Implementing a block device driver

Objective: Implement a working block device driver

After this lab, you will be able to

- Learn how to register a block device driver in the kernel
- Handle I/O requests in your driver
- Add asynchronous handling of I/O requests

Setup

Go to the `/mnt/labs/linux/block/` directory.

Extract the latest 2.6.30.x kernel sources in there, configure them for the ARM architecture and with the configuration file supplied in the `data/` directory, and compile them.

Boot this kernel on your CALAO board, using the supplied `nfsroot/` directory as root filesystem through NFS.

In the `src/` directory of this root filesystem, a kernel module skeleton is present.

Register a block device driver in the kernel

Using the block driver APIs presented in the training materials, implement what is required to register a block device driver and a single block device to the kernel.

For the moment, the `request()` operation can be left empty. You should be able to see your block device in `/sys/block/`, and if you read/write it using the `dd` command, it should block indefinitely because the `request()` operation is not implemented yet. Make sure that everything is properly cleaned up in the module exit function, and try to load and unload your module a few times.

Handle I/O requests

For this simple block device driver, we are going to create a single ramdisk. Using the `vmalloc()` allocator, you can allocate a fairly large region of memory, of a few megabytes for example, in the module init function, and free it in the module exit function: the contents of our ramdisk will be kept until the module is unloaded.

Then, implement the `request()` operation in a synchronous way, by simply using `elv_next_request()` and `__blk_end_request()`. `memcpy()` should be used to read and write to the memory region allocated to store the ramdisk informations.

Once done, you should be able to test your block device with `dd`, but also format it with `mkfs.ext2`, mount it, and use this new filesystem as usual!

Asynchronous operation

Now, to test the asynchronous capabilities of the block driver API, we are going to handle our requests in a function that is regularly called by a kernel timer.

Using the timer API described in `<linux/timer.h>`, set up a timer

that runs a function every second. Create a linked list, protected against concurrent accesses by a spinlock. In the `request()` operation, dequeue the requests from the request queue, and add them to the linked list. In the function triggered by the timer, handle each request of the linked list, and call `blk_end_request()` upon completion.

The block device should still work properly, but will of course be slowed down quite a lot: the requests might be delayed until the next call of the timer function, that is during at most one second.