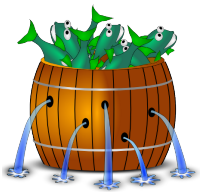

ADVANCED OPERATING SYSTEMS
Milestone 7: Command-line interface (Shell)
Fall Term 2016

Assigned on: **25.11.2016** Due by: **16.12.2016**

1 Overview

In this project, we will develop a command-line interface (shell) with which you can demo various parts of your implementation during either the milestones or any added functionality from the project part.

The work consists of:

- Reading from the console
- Ability to use UART device driver from other processes
- A command-line interface that supports a set of commands

2 Getting prepared

Input

In milestone 3 you implemented the terminal writing function. For this project, you should also implement serial input which will allow processes to request input from the user. In order to get this working, you should implement another syscall that reads a character from the kernel UART driver.

Since we already use `init` as a ”terminal” driver for printing, you can do the same for reading characters. The current driver implementation is interrupt driven. In order to handle the interrupt generated when a user types, we have to install an interrupt handler. How to install an interrupt handle is shown below

lib/aos/inthandler.c

```
inthandler_setup_arm(interrupt_handler_fn handler,
                     void *handler_arg,
                     uint32_t irq)}
```

The UART irq number is 37. When the UART now generates an interrupt, the handler is called which should read from the UART using the kernel syscall.

You will have to buffer the characters from the serial that are generated when no one tries to read from it. Furthermore, when multiple domains read from this buffer, the buffered characters might be split between multiple domains. You have think of a way to multiplex the buffered characters such that a read that requests multiple characters is not split (**Hint:** *Section 21* [1])].

Using the user-space serial driver from other processes

Now that we can read characters from `init`, we want to make sure that all processes can use this new serial driver. For this, we will add a new RPC call that allow any process to read characters from the serial driver. The RPC call is listed below, and the skeleton is provided in the code handout.

a) `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 order to use your own terminal read functions you need to set the libc function pointers accordingly. Change the lines in `barrelfish_libc_glue_init` to point to your new read functions. For this milestone you need to implement the terminal read functionality.

lib/aos/init.c

```
void barrelfish_libc_glue_init(void)
{

    _libc_terminal_read_func = aos_terminal_read;
     /* Already implemented */
    _libc_terminal_write_func = aos_terminal_write;
    /* ... */
}
```

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

You can find complete overview of what the PandaBoard UART can do in *Section 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`.

EXTRA CHALLENGE
Move 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. We suggest that you implement the serial driver as part of the `init` process. As with the other services form previous assignments, feel free to move the serial driver into it's own process.

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. 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. 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`).

You will need to receive interrupts from the UART device in your new process. This requires setting up the interrupt descriptor table and register an endpoint with an interrupt number. In order to register for interrupts, you need to invoke a capability (`IRQ_CAP`). This capability is passed to `init` and is located in the task CNode in `TASKCN_SLOT_IRQ`. You need to acquire it from `init`, either throught an RPC or by modifying the domain spawning. As soon as you have this capability in your CSpace, the function `inthandler_setup_arm` in the driver will succeed, as in init itself.

3 Implement a command-line interface

You should implement a separate process with a small command-line interface (a "shell") to provide a way to interactively test input and output. The command line interface reads from the command line using `getchar()` and should buffer them until the command is complete and is terminated with a newline or carriage return. If you make a spelling mistake during typing a command, you should be able to delete characters from the command line using the backspace key. The basic functionality we expect is from the command line interface is

- `echo` takes a string of characters and echoes them back.
- `led` turn on/off LED.
- `threads` a demo of working user-level threads.
- `memtest` takes the size of the memory region to test. as an argument and runs the memory test in a user-level thread.
- `oncore` run an application on a certain core (including passing arguments).
- `ps` showing the currently running processes.
- `help` show the available commands.

These commands should be run directly in the shells process. Additionally we expect that when a command is entered that matches a name from an available binary, the binary should be started (e.g. memeater).

Furthermore we expect you to implement some more commands related to file systems. If your group does not implement a file system as a project, you can simulate a non persistent file system in RAM. For simulating the file system we proved the code in `lib/fs/`. The get the RAM file system working you have to implement the two functions `filesystem_init()` and `filesystem_mount()` in `lib/fs/fs.c`. The file system will be stared locally and only accessible by the shell. The commands you should implement are shown below:

- `pwd` shows the current working directory.

- `cd` changing the working directory.
- `ls` show the contents of the current directory.
- `cat` print the contents of a file.
- `wc` print newline, word, and byte counts for each file
- `grep` search a file (or with -r a directory) for a certain word or string.

You are free to extend your shell with more commands that can help you to demonstrate other parts of your system.

EXTRA CHALLENGE
Implement an autocomplete function that tries to suggest available commands and directories/files. If there is only one available suggestion, complete the command.

EXTRA CHALLENGE
Implement I/O redirection e.g.

- **> and <** example: `echo hello > file1` writes the contents of "hello" into file1.
- **>> and <<** example: `cat file1 >> file2` appends the contents of file1 to file2.
- **|** example: `cat file1 | grep <string>` which pipes the output of cat file1 into grep.

EXTRA CHALLENGE
Make the file system accessible to other domains. If another member of your group implements a file system, you should synchronize on what you are doing for this extra challenge.

4 Lab demonstration

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

- Your UART driver is able to accept input.
- That all other applications are able to use the UART driver for standard input and output.
- A command line interface that implements the commands listed in 3

Once your code is demonstrated, you should submit it over Online submission system (accessible only from ETHZ network) on Friday 16.12.2016 midnight (23:59h).

5 Resources

1) OMAP4460 Technical Reference Manual