Project 7: Hypervisory (4%)

Purpose

In this project you will figure out how to use three levels of privileges via three different levels of virtual memory. We still will not run separate application and kernel/ukernel binaries (we need SD card access for that), but we can run each piece of code in its own separate domain, translated through its own page table. This is how datacenters operate, for example: the “guest” operating system runs within a virtual memory session, unaware that it is in fact running in virtual memory. This allows the hypervisor -- the operating system for operating systems — to protect itself and keep the system secure. So, for example, a hypervisor could juggle several different instances of Windows, plus several instances of Linux, plus instances of MacOSX as well, all on the same machine, all at the same time, all completely unaware of each other.

Your Task

Your code will build on the last project’s code: you will use both TTBR0 and TTBR1 to translate references, so that the ukernel’s references are translated whenever the machine is in privileged mode (e.g., whenever the ukernel on core1 is handling interrupts), and the “user” code’s references are also translated. The following page of ARM documentation shows what needs to happen:

![Diagram](image-url)
There are two page tables: one for the bottom portion of the address space (where the ukernel will live), and one for the top portion of the address space. What this documentation says is that the smallest bottom portion ends at 0x02000000 … meaning a 32MB space.

The ARM architecture defines three levels of access privilege:

User-level code runs in PL0, which your boot code invokes as USR mode; guest operating system code (our “ukernel”) runs in PL1, typically in SVC, SYS, IRQ, and FIQ modes; the main kernel runs in HYP mode which is PL2.

Note that ARM doesn’t actually restrict the ability of code running in PL1 to take over the machine. So their differentiation between PL1 and PL2 is kind of a fake security thing. But whatever. Presumably
Once you have it working, show us.

There is a new version of the "memmap" linker script, which looks like this:

```plaintext
MEMORY
{
  ram : ORIGIN = 0x0000, LENGTH = __SIZE__
}

SECTIONS
{
  .text : { *(.text*) } > ram
  .rodata : { *(.rodata*) } > ram
  .bss : { *(.bss*) } > ram
  .data : { *(.data*) } > ram

  .usercode 0x02000000 : { *(.usercode*) }
}
```

The "usercode" portion is new and tells the linker to put the code in that section way up into the virtual memory starting at 32MB, which happens to be covered by TTBR1.

So you will have two page tables: one to cover the ukernel, and another to cover user space. This will be enabled on cores other than core0, which will run the kernel in physical space.

**Build It, Load It, Run It**

Once you have it working, show us.