we will discuss the high-level design of our hypervisor. In section 5 we will describe several implementaton details of our hypervisor. In section 6 we will demo our hypervisor.

4 Design

Note that unlike in other architectures (e.g., x86), the process for switching privilege levels is remarkably homogeneous across the various privilege levels. This means that EL2 and EL1 code are actually remarkably similar in how they interact with lower privilege code. Thus, our EL2 hypervisor can be designed very similarly to our EL1 RustOS. For example, to translate the trap frame code from EL1 to EL2, we simply just replaced EL0 with EL1 in some of the system register names.

The virtual memory abstraction also remains intact, with the hypervisor enjoying a identity mapping of physical memory, and the OS receiving a IPA. However, the large amount of memory allocated to each guest creates a crucial design decision for us. Previously, in our RustOS, each user process received very little memory and we could just allocate all of it right away. However, to run a whole OS, we give each guest 256MiB out of the 1GiB available memory on the Raspberry Pi. It is very inefficient to map all of this memory, even though most of it will not be even used. Thus, we opt to implement *lazy paging*, where the memory is only mapped when the page is first used. This is accomplished by simply leaving the memory unmapped, and handling the stage 2 translation page fault in our hypervisor.

By controlling the guest's IPA, we can also virtualize all MMIO the guest sees. We simply leave the MMIO pages unmapped, and when we receive a translation fault in the MMIO region, we emulate the MMIO behavior. This allows our hypervisor to intercept interactions with peripherals such as the UART or timer controller.

5 Implementation

We implemented our hypervisor in distinct stages, similar to the Lab 4 procedure we followed. We based our hypervisor off of the Lab 3 code as a template, and progressively added back parts of our Lab 4 code as needed. Of course, much of the code was modified to suit a hypervisor rather than an ordinary OS. We will describe the stages we took below.

5.1 Entering EL2

Our first and foremost goal was to boot into our kmain at EL2. This was very easy, because the code was already in init.rs, so we removed the code for entering EL1 from EL2.

5.2 Enabling MMU

Unlike in Lab 4, where we left the MMU for the end, we decided to enable the MMU right away. This is because the MMU is absolutely essential for performing most of the other hypervisor functions, such as MMIO device virtualization, IPA, and so on. It is also necessary in order to load our RustOS kernel at its proper load address, 0x80000. Since the kernel and hypervisor images are not relocatable, virtual memory is required for both our hypervisor and RustOS to share memory. First, we added definitions for structs, registers, and bitfields related to stage 2 memory translation. Then, we configured the two registers that were stage 2 equivalents for `TCR_EL1`. Figure 5 shows the code used to do this.

One huge headache that we faced was cache coherency between the MMU and CPU data cache. At one point, the guest was able to run EL1 in QEMU, but not the hardware Pi. This was because the Pi's MMU was reading old

```rust
let ips =
 ↪  ID_AA64MMFR0_EL1.get_value(ID_AA64MMFR0_EL1::PARange);
MAIR_EL2.set(
    (0xFF <<  0) |// AttrIdx=0: normal, IWBWA, OWBWA, NTR
    (0x04 <<  8) |// AttrIdx=1: device, nGnRE (must be OSH
 ↪  too)
    (0x44 << 16), // AttrIdx=2: non cacheable
);
TCR_EL2.set(
    (0b1 << 31)  |// RES1
    (0b1 << 23)  |// RES1
    (0b00 << 20) |// TBI=0, no tagging
    (ips  << 16) |// IPS
    (0b01 << 14) |// TG0=64k
    (0b11 << 12) |// SH0=3 inner
    (0b01 << 10) |// ORGN1=1 write back
    (0b01 << 8)  |// IRGN1=1 write back
    ((VISOR_MASK_BITS as u64) << 0), // T0SZ=32 (4GB)
);
isb();
TTBR0_EL2.set(baddr);
HCR_EL2.set(HCR_EL2::VM | HCR_EL2::RES1);
nuke_tlb_host();
asm!("dsb sy");
isb();
```

Figure 5: Code to enable virtual memory for hypervisor.

```rust
impl GuestPageTable {
    pub fn new() -> GuestPageTable {
        let mut pt =
 ↪  PageTable::new_stage2(Stage2EntryPerm::READWRITE);
        VMM.mark_noncacheable(&pt);
        GuestPageTable(pt)
    }
    // ...
}
```

Figure 6: We do not cache pagetables in hypervisor memory, or else we will need to flush every time we edit them, as it may cause incoherency with the MMU.

data for the page table, because when we initialized the page table the memory was not written back immediately. We solved this first by aggressively flushing all pages the guest would use that the hypervisor setup. However, a much more elegant solution that we discovered was to simply mark all page tables and guest memory as non-cacheable memory for the host. This would prevent any stale cache issues. Figure 6 shows this.

5.3 Handling EL1 exceptions

Now that we could load the guest kernel, we wanted to be able to handle exceptions from the kernel. This would allow us to easily catch exceptions and find mistakes in our code and debug. To do this, we simply copied most of the exception and vector code from our Lab 4 implementation. However, we had to seriously adjust the trap frame. Figure 7 shows our new trap frame.

5.3.1 MutexGuard Monad

As a side note, we refactored `MutexGuard` and made it a Monad to make it more flexible. Essentially, we added a `map` method similar to `Option.map` that allows you to operate on the locked data inside the guard. Because the operation happens inside the guard, this ensures that the lock is held during the operation, ensuring safe inner mutability. This is extremely useful because it allows us to return modified `MutexGuards` in helper metods.

```
pub struct TrapFrame {
    pub VTTBR: u64,
    pub ELR: u64,
    pub TTBR0_EL1: u64,
    pub TTBR1_EL1: u64,
    pub SP_EL0: u64,
    pub SP_EL1: u64,
    pub SCTLR_EL1: u64,
    pub VBAR_EL1: u64,
    pub TPIDR_EL0: u64,
    pub TPIDR_EL1: u64,
    pub SPSR_EL1: u64,
    _pad1: u64,
    pub qn: [u128; 32],
    pub xn: [u64; 32] // lr = x30, xzr = x31
}
```

Figure 7: Hypervisor trap frame.

Figure 8 shows an example of this.

5.4 Virtualizing MMIO

After we had virtual memory in place, our next goal was IRQ virtualization. However, to virtualize IRQs, the host needs to know which IRQs the guest would receive. To figure this out, we need to intercept the host's accesses to interrupt controllers and devices generating interrupts. Consider the timer controller: the guest wants to receive timer interrupts, so it performs MMIO reads and writes to the timer controller. Thus, the host needs to intercept these operations to track what interrupts need to be emulated for the guest.

We implemented MMIO virtualization as described in section 4. However, actually implementing the read and write emulation required involved getting additional information out of the DataAbort instruction syndromes. We had to handle all possible memory load and store cases, such as sign extension, operand sizes, etc. Figure 9 shows the code needed for this.

At this stage, we were able to run the guest RustOS in IPA, with all MMIO being virtualized! The next step to add vIRQ emulation is rather straightforward. However, I ran out of time because I had to write a bunch

```rust
// scheduler.rs
pub fn expect(&'_ self) -> impl MutexFunctor<'_, Scheduler> +
↪   '_
{
    self.0.lock().map(|opt| opt.as_mut().expect("scheduler
↪   uninitialized") )
}
pub fn get_by_vmid(&'_ self, vmid: u8) -> impl
↪   MutexFunctor<Process> + '_ {
    self.expect().map(|scheduler|
↪   scheduler.get_by_vmid(vmid).expect("bad vmid"))
}

// traps.rs
let mut process = SCHEDULER.get_by_vmid(vmid as u8);
let vmap = &mut process.vmap;
```

Figure 8: Extending MutexGuard to support safe inner mutability.

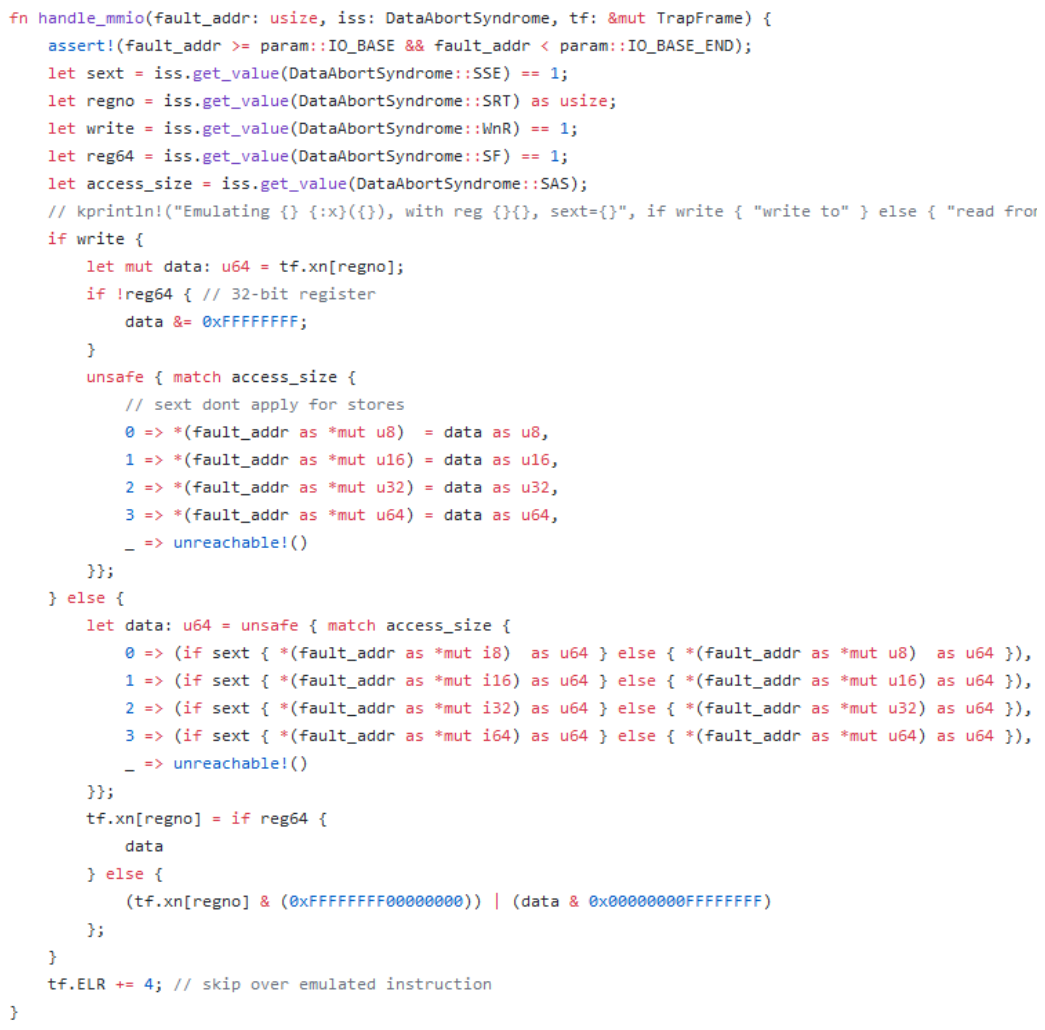
of other reports :(

6 Evaluation

Figure 10 shows a screenshot of the kernel running in the hypervisor. It works on the physical Pi for me.

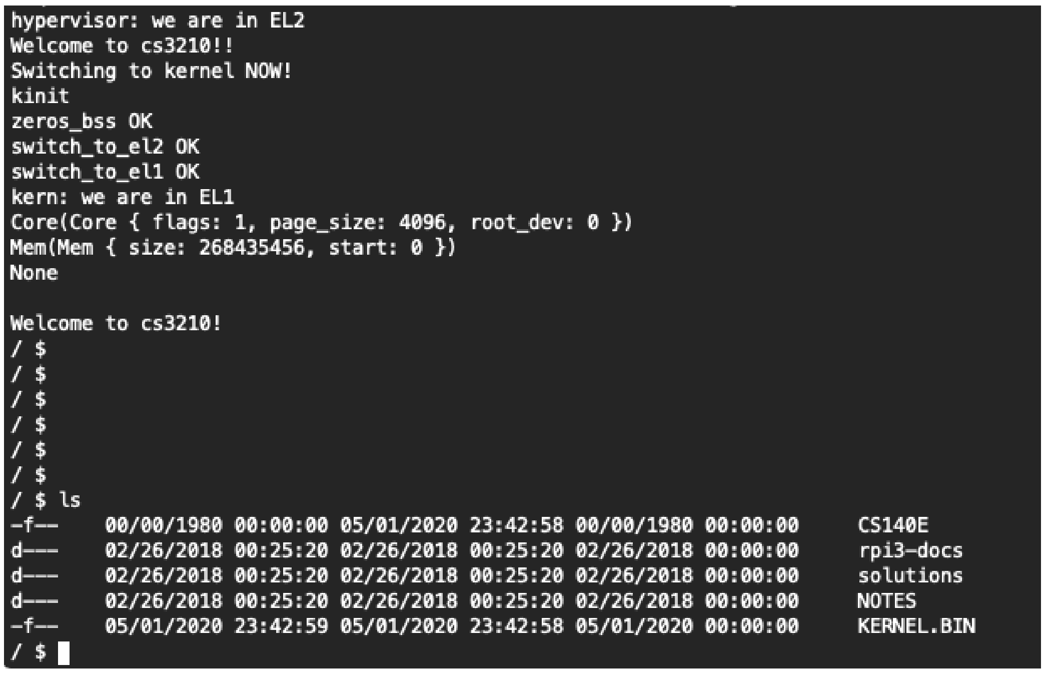
7 Discussion

Unfortunately, I ran out of time on this project so I was not able to implement virtual IRQ emulation. I'm planning to do this tonight after I finish submitting this report. The main lesson I learned throughout this process is how important carefully maintaining the caches are. I suffered a lot of debugging time due to cache incoherency or stale caches not being flushed before the data is accessed.



```
fn handle_mmio(fault_addr: usize, iss: DataAbortSyndrome, tf: &mut TrapFrame) {
    assert!(fault_addr >= param::IO_BASE && fault_addr < param::IO_BASE_END);
    let sext = iss.get_value(DataAbortSyndrome::SSE) == 1;
    let regno = iss.get_value(DataAbortSyndrome::SRT) as usize;
    let write = iss.get_value(DataAbortSyndrome::WnR) == 1;
    let reg64 = iss.get_value(DataAbortSyndrome::SF) == 1;
    let access_size = iss.get_value(DataAbortSyndrome::SAS);
    // kprintln!("Emulating {} {:x}({}), with reg {}{}, sext={}", if write { "write to" } else { "read fro
    if write {
        let mut data: u64 = tf.xn[regno];
        if !reg64 { // 32-bit register
            data &= 0xFFFFFFFF;
        }
        unsafe { match access_size {
            // sext dont apply for stores
            0 => *(fault_addr as *mut u8)  = data as u8,
            1 => *(fault_addr as *mut u16) = data as u16,
            2 => *(fault_addr as *mut u32) = data as u32,
            3 => *(fault_addr as *mut u64) = data as u64,
            _ => unreachable!()
        }};
    } else {
        let data: u64 = unsafe { match access_size {
            0 => (if sext { *(fault_addr as *mut i8)  as u64 } else { *(fault_addr as *mut u8)  as u64 }),
            1 => (if sext { *(fault_addr as *mut i16) as u64 } else { *(fault_addr as *mut u16) as u64 }),
            2 => (if sext { *(fault_addr as *mut i32) as u64 } else { *(fault_addr as *mut u32) as u64 }),
            3 => (if sext { *(fault_addr as *mut i64) as u64 } else { *(fault_addr as *mut u64) as u64 }),
            _ => unreachable!()
        }};
        tf.xn[regno] = if reg64 {
            data
        } else {
            (tf.xn[regno] & (0xFFFFFFFF00000000)) | (data & 0x00000000FFFFFFFF)
        };
    }
    tf.ELR += 4; // skip over emulated instruction
}
```

Figure 9: Read/write emulation for handling MMIO. We parse the DataAbort instruction syndrome (ISS) to correctly handle all cases.



```
hypervisor: we are in EL2
Welcome to cs3210!!
Switching to kernel NOW!
kinit
zeros_bss OK
switch_to_el2 OK
switch_to_el1 OK
kern: we are in EL1
Core(Core { flags: 1, page_size: 4096, root_dev: 0 })
Mem(Mem { size: 268435456, start: 0 })
None

Welcome to cs3210!
/ $
/ $
/ $
/ $
/ $
/ $
/ $ ls
-f--    00/00/1980 00:00:00 05/01/2020 23:42:58 00/00/1980 00:00:00     CS140E
d---    02/26/2018 00:25:20 02/26/2018 00:25:20 02/26/2018 00:00:00     rpi3-docs
d---    02/26/2018 00:25:20 02/26/2018 00:25:20 02/26/2018 00:00:00     solutions
d---    02/26/2018 00:25:20 02/26/2018 00:25:20 02/26/2018 00:00:00     NOTES
-f--    05/01/2020 23:42:59 05/01/2020 23:42:58 05/01/2020 00:00:00     KERNEL.BIN
/ $
```

Figure 10: The hypervisor can run a simple kernel in EL1 with virtual MMIO.

8 Related Work

The PiGrate team also was working on a hypervisor so they could migrate kernels. Also, many of my friends also really like hypervisors.

9 Future Work

We will add vIRQ emulation

10 Acknowledgement

I thank our professor Taesoo Kim for his hard work in making CS 3210 possible, as well as his kind and thoughtful guidance throughout my undergraduate research. I thank the CS 3210 TAs for the countless hours they spent to answer students' questions, grade assignments, and prepare lectures. Without them this course would not be possible.

11 Conclusion

We designed and implemented an Aarch64 hypervisor from scratch, based off of our basic Lab 3 OS skeleton code. We can support virtual physical memory, virtual MMIO, and in the future, virtual IRQs.