COMP3301/COMP7308 Practical 4

This practical will go through the boilerplate necessary to implement kernel side of a character device driver interface.

This prac asks you to implement a Hello World driver.

1 Device Special Files

One of the initial goals of the Unix operating system was to provide a consistent set of operations for dealing with traditional files, and also for manipulating devices and drivers. In practical terms this meant you could to use `open(2)`, `read(2)`, `close(2)`, and so on to access both files and device files.

Device files in turn are divided into block and character device files, which refers to the data transfer characteristics of the underlying interface and hardware. For example, serial devices provided data one byte or character at a time, and are therefore character devices. Disks devices generally present an interface that allows for a block of data to be transferred at a time, rather than transferring individual bytes of data, and are therefore considered block devices.

Instead of a normal file that refers to an inode which in turn refers to a series of bytes within a filesystem, device files refer to a set of functions in the kernel that implement behaviours such as open, close, read, and write. These functions are addressed by the type of device file (block or character), and a major number.

Read the definition of `struct cdevsw` in `src/sys/sys/conf.h`, and find the `cdev_ksyms_init` macro.

Each architecture supported by OpenBSD has their own set of block and character majors, though they generally share most of their functionality.

Find the `cdevsw` array for the amd64 architecture in `src/sys/arch/amd64/amd64/conf.c`, and paste a copy of the `cdev_ksyms_init` macro above it. Rewrite the `cdev_ksyms_init` macro to refer to `hello` instead of `ksyms`, and add it to the `cdevsw` array in slot 21, passing 1 as the first argument to the macro.

Add `cdev_decl(hello);` above the `cdevsw` array to prototype the functions referred to by the `cdev_hello_init` macro.

2 Hello driver implementation

Rather than create a new source file and add it to the build system for our hello driver, let's add it to an existing file. This should avoid the need to recompile the whole kernel.

Add the following to `src/sys/kern/subr_xxx.c` and fill it in:

```
#include <sys/fcntl.h>

static const char *hellostring = "hello string\n"; /* make this your own */

int
helloopen(dev_t dev, int flag, int mode, struct proc *p)
{
	/* code goes here */
}

int
helloclose(dev_t dev, int flag, int mode, struct proc *p)
{
        /* close should never fail, but Arla ruined things */
        return (0);
}

int
helloread(dev_t dev, struct uio *uio, int flags)
{
        /* code goes here */
}
```

`helloopen()` should restrict access to only minor 0, and only if the device is opened for reading, not writing. Refer to `ksymsopen()` in `src/sys/dev/ksyms.c` as a template.

`helloread()` should use uiomove to output some or all of `hellostring` as possible (excluding the terminating `nul` character). Again, refer to the ksyms code as a reference.

3 Accessing the device

Use `mknod(8)` to create a character device file for the hello driver under /dev, and read it using `cat(1)`.

4 Prac Marking Criteria

The pracs in this course are marked on a pass/fail basis. This means you must demonstrate to your tutor sufficient understanding and functionality before progressing. If you fail to demonstrate sufficient understanding through completing the task, then you will fail this prac. You will not be allowed to repeat this prac. All code demoed and assessed for the prac, must be your own work.

To pass this prac you must demonstrate that you have modified, built, and run an OpenBSD kernel that generates custom output via `cat /dev/hello0`.