Workshop 2

In this workshop the goal was for us to make a filesystem that logs whenever a file is read and/or written to, along with a device driver to interface with an Arduino. However, since the assignment also requires us to use the code from Workshop 1, which was not made using an Arduino but with a stub program, no device drivers were written.

FUSE

A filesystem keeps track of which files and directories are in it. It contains information such as the path to a file, the id of a file, whether the file is a directory or not, the size of the file and so on.

FUSE (Filesystem in User-Space) consists of two components, which are the fuse kernel module and the libfuse userspace library. When mounting a filesystem using the libfuse library, the libfuse kernel module will be used as a bridge between the filesystem and the linux kernel. Whenever an incoming request from the kernel is received, it will be passed to the main program using callbacks. In this workshop we will use this for logging.

Implementation

For this workshop a FUSE filesystem was modified so that it would log whenever a file was read, opened or written to. The filesystem used in this workshop is a slightly modified version of the big brother filesystem: `https://github.com/pierreis/bbfs`

The directory mounted is a directory containing the program made in workshop 1, which generates temperature data and stores that data in a file called "data". The filesystem then monitors when any file in the mounted directory is opened, written to or read from.

Mounting

To mount the filesystem open the terminal and mount using the following format:
"./bbfs [FUSE mount options] rootDir mountDir logFile"
Which for our filesystem would be:
"./bbfs -f ./WS1_Program ./MountDir ./logFile.txt"
aThe -f option makes it possible to dismount the filesystem using a keyboard interrupt instead of having to write in the dismount command in the terminal. Furthermore, it allows the fuse program to print into the terminal.

Initialization

When the FUSE file system is mounted it defines the operations as seen in listing 1. Here it assigns which functions to run when a certain operation is done in the filesystem, e.g. when a file is opened it runs the bb_open function. These functions determine what will be done whenever the given operation is performed.

```
1  struct fuse_operations bb_oper = {
2      .mkdir = bb_mkdir,
3      .rmdir = bb_rmdir,
4      .opendir = bb_opendir,
5      .readdir = bb_readdir,
6      .getattr = bb_getattr,
7      .rename = bb_rename,
8      .open = bb_open,
9      .read = bb_read,
10     .write = bb_write,
11     .init = bb_init,
12     .destroy = bb_destroy,
13 };
```

Listing 1: Some of the filesystem operations

Operation example

The bb_open operation can be seen in Listing 2. This operation will run whenever the filesystem receives an open request from the kernel. On line 6 is the function logging that a file has been opened. In line 7 it then updates the fullpath, which is the path from the rootDir (In this case the mountpoint of the filesystem) to the file opened. In line 9 it uses log_syscall which returns a file descriptor for the file at the path "fpath". In line 14 it assigns the file descriptor to fi->fh.

```
1  int bb_open(const char *path, struct fuse_file_info *fi) {
2    int retstat = 0;
3    int fd;
4    char fpath[PATH_MAX];
5
6    log_command("Open(path\"%s\", fi=0x%08x)", path, fi);
7    bb_fullpath(fpath, path);
8
9    fd = log_syscall("open", open(fpath, fi->flags), 0);
10   if (fd < 0) {
11     retstat = log_error("open");
12   }
13
14   fi->fh = fd;
15
16   log_fi(fi);
17
18   return retstat;
19 }
```

Listing 2: Open operation

Logging

```
1 Jan. 09. 2022 Time:[22:05:30] || Open(path"/data", fi=0x01a61cf0)
2 Jan. 09. 2022 Time:[22:05:30] || Write(path="/data", buf=0x01240060,
    size=4096, offset=0, fi=0x0123ed50)
3 Jan. 09. 2022 Time:[22:05:30] || Write(path="/data", buf=0x00a1d060,
    size=4096, offset=4096, fi=0x00a1bd50)
4 Jan. 09. 2022 Time:[22:05:30] || Write(path="/data", buf=0x01a63060,
    size=4096, offset=8192, fi=0x01a61d50)
5 Jan. 09. 2022 Time:[22:05:30] || Write(path="/data", buf=0x00a1d060,
    size=1428, offset=12288, fi=0x00a1bd50)
```

Listing 3: logFile example

In Listing 3 is an example of what happens when data is logged to the logfile. This example show what happens when the code from workshop 1 writes to the file called "data". In the first line we can see how the file at the path /data is opened. This is the file in which data from the workshop 1 code is stored. After that we can see that four write operations are done. Here we now have some more information, which is the buf, the size, and the offset. The buf is the id of the buffer from which it writes data. The size is the size of the buffer content, and the offset is the offset of which the data is written. The reason for write operations to be taking up four logging entries when logging is due to the way the workshop 1 code was made, in which the buffer size couldn't exceed 4096. It therefore have to write the data in four operations.

Drivers

Kernel modules and device drivers

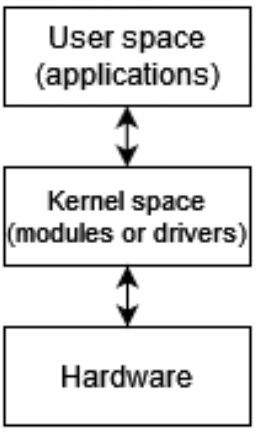
The Linux kernel is the main component of the operating system, which is the core interface between the hardware of the computer and the programs it is running. It is a large body of executable code which defines many types of interactions and functionalities. The functionality of the kernel can be expanded upon by way of loadable kernel modules, which can be dynamically inserted into the kernel at runtime. This means that the Linux system does not have to restart if new functionality needs to be added.

Modules are made up of object code which can be dynamically linked to the running kernel by using the *insmod* and *rmmod* programs. Device drivers are a type of kernel module, which offer access to hardware as it is added to the system. This could for example be a USB mouse, or a Software Defined Radio. Device drivers can be thought of as a software layer that lies between applications and the actual hardware device - providing a layer of abstraction. They contribute the mechanism of accessing hardware devices, but do not impose policy on how that hardware device should be used.

It is worth mentioning that hardware controllers usually interface the actual hardware, and the device drivers handle the data coming from the hardware controllers as part of the kernel. For example, a UART (Universal Asynchronous Receiver Transmitter) hardware device inside a USB serial port will handle data transmission and receipt with a fitting data bus, while the device driver handles the actual data with its specified design. The device driver's task, then, is to handle buffering of data, interrupt handling, and error handling.

User space and kernel space

When writing device drivers, the distinction between *kernel space* and *user space* is important. All system calls reach the hardware in some way at some point, but the kernel space defines how these calls are managed. Device drivers and modules are part of the kernel space as soon as they are inserted. The kernel space provides the user space with a simple interface for accessing hardware, and device drivers serve to expand the kernel space's ability to do so. The user space is the memory allocated for running applications and user programs, which includes end-user programs like the UNIX *shell*, window managers, and whatever other programs the user might want to run. The applications in user space need to access the hardware, of course, and it does this through calls to the kernel space.



Figur 1: Structure of user space and kernel space

Types of device drivers

Three main types of device drivers exist - namely Char drivers, Block drivers, and Network drivers. In the following text, Char drivers and Block drivers will be explained.

Char drivers communicate by sending and receiving individual bytes of data. They are usually interpreted as a stream of bytes, like a file, in which the data from the hardware is read. The difference between a regular file and a char driver is that you can always access the whole file by moving back and forth in the data, while char drivers usually act as a data channel, which is accessed sequentially. An example of a char driver could be the driver handling a USB keyboard. The user presses keys on the keyboard, which transmits signals about which key is being pressed. A UART controller (as described before) handles the actual data transmission, while the device driver handles individual bytes. One way a device driver can handle this is by using a simple asynchronous ringbuffer.
Char devices are usually operated by file-calls, such as *open*, *read*, *write*, and *close*.

Block drivers are used for devices in which data is organized in blocks. A good example of this would be a disk drive, such as a CD and its hardware reader device. The hardware itself consists of a space for the disk which makes it spin fast and a laser unit which can read and sometimes write data to the disk, and it supports the same read, write, open and close calls as the char device. It also supports a *seek* call, which looks for a specific block of data using the appropriate algorithms to do so.
The block nature of the data in block devices means that it is possible to access larger portions of data at a time.

How to write and run a driver

When loading a device driver into the kernel, some preliminary tasks should be done. These tasks could include, but are not limited to, resetting the device, reserving memory space in RAM and reserving I/O ports. These tasks should be undone upon the module's removal from the kernel. When the *insmod* and *rmmod* commands are issued, they call to two functions which must be explicitly defined: *module_init* and *module_exit*. The initialization and exit functions can be defined and named however the programmer wants, but to be identified as the corresponding loading and exiting functions, they must be passed as the parameters of the functions *module_init()* and *module_exit()*. A simple hello world program can be seen in listing 4. It includes the *init.h*, *module.h*, and *kernel.h* header files, and initializes an init_hello and exit_hello function, and passes them as the parameters of the module functions described above.

```
1 #include<linux/init.h>
2 #include<linux/module.h>
3 #include<linux/kernel.h>
4
5 static int init_hello(void){
6     printk("Hello, I'm a driver!\n");
7     return 0;
8 }
9 static void exit_hello(void){
10     printk("Driver says goodbye!\n");
11     return;
12 }
13
14 module_init(init_hello);
15 module_exit(exit_hello);
```

Listing 4: Simple hello world program

Using a driver to interface with Arduino

One part of the assignment was to write a device driver to interface with an Arduino generating data periodically. The data it would be generating would be identical to the TempStub.c program written for Workshop 1, and used in this workshop with the filesystem. A pseudocode will be written for a char driver.

The Arduino would be connected through USB to the computer. Physically, a hardware controller is near the USB port, which handles the physical transmission and receipt of data. In this case, a UART controller would receive the incoming bytes and push them to its receiver shift register, an integrated circuit. The device driver then reads these bytes and puts them in a ringbuffer. The application, in this case the main.c program from workshop 1, would read a byte at the head of the ring buffer and then move the head's position one over. It is not necessary to delete the data from the ringbuffer when the data is read, as the device driver simply overwrites the data.

```
Initialize:
    Connect to arduino through UART
    Create ring buffer of size n

ReadHardware:
    Read one byte from UART
    Insert the read byte at the tail of the ring buffer
    // Tail is the tail of the ring buffer
    Tail = (Tail+1)%n
    if(BufferOverflow):
        // Head is the head of the ring buffer
        Head = (Head+1)%n

ReadFromDriver:
    if(BufferUnderflow):
        Return wait command
    else:
        Var = Byte at the head of the ring buffer
        Head = (Head+1)%n
        return Var
```

Listing 5: Pseudocode for driver interface for Arduino