In the second milestone for the course, the goal is to learn how to initialize the MMU, and creating and switching to the first user-level process. This is still relatively low-level but nonetheless an important part of any modern operating system. We’re moving towards a basic OS with multiple processes, system calls, and virtual memory.

So far, the code you have been running on the Pandaboard has had the MMU disabled: addresses in the code have exactly corresponded to physical addresses in memory. During this assignment you will enable the MMU, which means that all addresses will be virtual (and will therefore require translation). You will also turn on the processor caches.

1 Step 1: Set up page tables for the kernel

The first step is to decide how you want your page tables to look. As described in the lectures, the OMAP4460’s processor provides you with two translation table base registers (TTBRs) that indicate the addresses of L1 page tables. Using the Translation Table Configuration Register (TTCR), you can indicate which addresses get translated using which TTBR.

In addition to the lecture material, see the ARM Architecture Reference Manual for ARMv7-A,R, chapter B3 for an in-depth discussion of the ARMv7 Virtual Memory System Architecture. The most interesting bits (for this step) are sections B3.3.2 (“Translation table base registers”), B3.2.3 (“Enabling and disabling the MMU”), and B3.3 (“Translation tables”).

When designing your address space layout, you’ll want to bear in mind that soon (in fact, Step 4 below), you’ll be running user programs as well as the kernel. As we saw in the lectures, many operating systems map the kernel in one part of the address space (protected so that user programs can’t get at it), and the user address space in the rest. That way the kernel’s address space mappings don’t change when different programs run.

After you’ve decided on a page table layout, you need to allocate space for the initial bootstrap page tables and write the physical address of the start of each page table into TTBR0 and TTBR1. Also you will need to configure which addresses are translated by which base register by using the TTBCR register.

Note: we provide you with are functions that modify the translation table base and control registers. You can find them in kernel/include/arch/armv7/cp15.h.

Finally, you need to enable the MMU. The function to do this (the one shown in the lectures) is cp15_enable_mmu(), and it can be found in kernel/arch/armv7/cp15.S.

As soon as you do this, bear in mind that the addresses of lots of parts of the memory map will change (indeed, they might not be there at all any more).
If you look at the physical memory map of the OMAP4460 (it’s Table 2-1 on page 293 of the OMAP4460 Technical Reference Manual), you will see that DRAM starts at 0x8000000 (i.e. 2GB). The kernel is executing out of DRAM, so it would be a good idea to make sure that this area of memory is still there when the MMU turns on.

You will also see that the address of the UART is in the L4_PER region, which is somewhere different — it’s in the second quarter of the physical address space. Depending on how you have set up your address space, this might still be accessible now the MMU is on, or it might not. We provide a function to create a new mapping in the kernel’s address space for a region of physical memory, so you might need to use it to get the UART back. Without this, you won’t get any more console output. Note that it’s likely to be at a different address now…

2 Step 2: Setup an initial address space and control block and switch to init

The first process to run in user level over an OS is traditionally called “init”.

In order to start the first user level process, you need to create an execution environment for that process. This consists of:

- an address space (a set of page tables)
- a control block (an in-kernel structure identifying the new process)
- some physical memory containing the program itself

We provide most of the setup functionality and the layout of the control block. In particular, we will handle copying the program itself from the multiboot image we discussed in Milestone 0 to somewhere in memory from where it can be executed.

Your task in this step is to create a set of page tables for the Init process, setup the Init process’ registers according to the ARM calling conventions covered in the lecture, and read the provided code and get a basic understanding of the necessary steps for setting up the first process.

As setup functionality we provide expects all the L2 page tables to be in one contiguous block of memory, you should allocate them accordingly. We provide two functions: “alloc_phys” “alloc_phys_aligned” that allocate physical memory from a given start address.

**EXTRA CHALLENGE**

Allocate your L2 page tables as individual chunks. Note that you will have to adapt the startup functionality we provide to take into account your page table allocation scheme.

The init binary we provide will do a NOP system call. As your implementation does not handle system calls yet, this will halt your system with a “panic” message. However, DON’T PANIC! This particular panic message will show you that so far everything is working.

3 Step 3: Understanding the structure of a User-level program

As you can see, there are a collection of files in the source directory of the Init program. In this step you will get a basic understanding of the bits and pieces of a user-level program in our environment (which, as it happens, it not too different from other operating systems).
First off, have a look at /lib/crt/arch/arm/crt0.S. This file contains the entry point (the first code to be executed) for all user-level programs. Init is handled specially because in a larger operating system, as the first process, it would have to do some extra work. As in the OS, in crt0 we want to get out of assembly-land as soon as possible, so this happens as soon as we’ve setup the arguments to sos_init().

The implementation of sos_init() can be found in usr/init/init.c and currently it initializes some C library function handlers, sets up the command line arguments that a C main() function expects, and calls main() (which is where most people think a C program starts). From there on, execution works as you’d expect from a C program.

The remaining files in /usr/init/syscall contain an implementation of system calls using ARM’s software interrupt functionality. In particular, /usr/init/syscall/syscall.S contains the function that prepares the registers for the software interrupt and executes the swi instruction. In /usr/init/syscall/syscalls.c you can find the implementation of the NOP syscall. Note that the syscall numbers are defined in include/barrelfish_kpi/syscalls.h, because their values need to be agreed on between the kernel and user programs.

Note that in order to build the init program you need to update the AOS_MODULES variable inside the symbolic_target.mk file, located in your build directory. To include the module in your multiboot image you need to add it to menu.lst.aos which resides in your build directory as well (add the following line “module /armv7/sbin/init”).

4 Step 4: Print from init

As a final step you will have to change the provided init program (to be found in /usr/init/), to print “Hello, world” by implementing and using a new “print” system call.

Implementing a system call consists of two parts: one part is in the kernel and the other part is in the user program. You can find the kernel part – the “syscall handler” – in kernel/arch/arm/exn.c and the user part in /usr/init/syscall/syscalls.c and /usr/init/syscall/syscall.S.

Right now, the syscall handler provides parsing of the syscall number and the number of arguments to the syscall, as well as returning to the user process. The user part provides the actual syscall mechanism and an implementation of the NOP syscall.

A reasonable approach would be to first modify the syscall handler to print some kind of message when it gets a NOP syscall and then return to the user. To do this you need to extend the provided kernel syscall handler after removing the panic() line.

After that you can implement the sys_print function in libsos and add functionality to the syscall handler that acts on receiving a print syscall.

EXTRA CHALLENGE

You can try to integrate your print system call with the provided C library. To do that you need to implement a handler for libc-terminal_write_func.

In order to avoid having an implicit call to malloc() in printf(), you can use:

    setvbuf(stdout, your_buffer, _IOLBF, sizeof(your_buffer));

to provide printf with a static buffer at address your_buffer.

EXTRA CHALLENGE

Add another system call to turn the LED on or off. Wrap it up into a user-space library function, and demonstrate a program to use it.