CEng 536 Advanced Unix
Fall 2022
Homework 3
Due: Jan 21st, 2024

1 Description

In this project you will implement a pseudo character device driver for Linux kernel. The device will be a simple shared queue with primitive encryption. Name you device driver as cipher.
Each minor device should behave as a separate cipher queue. By default, read operations are non-blocking. Blocking support will be bonus. Writes are not blocked. The device will have theoretically infinite capacity. There is no module paramters. Number of minor devices is fixed to 8 and major number is assigned by the system. Writes are limited by 8192 bytes.
You need a two `ioctl()` commands for message buffers defined in the `cipher.h` as follows:

```
....
#define CIPHER_NR_DEVS  8

/* Use 222 as magic number */
#define CIPHER_IOC_MAGIC  222
#define CIPHER_IOCCLR    _IO(CIPHER_IOC_MAGIC,0)
#define CIPHER_IOCSKEY _IOW(CIPHER_IOC_MAGIC,1,char)
#define CIPHER_IOCQREM _IOR(CIPHER_IOC_MAGIC,2)
#define CIPHER_IOC_MAXNR      2

#endif /* _CIPHER_H_ */
```

This calls are explained as:

`CIPHER_IOCCLR`
: This message clears device into the initial state. There is no message. Cipher is default. The state of all existing readers and writers will resume from an empty buffer.

`CIPHER_IOCSKEY`
: `ioctl(fd, CIPHER\_IOCSKEY, key)` call will set zero terminated string, key as the current cipher. Changing cipher will have a similar effect as CIPHER_IOCCLR, the device will be reset to inital state.

`CIPHER_IOCQREM`
: `ioctl(fd, MQBUF_IOCQREM)` call will return the remaining number of bytes to read. In case of write, it returns remaining number of bytes to overwrite in the buffer. If current file offset is at the EOF, 0 will be returned.

2 Implementation

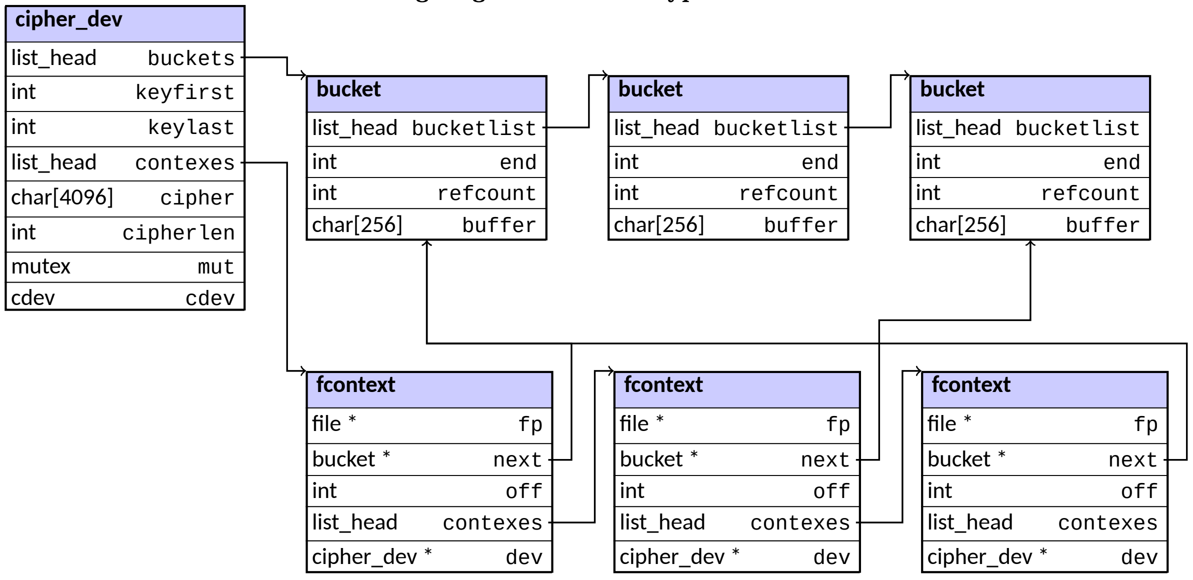
In the template implementation, the following structure definitions are made:

```
struct bucket {
    struct list_head bucketlist;
    int end;
    int refcount;
    struct mutex mut;    /* use if you like a finer critical region */
    char buffer[256];
};

struct cipher_dev {
    struct list_head buckets;
    int keyfirst;     /* key offset of first bucket */
    int keylast;     /* key offset of EOF bucket */
    struct list_head contexes; /* all fcontext structures refering this */
    char cipher [4096];     /* max cipher is allocated */
    int cipherlen;          /*  length of cipher */
    struct mutex mut;
    struct cdev cdev;     /* Char device structure. */
};

struct fcontext {    /* used for context of each opener struct file */
    struct file *fp;
    struct bucket *next;    /* current bucket to read or write */
    int off;    /* our offset in the current bucket */
    int keyoff;   /* offset in the key */
    int mode; /* file mode */
    struct list_head contexes;
    struct cipher_dev *dev;
};
```

For each minor device, the following diagram denotes a typical state of the buffer:



In this design, data of each minor device is kept on a list of buckets. Each bucket is 256 bytes long. As the devices is written, new buckets are added at the tail and as all devices complete reading a bucket, it is deallocated from the beginning.
Reads and writes are exclusive in the device, meaning it can be either opened as read only or write only. All operations are non blocking. Only O_APPEND mode is supported causing a write open will set its offset to end of buffer. seek operation is not supported, the offset parameters in the kernel functions are ignored. Each open will create a fcontext structure keeping information about current state of reading/writing the device. The current bucket pointer next and offset off gives the next position to read and write on the bucket. Each bucket keeps a reference count and as all data is complete, reference count is decremented by the completion

of read/write on bucket. Readers deallocate a bucket if reference count gets 0. Writers decrement it to make it zero but do not deallocate them for following readers to read. When a device is first opened, reference count of all buckets are incremented. This way a bucket is kept in memory as long as there is a potential reader. fcontext is deallocated when the file is closed. The device keeps a list of contexes in order to implement CIPHER_IOCCLR which needs to update all contexes with clearead buckets.
As the name of the device implies, this is an encrypting device. The encryption is simple, the bytes to be written are XOR'ed with the key and stored in bucket. When it is to be read, it is XOR'ed back and sent to user space. The key is cycled as data is written. The fcontext keeps a key offset in order to keep next key location to use.
Implement this device driver with open, read, write, close, and ioctl , functions in Linux. You can use the `scull` sample device driver code given in the github repository: `https://github.com/onursehitoglu/ldd4`
Also you need to use the guidelines in the online book:
`https://lwn.net/Kernel/LDD3/`

2.1 Blocking Operation Bonus (20 pts)

You can implement blocking operation as default operation and implement O_NONBLOCK as the nonblocking operation. In blocking operation, reads will block if the queue is empty. The first write() will release the block. Use waitqueue.h to implement the blocking. Writes do not block.

2.2 Poll Operation Bonus (10 pts)

This bonus only makes sense when blocking operation is implemented. Implement poll() operation as well so that your device can take part in select() and poll() system calls. scull `pipe.c` source code has examples of coding this.

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).