COL331 Assignment 3: Hard Track

Aniruddha Deb
2020CS10869

April 2023

1 Implementation Overview

The implementation involves creating a single device driver which initializes two devices, a reader, located at `/dev/lifo_reader` and a writer, located at `/dev/lifo_writer`. Writing to the reader is disallowed, and reading from the writer is disallowed. The devices have read and write Permissions for all users.

Internally, we use a single structure with two device drivers, a 1MB buffer and a pointer to the head of this buffer to implement the driver. The writes reverse the order of the string while writing to this buffer (thereby making it LIFO), and the reads read in the order in the buffer. There is explicit synchronization to prevent race conditions, and two separate `file_operations` structures are created for the reader and the writer, both supporting different operations. The main struct for storing the driver state is shown below:

```
struct lifo_cdev_data {
    struct cdev reader;
    struct cdev writer;
    // in-memory 1MB buffer
    char* buffer;
    ssize_t head;

    // synchronization
    struct semaphore sem;
    uid_t owner;
    pid_t process;
};
```

2 Implementation Details

2.1 Module access rules

Only one user can have ownership of either kernel module at a given time, and only one process may have ownership of either the reader device or the writer device at a given time. This is to prevent race conditions that may lead to disorder in the stream

Therefore, we have the following rules:

1. Only one process can access either the lifo reader or lifo writer at any given instant
2. If another process tries to access the lifo reader or lifo writer when they're in use:

(a) If they're from another user, the request is rejected (we return -EBUSY) till all the tasks of the current owner are done

(b) If they're from the same user, the request blocks till the previous accesses are over.

3. If the same process tries to both read and write from the device, then since the reads block till the writes are over (and vice versa), the process may go into deadlock, since it may wait for a write and itself be unable to write as it's waiting. Hence, we prevent this by returning -EINVAL if this occurs.

The relevant code for this is as follows:

```
int __acquire_lifo_semaphore(void) {
    if (lifo_cdev.owner != current->loginuid.val && lifo_cdev.owner != 0)
        return -EBUSY;
    if (lifo_cdev.process == current->pid) return -EINVAL;
    down(&lifo_cdev.sem);
    lifo_cdev.owner = current->loginuid.val;
    lifo_cdev.process = current->pid;
    return 0;
}
```

2.2 Creating device files

To create a device file, a device region needs to be allocated first. This corresponds to a major number, and a range of minor numbers. We let the OS allocate us a major number, rather than explicitly setting it ourselves to avoid errors. This is done using `alloc_chrdev_region`. After this, the device class has to be created using the major number. Device files are then created using the `cdev_init` function and added to the OS using `cdev_add`.

After this, we allocate 1MB of memory for the kernel buffer and initialize the cdev semaphore, and set the devnode function to ensure that appropriate Permissions are set for the devices that have been created.

2.3 Preventing reads (writes) from writer (reader)

This is implemented by having the respective methods return an error code (-EINVAL) when someone attempts to run them. The Operating System takes care of the invalid return code and makes it unable to read/write from these files.

```
$ cat /dev/lifo_writer
cat: /dev/lifo_writer: Invalid argument
```

2.4 Compiling as a kernel module

To compile as a kernel module, the driver file was placed in the `drivers/char` folder, and a driver description was written in the `drivers/char/Kconfig` file for this driver. After this, the driver was enabled in the menuconfig and the kernel was compiled with this driver.

2.5 Other details

EOF was not explicitly set when there are no writers, as the EOF character is not 0x05 but is implementation defined. It is also error-prone to encode a byte as a termination character in a byte

stream as the probability that the byte stream itself contains that character will be high, and may lead to premature termination (especially if we're using the byte stream to encode multiple-byte wide characters.

EOF is detected by utilities eg. cat by returning 0 from the `reader_read` method when there are no bytes to read. In other cases, the buffer is zeroed by using the `clear_user` method.

3 Testing

Elementary testing was done by simple file-based I/O utilities in bash

```
$ printf "Hello" > /dev/lifo_writer
$ head -c 5 /dev/lifo_reader
olleH$
```

For writing an arbitrary number of bytes, a C++ program was written with the following pseudocode:

```
FILE *lifo_writer = fopen("/dev/lifo_writer", "wb");
int n_bytes;
while (1) {
        std::cin >> n_bytes;
        if (n_bytes == 0) break;

        char *buf = (char*)malloc(n_bytes);
        for (int i=0; i<n_bytes; i++) buf[i] = (char)i;

        int n_bytes_written = fwrite(buf, 1, n_bytes, lifo_writer);
        fflush(lifo_writer);
}
```

A reader was simultaneously used (which could read an arbitrary no. of bytes) to test the various access cases described above: the reader was simply `head -c <no. characters>`. `dmesg` was used to check the logs to see if the program was working correctly.