CEng 536 Advanced Unix
Fall 2025
Homework 3
Due: Jan 18th, 2024

1 Description

In this project you are going to implement a pseudo character device driver for Linux kernel resembling a simple frame buffer device. A frame buffer device is used to represent a text or graphical screen where each unsigned byte stores the two dimensional pixel or character data. Your frame buffer will provide read-write access to the rectangular regions (viewports) in the framebuffer area.

The rectangular area to be input/output will be selected via `ioctl()` calls to the device as:

```
struct fb_viewport {
    unsigned short x, y;
    unsigned short width, height;
};
...

/* 80x20 window at (10,10), total size 1600 bytes */
struct fb_viewport fba = { 10, 10, 80,20 };
int fd = open("/dev/fb5360","O_RDWR");
if (ioctl(fd, FB536_IOCSETVIEWPORT, &fba))
    /* handle error */;

char data[] = { 0, FF, 0, FF, 0, FF, 0, FF ,0 , FF};

/* following set all window with 0 255 pattern */
for (int i = 0; i < 160; i++) {
    write(fd, data, 10);
}
lseek(fd, 0, SEEK_SET);
/* now clean all to 0 */
data[1] = data[3] = data[5] = data[7] = data[9] = 0;
for (int i = 0; i < 160; i++) {
    write(fd, data, 10);
}
```

The code above sets a viewport in the frame buffer at offset 10,10 with width 80 and 20. All following reads and writes will be on this viewport. Assume it is a 1600 bytes fixed size file. The framebuffer is written/read in row-wise order, first k bytes overwrite the bytes at $(x, y) to (x + k - 1, y)$. When k overflows width, it continues from $(x, y + 1)$. The viewport can be set by `ioctl()` call.
The frame buffer (ie. whole screen) is also in row-wise order, represented as a continous buffer. If the framebuffer has $W \times H$ geometry and viewport has (x, y, w, h), the top left corner of the view port has the offset $yW + x$ and the lower right corner has the offset $(y + h - 1)W + (x + h - 1)$.
The default frame buffer size is 1000 by 1000. When a process opens a device for the first time it has the whole frame buffer as the viewport (i.e. x=0, y=0, width=1000, height=1000). The frame buffer size can be adjusted by the module parameters width and height. Each minor device can have a different size when changed via `ioctl()` (FB536_IOCTSETSIZE). Maximum value for height and width is 10000.

I/O operations on framebuffer device do not block. lseek() operation works as expected. Changes the current offset in number of bytes with respect to the top left corner of the window. Only blocking operation is the FB536_IOCWAIT ioctl() call for devices opened for reading. It waits until a task updates the viewport. It returns with a success after an update. It blocks for a possible future writer as well as current writers of the device. It only wakes up when a write intersects with the current viewport of interest.
read() operations directly gets the value of the pixel. On the other hand write() operations change the pixel value based on a operator that can be set through ioctl() operation FB536_IOCTSETOP for the current file state. The following operations are defined:

FB536_SET The default behaviour, setting the pixel value to the written character. It is the default.

FB536_ADD Adds written character value to pixel value. If value overflows, it is set to 255.

FB536_SUB Subtracts written character value to pixel value. If value underflows, it is set to 0.

FB536_AND Pixel value is set to bitwise 'and' (&) of written character value and pixel value.

FB536_OR Pixel value is set to bitwise 'or' (|) of written character value and pixel value.

FB536_XOR Pixel value is set to bitwise 'xor' (^) of written character value and pixel value.

All kernel structures should be protected against race conditions including the framebuffer data. You can keep framebuffer data in a single continous block or array of rows. The read, write operations, and some ioctl operations should work in a protected region.

2 Implementation

Download and change one of the scull devices under:
https://github.com/onursehitoglu/ldd4. scullc() is simplest but has enough features for you.
All module parameters you should support is given in the following example:
modprobe fb536 numminors=8 width=300 height=300
numminors is the number of minor devices created. Default value is 4. width is the default width of each frame buffer device, defaults to 1000. height is the default height of each frame buffer device, defaults to 1000.
Use debian trixie distribution as we had in the previous homework. Use qemu-debian-create-image.
Install linux-headers-$(uname -r) along with git, ssh-client, make, gcc after boot.
Your module should allocate a character device (dynamic major number) and register the following operations:
open(), read(), write(), release(), lseek(), unlocked_ioctl()
When reads and writes are shorter then the data size of a window, the offset of the file structure is set and next read/write continues from there. When reads and writes are larger than the data size, only remaining part of the window is processed and actual number of bytes read/written — which should be less than the requested — is returned. The next read/write will return 0 indicating an end of file.
Your device should support following ioctl functions:

```
#include<linux/ioctl.h>

struct fb_viewport {
    unsigned short x, y;
    unsigned short width, height;
};
```

```
#define FB536_SET    0
#define FB536_ADD    1
#define FB536_SUB    2
#define FB536_AND    3
#define FB536_OR     4
#define FB536_XOR    5

#define FB536_IOC_MAGIC  'F'

#define FB536_IOCRESET _IO(FB536_IOC_MAGIC, 0)
#define FB536_IOCTSETSIZE _IO(FB536_IOC_MAGIC, 1)
#define FB536_IOCQGETSIZE _IO(FB536_IOC_MAGIC, 2)
#define FB536_IOCSETVIEWPORT _IOW(FB536_IOC_MAGIC, 3, struct fb_viewport)
#define FB536_IOCGETVIEWPORT _IOR(FB536_IOC_MAGIC, 4, struct fb_viewport)
#define FB536_IOCTSETOP _IO(FB536_IOC_MAGIC, 5)
#define FB536_IOCQGETOP _IO(FB536_IOC_MAGIC, 6)
#define FB536_IOCWAIT _IO(FB536_IOC_MAGIC, 7)

#define FB536_IOC_MAXNR 7
```

FB536_IOCRESET
resets all framebuffer data setting all pixels to 0.

FB536_IOCTSETSIZE
sets the size of the frame buffer device (affecting all processes keeping the device open). This is a destructive operation which will set all pixels to 0, loosing previous frame buffer content. `ioctl(fd, FB536_IOCTSETSIZE, size)` will set the framebuffer width to `size >> 16` and framebuffer height to `size & 0xffff`. Higher 16 bits are the width, lower 16 bits are the height. You can use it as:
`ioctl(fd, FB536_IOCTSETSIZE, width << 16 | height)`
with correct (greater than 255, less than 10001) width and height values. Otherwise returns with -EINVAL. Note that some processes may have currently defined a larger viewport than the new framebuffer. In that case, they will get and end of file on their first read or write operation. You need to make a viewport check as the first operations of each I/O function.

FB536_IOCQGETSIZE
returns the current height and width information as the return value of the `ioctl()` call, as an integer, the same way in the FB536_IOCTSETSIZE operation.

FB536_IOCSETVIEWPORT
sets the current window size of the file structure. Called as:
`struct fb_viewport fba = { 10, 10, 80,20 };`
`ioctl(fd, FB536_IOCSETVIEWPORT, &fba));`
When the described rectangle in the values overflows the framebuffer boundary, it returns -EINVAL and does not update the current viewport.

FB536_IOCGETVIEWPORT
gets the current window size of the file structure in the provided buffer as:
`struct fb_viewport fba;`
`ioctl(fd, FB536_IOCGETVIEWPORT, &fba));`

FB536_IOCTSETOP
sets the current operation for the file structure. All following writes are subject to the new operation. It returns -EINVAL if file is opened read only.

FB536_IOCQGETOP
returns the current operation for the file structure as the return value of the ioctl() call. It returns -EINVAL if file is opened read only.

FB536_IOCWAIT
blocks until some other process updates the viewport in any way (it does not need to change the values). Even a single byte intersection will make it unblock. The framebuffer size changes also unblock the call since buffer is initialized to zero.

You need to wake up all threads with viewports of interest and only threads with viewports of interest. Non-intersecting wiewports should not be waken up. Create a synchronization value (ie. continuation) per blocking thread or file structure for waking up threads selectively.

An extraordinary case is another thread calling FB536_IOCSETVIEWPORT with the same `struct` file with the wait. Call will also unblocked for this case. You can keep the blocking synchronization value per `struct` file, then for all file structures of interest, wake all blocking threads.

Use any kernel library like list.h, mutex, continuation, waitqueue as you need.

3 Submission

Submit your code as a `tar.gz` file (no fancy formats like zip, rar and friends). It should contain necessary files as in scull driver (Get rid of extra implementations like pipe.c).