1 Overview

In this milestone, we will develop a user-space driver for the PandaBoard’s UART serial port and use it for standard input/output of all user-space processes.

The work consists of:

- Making sure user-level threads work
- User-space device driver for the UART
- Ability to use user-space device driver from other processes
- A command-line interface that supports at least an echo command

2 Getting prepared

User-level Threads

The first step for this assignment is to make sure that your self-paging system works properly in the presence of multiple user-level threads. As you may have noticed by now, in the thread initialization code (lib/barrelfish/threads.c:thread_create_unrunnable()) the function paging_init_onthread() is called with the newly created thread as an argument. Your job is to implement (if you haven’t done so already) paging_init_onthread() by setting up exception stack and handler for the new thread.

If you have a working implementation for paging_init_onthread() you should be able to call thread_create to create multiple threads within init and memeater.

Device Memory Management

Device drivers are special programs because they need to access memory locations where the hardware registers are mapped. Capabilities are there to help you solve the problem by allowing user-space device drivers access to these memory locations in a safe manner.
The first step of getting a capability to the relevant registers for a user-level device driver is included in the handout for this milestone: the kernel gives the init process a capability which identifies the whole device memory address range (this capability is stored in the task cnode in slot TASKCN_SLOT_IO).

What you need to do now is to manage the address range identified by that capability. The inter-process interface to that second memory management system (which you can think of as the “RAM” server for device registers) is another RPC call (aos_rpc_get_dev_cap). While you won’t necessarily need this RPC call for the serial driver, you should still make sure that you can provide regions of device memory to other user-level programs.

Now is also the time to implement paging_map_fixed_attr if you haven’t done so already, as you will need to be able to map a given capability to some virtual address in your serial driver (using paging_map_frame_attr which in turn calls paging_map_fixed_attr).

### 3 Moving the UART driver to user-space

The first step of implementing the user-space device driver is to map the device address ranges in the driver process. As we don’t have user-level process management yet, we suggest that you implement the serial driver as part of the init process. As with the memory server for the last assignment, feel free to move the serial driver into its own process (in that case you should consider using the RPC call mentioned above).

You can have a look at the code in the kernel (or the OMAP Technical reference manual) to find the address range used by the UART device, and you should be able to map it using the infrastructure discussed in Device Memory Management.

Now, you can begin work on implementing a user-space driver. This is mainly about directly accessing the device registers from user-space to manipulate the device without having to go to the kernel all the time. You can have a look at the UART driver in the kernel as a reference point to learn how you have to interact with the UART device on the PandaBoard.

Doing so, you will see that the kernel driver uses a domain-specific language called Mackerel to generate code to access and update device registers. If you like, you can also use Mackerel to write the device driver in user-space. In that case you may have to understand how Mackerel generates the code, and how it is included by the build system. You can find more about it from Mackerel documentation.

At this point you should be able to implement serial output without much trouble.

### Input

After you get serial output to work, you should also implement serial input which will allow processes to request input from the user. There are two modes of processing input from the PandaBoard UART: polling and interrupt-based. The second issue with handling input is deciding how to multiplex the input between different competing processes and how to buffer input when no process is interested in it.

As you might imagine, using polling for input is easier to program than an interrupt-based system, so you should try to get serial input to work using polling first. For this you might want to use a dedicated thread that constantly polls the serial input line and buffers all the user input. User processes can then read that buffered input when they want to. You may need to play a few tricks with the thread scheduler so that you don’t loose input characters because the polling thread isn’t scheduled frequently enough.

At this point, you should be able to buffer characters that are sent over the serial port in user-space.

**EXTRA CHALLENGE**

Change your serial driver to use interrupts. You can find details on UART operation for the OMAP4460 SoC in the Technical Reference Manual for the OMAP4460 in section 23.3 UART/IrDA/CIR. In partic-
ular sections 23.3.5 and 23.3.6 which talk about the programming model and describe the register set respectively are of interest.

Also, to be able to handle those interrupts in your user space UART driver, you’ll have to register for and handle the upcall that is generated in response to getting a hardware interrupt in kernel mode (see `kernel/arch/arm/exn.c` for the kernel interrupt handler).

You can find complete overview of what the PandaBoard UART can do in Section 23.3.5 of the Technical Reference Manual for the OMAP4460. If you want to use some of the more advanced functionality, we recommend looking at the fairly complete Mackerel device definition for the PandaBoard UART in `devices/omap/omap_uart.dev`.

4 Using the user-space serial driver from other processes

Now that we have a serial driver, we want to make sure that all processes can use this new serial driver. For this, we will add a pair of new RPC calls that allow any process to write and read characters to and from the serial driver. Those two RPC calls are listed below, and the skeletons are provided in the code handout.

   a)aos_rpc_serial_putchar
   b)aos_rpc_serial_getchar

Any application should be able to use these calls for input/output of characters.

Use RPC calls for libc functions

In this step, you will modify your system such that standard input and output requests (printf and scanf) from all applications will go to the user-space UART driver instead of the kernel. You can start by making sure that the RPC calls work in `memeteer`.

Note that you should not modify `debug_printf` to use the user-space serial driver so that you have a printf function that you can fall back to when you a) know you don’t have a connection to the serial driver yet, and b) you are in a part of the code where you can’t communicate with the serial driver. This will also be helpful for you to see what is going on should the communication RPCs not work. Additionally, you should make sure that it’s possible to visually distinguish between output that uses the user-space serial driver and output that uses `sys_print()`.

At this point, all user-space output that isn’t debugging output should be handled by your serial driver.

5 Implement a command-line interface

You should extend `memeteer` with a small command-line interface (a “shell”) to provide a way to interactively test input and output using your serial driver. This command-line interface should at least include the following commands:

   • echo which takes a string of characters and echoes them back
   • run_memtest which takes the size of the memory region to test as an argument and runs the memory test in a user-level thread.
6 Lab demonstration

You are expected to demonstrate the following functionality during the lab-session:

- That you can print characters using the user-space UART driver.
- That your user-space UART driver is able to accept input.
- That all other applications are able to use the user-space UART driver for standard input and output.
- A simple command-line interface that provides at least the commands `echo` and `run_memtest`.

Try to make sure your code is bug-free. We’ll expect that implementing the functionality required for this milestone doesn’t break the memory server you implemented for the last milestone.

Once your code is demonstrated, you should submit it over the submission system.