The (weights) of questions are relative and may not necessarily add up to 100.

No code submission required

Read the project description (https://fxlin.github.io/p1-kernel/) before proceeding

Part A: Q&A

This assignment refers to the code under **exp6/**.

1. (5) How many levels of pgtables will we have, if we map memory in sections (2MB)?

   There will be three levels of page tables if we map in sections (2MB).

2. (5) What's the base address of kernel's virtual address space? What's the rationale for choosing such a value?

   The base address of the kernel's virtual address space is 0xffff0000_00000000. The rationale for the base address is that the linker will think that the image is going to be loaded at the base address and will offset as appropriate. We set the kernel's virtual address space to be this value because we want to create a separation between the kernel process and user process. To do so, we set the first 16 bits as 0xffff as an indicator that this address and onwards can only be accessed by the kernel.

3. (5) What's the base address of each process's virtual address space? The rationale for choosing such a value?

   The base address is 0x00000000_00000000 for the same rationale as the previous question except since this is in the user's space, we want to first 16 bits to be 0x0000 because this indicates that 0x00000000_00000000 and up to 0x0000ffff_ffffffff can be accessed by the user process.

4. (5) When kernel just starts and MMU is off, the CPU should be accessing memory using physical addresses. If that's the case, how could the kernel possibly access variables (e.g. bss_start) or functions (e.g. memzero), which are linked at virtual addresses (see pictures below)?

   ```
   @granger1 (master)[exp6]$ nm build/kernel8.elf |grep bss_begin
   ffff000000085748 B bss_begin
   @granger1 (master)[exp6]$ nm build/kernel8.elf |grep memzero
   ffff000000084964 T memzero_
   ```

   (Hint: check the disassembly of the kernel binary that access the variables/functions)

The (weights) of questions are relative and may not necessarily add up to 100.

The kernel accesses variables such as **bss_start** or function such as **memzero** using absolute kernel addresses. In other words, absolute base kernel address starts at 0xffff0000_00000000 as hardcoded and defined in the header. The linker will think that the image is going to be loaded at 0xffff0000_00000000 and the kernel will just offset from that base address to get the absolute kernel address of variables such as **bss_start** or functions such as **memzero.**

5. (2)

Right after kernel switches to EL1 and clears the BSS, the kernel populates its pgtables via __create_page_tables:

```
// boot.S
__create_page_tables:
   mov    x29, x30       // save return address
```

The first thing __create_page_tables does is to save LR (i.e. x30) to x29. LR points to the address that __create_page_tables will return to. Normally, a function would save LR on its stack; this avoids losing the LR value when this function invokes another function with instruction BL, which overwrites LR; when such a function returns, it pops the saved LR value from its stack.

Is it a bug that we save LR to x29?

No, it is not because no code uses the register **x29**. In addition, it saves the return address of the callee. In addition, the **x30** register will get overwritten multiple times by other function calls.

Instead of saving LR to x29, can we save LR by pushing it to stack? Try it out yourself and explain your observation.

In the **__create_page_tables** assembly function, I called *stp x30, xzr, [sp, #-8]* which stores the value in the **x30** register to the stack. When running, the kernel simply crashes. This is because the **sp** value is not initialized yet.

Changelog

Jan 2024. Clarification.