CEng 536 Advanced Unix
Fall 2016
Take Home Final/Kernel Project 2
Due: 24/01/2015

1 Description

In this project you are going to implement a pseudo character device driver for Linux kernel supporting a simple framebuffer device. A framebuffer 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 in the framebuffer area.

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

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

struct fb_area fba = { 10, 10, 80,20 }; /* 70x10 window*/
int fd = open("/dev/fb5360","O_RDWR");
if (ioctl(fd, FB536_IOCSETWINDOW, &fba))
        /* handle error */;

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

/* following set all window with 0 50 pattern */
for (int i = 0; i < 70; 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 < 70; i++) {
        write(fd, data, 10);
}
```

The default frame buffer size is 1000 by 1000. It 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. Read/write operations are carried out in row-wise order within the window. Each file structure will have a different window that can be set by the `FB536_IOCSETWINDOW`. The default window when the file is opened for the first time is set to the whole framebuffer of the minor device.

`lseek()` operation works as expected. Changes the current offset in number of bytes with respect to the top left corner of the window.

`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 structure. Following operations are defined:

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

`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 framebuffer data. You can keep framebuffer data in a single continous block or array of rows. In any way, `read` and `write` operations, and some `ioctl` operations should work in a protected region. Using a single mutex per minor framebuffer is accepted. If reader/writer locks in two dimensions as in previous project is implemented, a 20 points bonus will be granted. Explicitly state you have successfully implemented the bonus part in a `README` file in your submission.

2 Implementation

Download and change one of the scull devices under:
`https://github.com/duxing2007/ldd3-examples-3.x`. `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.
You can use your favorite Linux distribution as long as kernel is new (> 3.6). Install `linux-headers` package, compile and test your device.
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.
Your device should support following ioctl functions:

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

struct fb_area {
        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    _IOW(FB536_IOC_MAGIC, 0)
#define FB536_IOCTSETSIZE _IOW(FB536_IOC_MAGIC, 1)
#define FB536_IOCQGETSIZE _IOW(FB536_IOC_MAGIC, 2)
#define FB536_IOCSETWINDOW _IO(FB536_IOC_MAGIC, 3)
#define FB536_IOCGETWINDOW _IO(FB536_IOC_MAGIC, 4)
#define FB536_IOCTSETOP _IOW(FB536_IOC_MAGIC, 5)
#define FB536_IOCQGETOP _IOW(FB536_IOC_MAGIC, 6)

#define PQDEV_IOC_MAXNR 6
```

`FB536_IOCRESET` resets all framebuffer data setting all pixels to 0.

`FB536_IOCTSETSIZE` sets the size of the framebuffer device (affecting all processes keeping the device open). This is a destructive operation which will set all new pixels to 0, loosing previous framebuffer 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 (< 10000) `width` and `height` values. Note that some processes may access a larger window than the new framebuffer. In that case, they will be cropped to access the new area. You can check and crop the window, and set the current offset of the file structure in the first following I/O access to the device. If specified values are not in the valid range of 0 to 9999, call returns `-EINVAL` and framebuffer is not changed.

`FB536_IOCQGETSIZE` returns the current height and width information as the return value of the `ioctl()` call.

`FB536_IOCSETWINDOW` sets the current window size of the file structure. Called as:

```
struct fb_area fba = { 10, 10, 80,20 };
ioctl(fd, FB536_IOCSETWINDOW, &fba));
```

When the described rectangle in the values overflows the framebuffer boundary, it returns `-EINVAL` and does not update the current window.

`FB536_IOCGETWINDOW` gets the current window size of the file structure in the provided buffer as:

```
struct fb_area fba;
ioctl(fd, FB536_IOCGETWINDOW, &fba));
```

`FB536_IOCTSETOP` sets the current operation for the file structure. All following writes are subject to the new operation.

`FB536_IOCQGETOP` returns the current operation for the file structure as the return value of the `ioctl()` call.

Please specify in a `README` file in your submission if bonus is implemented.

3 Submission and External libraries

You need to submit a 'tar.gz' archive (no zip) containing scull like Makefile and your source. Makefile, a .c file, a .h file, load and unload scripts are sufficient.
Please ask all questions to:
`news://news.ceng.metu.edu.tr:2050/metu.ceng.course.536/`
`https://cow.ceng.metu.edu.tr/News/thread.php?group=metu.ceng.course.536`