ADVANCED OPERATING SYSTEMS
Milestone 7: File-systems and ELF loading
Fall Term 2016

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

1 Overview

In this milestone you will bring together the bits and pieces you've implemented so far and make the last steps towards a complete operating system. You'll receive a simple block driver for the SD-Card reader on the Pandaboard and write a filesystem for the driver. After that, you will use your file system to read ELF binaries and be able to spawn them as processes inside your system.

This exercise will be the last exercise where you'll need to implement something. We will leave you much freedom in how you do the things we want from you as a minimal requirement, and also give you a lot of room for additional bonus objectives. In any case, try to finish the minimal requirements first before starting with other projects.

The work consists of:

- Implement a file system on top of a block device
- Handle reading of files and directories
- Integration of your filesystem implementation into the standard functions such as `fopen` and `fread`
- Support loading of ELF files from SD-card

2 Getting prepared

For this milestone, you will need to pull the new changes from the repository. This will add a new domain (`/usr/drivers/omap44xx/mmchs`) and new libraries (`/lib/fs`) to your source tree and to the list of modules to be built. The filesystem library (`/lib/fs`) contains a simple implementation of a ramfs an in memory filesystem. You may have a look which functions you will need to implement. We provide you with a simple test utility (`/usr/test/filereader`) that you can use during development.

Again, make sure `/usr/drivers/omap44xx/mmchs` is also added to your menu.lst. Run `Hake` again and do a clean build:

commands to run

```
make rehake
make clean
make PandaboardES
```

3 Obtaining Device Capabilities

The provided SD-card driver requires a capability to the register regions of the device. In order to use the device itself, it needs power and clocks which need additional capabilities. The driver uses the following RPC call which you will need to implement. This will request a capability for a specific memory region. See the function `map_device_register` in `main.c` of the driver. We will provide you with a function that uses a privileged capability operation to create the requested capability.

EXTRA CHALLENGE
Init is given the capabilities to the physical memory, including the device regions. You will find the device capabilities in a specific location in init's capability space. In this extra challenge your task is to implement a more sane way to deal with device capability request. You will need simiar functionality as you did with the physical memory management already, but additionally support allocating a *specific* region.

include/aos/aos_rpc.h

```
/*
 *
 * param chan the rpc channel* param paddr physical address of the device
 * param bytes number of bytes of the device memory* param frame returned frame
 */
 errval_t aos_rpc_get_device_cap(struct aos_rpc *chan, lpaddr_t paddr,
                                 size_t bytes, struct capref *frame);
```

4 The Filesystem

Traditionally, file-systems build on top of a block device driver that is able to read and write byte buffers (usually 512 bytes), so called blocks, from and to a disk, flash drive, SD-card or another storage medium. In this exercise you'll receive a block driver for the SD-card reader on the Pandaboard. The driver has a simple interface to read and write one block at the time. Use the API in `usr/mmchs_driver/mmchs.h` to interface with the driver.

4.1 The MMCHS Driver

When the driver starts it enables the clocks and power to the MMCHS device and then initializes it. The driver will export the following interface to be used by your filesystem implementation.

usr/drivers/omap44xx/mmchs/mmchs.hh

```
errval_t mmchs_read_block(size_t block_nr, void *buffer);
errval_t mmchs_write_block(size_t block_nr, void *buffer);
```

The two functions will read or write blocks of 512 bytes in size to and from the SD card. The `buffer` argument points to a pre-allocated buffer in the domain's address space which is at least 512 bytes in size. If needed, you can extend this interface to support reading or writing multiple blocks at once (see extra challenge).

EXTRA CHALLENGE

Unfortunately, the block device driver you'll get is very simple and slow. Fortunately, during this course, you have now mastered all the necessary skills to write a fast device driver. We will award bonus points for any improvements you can do in the driver. This includes for example reading / writing more than one block at the time, or using DMA and interrupts to transfer data to and from the card instead of polling. Be sure to measure the performance improvements you get from your changes and show them to us in the presentation.

All the necessary information to complete these assignments can be found in the OMAP4460 TRM. Most relevant parts are Chapter 24, especially the MMC/SD/SDIO Programming Guide in Section 24.5 and the SD Specification Part 1 (interesting Sections are 4.7.4 and 7.3.1.3). Both manuals can be found on the course website.

4.2 FAT32

Your task is to write a file-system for the block device driver. We want you to implement a file-system that can read a FAT32 partitioned SD-card. You find a link to the official FAT32 specification from Microsoft on the AOS website. FAT is a fairly simple file-system (the specification is only 34 pages!) and therefore well understood and well documented.

In order to test your file-system, you need to create a FAT32 file system on the SD cards we handed out to you. On a Linux system, you do that by issuing the following command:

```
mkfs.vfat -I -F 32 -S 512 -s 8 /dev/sdX
```

This command creates a FAT image on the SD-Card with a logical sector size of 512 bytes and 8 sectors per cluster. Make sure you use the right device from `/dev` when formatting! You can use `dmesg` to figure out which device was recently mounted. Once you have partitioned the card you can start the provided mmchs driver from the hand-out. If everything worked correctly, the driver will print the contents of the first sector. You can easily verify that you're dealing with a FAT disk in case the first byte on the block is 235 (0xEB) and the third byte is 144 (0x90).

You can assume that there will be just one partition on the SD-card.

FAT filenames are restricted to 11 characters without long-filename support (VFAT). You can just use small direntries in your file-system implementation. Note, that the filename has to be unique and UPPERCASE. In the specification there is a algorithm to transform any name into a suitable representation. Keep in mind, that this short name doesn't distinguish between upper and lower case. For your initial implementation you can assume that all filenames are unique and there are no collisions when creating the short name.

EXTRA CHALLENGE
Add the support for long file entries. The FAT specification also describes long directory entries which support up to 255 unicode characters. You can implement this support as an extra challenge to actually have a distinction between upper and lower case and support longer file names.

EXTRA CHALLENGE
With only short directory names, some filenames end up in a collision with each other, e.g. mylongfile.txt and mylongfile2.txt will map to the same short name MYLONGFI.TXT which is obviously a collision. The FAT specification describes how to handle such a case. The two would get a short file name as MYLONG∼1.TXT and MYLONG∼2.TXT. In this extra challenge you are required to handle such short directory entry collisions.

4.3 Design Space

As already stated, we give you freedom on how you want to implement the support for your file system. In the end you will need to have the functionality that any domain in your system can call `fopen` and `fread` for instance (see setup below). There are multiple ways to implement this functionality. Make sure you will explain and describe your design choice in the final report with enough detail. In any case, you will need to serialize the access to the SD-card device to avoid race conditions when accessing blocks. Clearly, direct access to the SD for any domain is not practical and you will need to run the SD-card driver in a separate domain.

This requires a way to find the driver service. You can either implement some basic name service functionality or to store endpoints to the file service at a well know location.

For each variant there will be a different set of RPCs you will need to implement. For instance, if you decide to implement a file service you need to provide different RPC calls than if you decide to implement the filesystem as a library. In the latter case you just export block read and write functions for instance. Make sure you also explain the RPCs you've added.

4.4 Initializing the Filesystem

We provide you with a stub library (`lib/fs`) which you can implement. The library currently provides two function definitions:

```
/**
* brief initializes the filesystem library** return SYS_ERR_OK on success
*         errval on failure
*
* NOTE: This has to be called before any access to the files
*/
errval_t filesystem_init(void);

/**
* brief mounts the URI at a give path** param path  path to mount the URI
* param uri uri to mount** return SYS_ERR_OK on success
*         errval on error
*
* This mounts the uri at a given, existing path.
```

```
*
* path: service-name://fstype/params
*/
errval_t filesystem_mount(const char *path, const char *uri)
```

These functions are supposed to do all the initialization you need for your file-system. See the comments in the function bodies about the steps you need to do. Depending on your design, you will need to connect to the block or file-system service and initialize the file and directory functions.

The second function will mount the SD card into your file system. See `/usr/test/filereader` for the intended use of the two functions.

Hooking your file functions into the C library You will need to tell the C library (newlib) and libaos how to open files and directories and how to access them. You can do this by calling the following functions:

newlib function registration

```
void
newlib_register_fsops__(fsopen_fn_t *open_fn, fsread_fn_t *read_fn,
                        fswrite_fn_t *write_fn, fsclose_fn_t *close_fn,
                        fslseek_fn_t *lseek_fn);
void aos_register_dirops(aos_mkdir_fn_t mkdir_fn, aos_rmdir_fn_t rmdir_fn,
                         aos_opendir_fn_t opendir_fn,
                         aos_readdir_fn_t readdir_fn,
                         aos_closedir_fn_t closedir_fn,
                         aos_stat_fn_t stat_fn)
```

This will initialize function pointers to be used by the C library accordingly to re-route stdio calls to your implementation.

Note, the file functions will take file descriptors which is of type int. We provide you with a simple FD table implementation which is already used for ramfs. You will need to integrate your implementation with this file descriptor table.

4.5 Accessing the Filesystem

We require you to implement the basic support for a read-only filesystem. This includes operations such as

- Opening of existing files `fopen`
- Reading file contents: `fread` and friends
- Positioning the cursor `ftell`, `fseek` and friends
- Listing contents of directories: `opendir` and `readdir`
- Obtaining information about the file: `fstat` and friends

Every access essentially boils down to figuring out which blocks of the SD card to read and how to interpret them. Directories store long and short directory entries whereas files store the file contents. The size of a file is stored in the corresponding directory entry in the parent directory sector.

Testing We provided you with a small test program (`/usr/test/filereader`) that does read-only file and directory operations on the SD-card. The test is not complete and you can extend it. You will find a script in `tools/prepare-sdcard.sh` to create the folder and directory hierarchy.

EXTRA CHALLENGE
In this extra challenge you will have to implement write support for the file system:

- Opening (creation) of non-existing files `fopen`
- Writing file contents: `fwrite` and friends
- Truncating files `ftruncate`
- Creating new directories
- Removing files and (non-empty) directories.

This will require to allocate new unused blocks/clusters and free no-longer needed ones. You will also need to be able to extend existing cluster chains with the newly allocated blocks in order to grow files.

The file allocation table holds information on all the existing clusters in the system. An entry is either free (0x000 0000), points to the next block in the chain, or marks an end (0xfff fff8 / 0xfff ffff). Note the top 4 bits are reserved in the entries.

4.6 Performance Evaluation

As a last step you'll have to evaluate the read performance of your file-system. Your evaluation should at least contain the maximum read bandwith you achieved. You need to be able to explain your results, where the potential bottlenecks are in your implementation and how you could improve them.

We provide a very simple driver for the Cortex-A9 global timer (see Cortex-A9 MPCore TRM, p4-8ff). To use this driver you'll need to add the `omap_timer` library to the program (in `addLibraries` in the `Hakefile`) in which you will use it. Here's a very simple example:

```
#include <omap_timer/timer.h>

int main(void)
{
  // initialize the timer
  omap_timer_init();
  // enable the timer
  omap_timer_ctrl(true);
  uint64_t start = omap_timer_read();
  // do work
  uint64_t end = omap_timer_read();
  uint64_t time = end - start;
  // do analytics on elapsed time.
}
```

5 ELF loading

Now that your file-system is working, you are able to add ELF loading support to your OS. Until now, your build system assembled a multiboot compliant boot image at compile time which contained a set of modules (i.e., the ones you specified in your menu.lst file inside your build directory), those modules are already just regular ELF files, therefore you can think of it as an archive of ELF files. In a real system we do not want to load only binaries that ship with the kernel. Using your file-system, you can now add support to load ELF files from an SD-card. That involves loading an ELF file from your card into memory, then doing necessary ELF relocation and setting up the the address space of your newly allocated process in a way that it can execute the binary. Note that how you implement this part and also how much work you have to do for this assignment depends a lot on how your system looks like at this point (especially the process management, your file and memory management subsystems).

We will provide you with code that already does much of the ELF handling for you (have a look at the ELF library in `lib/elf`). For an example of how to use the library, you can have a look at how the kernel code loads the init process from the multiboot image.

6 Lab demonstration

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

- You have an implementation of a read-only FAT32 file-system. You can demonstrate this by running a program that reads from files and displays the directory content.
- You can demonstrate that you are able to load ELF images from your SD-Card and spawn it as a new process using your command line interface.

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).