CS 444 GROUP 15

Writing assignment 2

Writer:

Changxu YAN

931-927-069

November 21, 2016

I, Changxu Yan, hereby state this is my own work, with no help given or received.

To COMPARE THESE TWO APIs, we need to understand what they are. For POSIX, there are a bunch of standards for "UnixLike" APIs, all called POSIX followed by some obscure serial number. Windows API is Microsoft's core set of application programming interfaces (APIs) available in the Microsoft Windows operating systems. POSIX support more "UnixLike" Operating systems, such as linux, ubontu and MacOS, it has better portablity compare to WIN32 API. These two interfaces has lots of differences and similarities. In here, I will talk about File I/O, signal handling and processes.[1]

FILE I/O

Both POSIX and Windows API are using the universal I/O model. All system calls for performing I/O refer to open files using a file descriptor, a (usually small) nonnegative integer. You can find file descriptor for all types of files: pipe, FIFOs, sockets, terminals, devices, and regular files.

In POSIX API, we open and create a file descriptor using

```
file_descriptor = open(path,flags,mode);
```

returning a -1 means an error occurred and errno is set appropriately. The most common flags I use are O_RDONLY, O_WRONLY, O_RDWR, and O_CREAT, O_APPEND. Using O_CREAT flags will give you the option that if there is not a file exist in that name, it will create new one with the mode permission, such as S_IRWXU for user has read write and execute permission. Using O_APPEND is equivalent to use

```
lseek(int fd, off_t offset, int SEEK_END)
```

to position offset at the end of the file.

```
num_read = read(fd, buffer, count)
```

This function reads from the open file referred to by fd then put the content in the buffer. it returns the number of how many 8 bits it reads.

```
num_written = write(fd, buffer, count)
```

is kind of similar to read, but writes from buffer to the open file referred to by fd. It returns the number of how many 8bits it written. The way to close a file is fairly straight forward, just

```
close(file_descriptor);
```

Below is the file copy tool in C using POSIX API.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <errno.h>
#include <fcntl.h>
#include <string.h>
#define FILEPATH1 "file.txt"
#define FILEPATH2 "file_copy.txt"
#define BUF_SIZE 1024
int main(int argc, char *argv[]){
    int fdread, fdwrite;
    char buf[BUF_SIZE];
    ssize_t num_read;

    if ((fdread = open(FILEPATH1, O_RDONLY)) == -1){
        printf("Failed to open file %s", FILEPATH1);
        exit(EXIT_FAILURE);
    }

    if ((num_read = read(fdread, buf, BUF_SIZE)) == -1){
        perror("Read: ");
        exit(EXIT_FAILURE);
    }
    buf[num_read] = '\0';

    if ((fdwrite = open(FILEPATH2, O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR)) == -1){
        printf("Failed to open file %s", FILEPATH2);
        exit(EXIT_FAILURE);
    }
```

```
    if ((write(fdwrite, buf, strlen(buf))) == -1){
        perror("Write: ");
        exit(EXIT_FAILURE);
    }

    return 0;
}
```

In WIN API, the way to copy a file has lots of simalarity as POSIX API.

```
#include <windows.h>
#include <tchar.h>
#include <stdio.h>
#include <strsafe.h>

#define BUFFERSIZE 81

void __cdecl _tmain(int argc, TCHAR *argv[])
{
    HANDLE rFile, wFile;
    DWORD dwBytesRead = 0;
DWORD dwBytesWritten = 0;
    char ReadBuffer[BUFFERSIZE] = {0};
DWORD dwBytesToWrite = (DWORD)strlen(ReadBuffer);

hFile = CreateFile(argv[1],              // file to open
                       GENERIC_READ,          // open for reading
                       FILE_SHARE_READ,       // share for reading
                       NULL,                  // default security
                       OPEN_EXISTING,         // existing file only
                       FILE_ATTRIBUTE_NORMAL, // normal file
                       NULL);                 // no attr. template
```

```
if (rFile == INVALID_HANDLE_VALUE)
{
    DisplayError(TEXT("CreateFile"));
    _tprintf(TEXT("Terminal failure: unable to open file \"%s\" for read.\n"), argv[1]);
    return;
}

// Read one character less than the buffer size to save room for
// the terminating NULL character.

if( FALSE == ReadFile(rFile, ReadBuffer, BUFFERSIZE-1, &dwBytesRead, NULL) )
{
    DisplayError(TEXT("ReadFile"));
    printf("Terminal failure: Unable to read from file.\n");
    CloseHandle(hFile);
    return;
}

// write to another file
wFile = CreateFile(argv[1],                // name of the write
                   GENERIC_WRITE,          // open for writing
                   0,                      // do not share
                   NULL,                   // default security
                   CREATE_NEW,             // create new file only
                   FILE_ATTRIBUTE_NORMAL,  // normal file
                   NULL);                  // no attr. template

   if (wFile == INVALID_HANDLE_VALUE)
   {
       DisplayError(TEXT("CreateFile"));
       _tprintf(TEXT("Terminal failure: Unable to open file \"%s\" for write.\n"), argv[1
       return;
   }
```

```
      _tprintf(TEXT("Writing %d bytes to %s.\n"), dwBytesToWrite, argv[1]);

      bErrorFlag = WriteFile(
                      wFile,           // open file handle
                      ReadBuffer,      // start of data to write
                      dwBytesToWrite,  // number of bytes to write
                      &dwBytesWritten, // number of bytes that were written
                      NULL);            // no overlapped structure

      if (FALSE == bErrorFlag)
      {
          DisplayError(TEXT("WriteFile"));
          printf("Terminal failure: Unable to write to file.\n");
      }
   CloseHandle(rFile);
   CloseHandle(wFile);
}
```

In WIN API, the way to create a file is very different. A struct like HANDLE must be used to create a file.

```
HANDLE WINAPI CreateFile(
  _In_      LPCTSTR lpFileName,
  _In_      DWORD dwDesiredAccess,
  _In_      DWORD dwShareMode,
  _In_opt_  LPSECURITY_ATTRIBUTES lpSecurityAttributes,
  _In_      DWORD dwCreationDisposition,
  _In_      DWORD dwFlagsAndAttributes,
  _In_opt_  HANDLE hTemplateFile
);
```

lpFileName is a characters array to store the path and name of the file which is the same as POSIX API.

”dwShareMode [in] The requested sharing mode of the file or device, which can be read, write, both,

delete, all of these, or none (refer to the following table). Access requests to attributes or extended attributes are not affected by this flag." This sharing mode setting is not the mode in the POSIX API mode setting. To set different mode, you will using the 4 bytes address like 0x00000000 and 0x00000001 to achieve that.

"If the function succeeds, the return value is an open handle to the specified file, device, named pipe, or mail slot. If the function fails, the return value is INVALID_HANDLE_VALUE. To get extended error information, call GetLastError."

SIGNAL HANDLING

A signal is a software interrupt delivered to a process. In POSIX API, the way we fulfill signal handling is by sigaction(int sig, struct sigaction *act, struct sigaction *oact). before we can actually catch a signal, we need including ¡signal.h¿ and create extern void psignal(int sig, const char *msg). The most familier signal to me are the signal using in projects.

SIGHUP report that user's terminal is disconnected. SIGINT: program interrupt. (ctrl-c).SIGQUIT terminate process and generate core dump.

```
#include <signal.h>
static void handler(int signum){
    /* Take appropriate actions for signal delivery */
    psignal(signal, "Caught it");
}
int main(){
    struct sigaction sa;

    sa.sa_handler = handler;
    sigemptyset(&sa.sa_mask);
    /* Restart functions if interrupted by handler */
    sa.sa_flags = SA_RESTART;

    if (sigaction(SIGINT, &sa, NULL) == -1)
    /* Handle error */;
    /* Further code */;
```

```
}
```

In Win API, the signal handling is pretty simlar to POSIX. You have a int signal value, and a funtion to be excuted.

```
void (__cdecl *signal(
   int sig,
   void ((__cdecl *func ) (int [, int ] )))
   (int));
Return Value:
```

Win API signal returns the previous value of func that's associated with the given signal. According the example in Microsoft website, if the previous value of func was SIG_IGN, the return value is also SIG_IGN. A return value of SIG_ERR indicates an error; in that case, errno is set to EINVAL.

PROCESSES

Process is an instance of a computer program that is being executed. It contains the program code and its current activities. Each process has a unique id call pid, when you want to do something with the particular process, you can always refer to the pid. In POSIX API, we used forking to create a new process. fork() returns interesting things, it will return 0 for child case, -1 for error as usual, child process id to parent process. Thus, we can use a switch to forking different processes. Child has a unique process ID and its own copy of the parent's descriptors. When we using forking, we have to make sure the parent process get killed after child process, otherwise, the child process will be in the middle of nowhere then become a zombie process.

```
pid_t child;
int status;

switch ((child = fork())){
    case 0:
    //this is the child case
```

```
        printf("I am a child, my process ID is %d (%d)\n", getpid(), getppid());
        execlp("ps", "-t", "-l", (char*)NULL);
        break;
    case -1:
    //this is the error case
        perror("Could not create child");
        exit(-1);
    default:
    //this is the parent case
        printf("I am the parent, my process ID is %d\n", getpid());
        wait(&status);
        printf("Waited on child %d\n", child);
    break;
}
```

Creating child processes is a foreign concept to Windows API. However, there are several Windows functions that emulate the forking process in POSIX API. This is not meaning that Windows API does not support multiple processes; it just works in a different way. The CreateProcess function in Windows is the equivalent of using fork(), execl() and system() respectively in UNIX. There are Windows equivalent for exit(), getpid(), wait() and kill() in UNIX.

The way creating child processes in Windows is different from POSIX API. Windows treats things more as objects rather than files and is evident in some of the Windows functions for process management. The process handlers for Windows are WaitForMultipleObjects and WaitForSingleObject, which is like calling waitpid() or wait() in UNIX, but notice the names of these functions. They are focused around objects and not ”children” like in UNIX. The biggest difference between multiple processes in Windows and UNIX has to be the concept of the parent and child. Because of this, there are no Windows equivalents for the exec() functions and its variants.

In WINAPI, when you creating a new process, the new process runs independently from the creating process, usually refer to parent- child relationship which is same as POSIX processing.

```
#include <windows.h>
#include <stdio.h>
#include <tchar.h>
void _tmain( int argc, TCHAR *argv[] ){
    STARTUPINFO si;
    PROCESS_INFORMATION pi;

    ZeroMemory( &si, sizeof(si) );
    si.cb = sizeof(si);
    ZeroMemory( &pi, sizeof(pi) );

    if( argc != 2 ){
        printf("Usage: %s [cmdline]\n", argv[0]);
        return;
    }

    // Start the child process.
    if( !CreateProcess( NULL,   // No module name (use command line)
        argv[1],        // Command line
        NULL,           // Process handle not inheritable
        NULL,           // Thread handle not inheritable
        FALSE,          // Set handle inheritance to FALSE
        0,              // No creation flags
        NULL,           // Use parent's environment block
        NULL,           // Use parent's starting directory
        &si,            // Pointer to STARTUPINFO structure
        &pi )           // Pointer to PROCESS_INFORMATION structure
    ) {
        printf( "CreateProcess failed (%d).\n", GetLastError() );
        return;
    }

    // Wait until child process exits.
    WaitForSingleObject( pi.hProcess, INFINITE );
```

```
    // Close process and thread handles.
    CloseHandle( pi.hProcess );
    CloseHandle( pi.hThread );
}
```

If CreateProcess succeeds, it returns a PROCESS_INFORMATION structure containing handles and identifiers for the new process and its primary thread. You also need a wait function to wait on child process to stop before parent process. The thread and process handles are created with full access rights, although access can be restricted if you specify security descriptors. When you no longer need these handles, close them by using the CloseHandle function.

Reference:

http://msdn.microsoft.com/en-us/library/xdkz3x12.aspx

http://msdn.microsoft.com/en-us/library/windows/desktop/ms682512(v=vs.85).aspx

http://scott.sherrillmix.com/blog/category/programmer/latex/

http://msdn.microsoft.com/en-us/library/windows/desktop/bb540534(v=vs.85).aspx

REFERENCES

[1] R. Love, *Linux Kernel Development*, 3rd ed. Pearson Education, Inc., 2010.

```
/*
*
* Modified version of the sbull.c driver from
* http://hi.baidu.com/casualfish/item/7931bbb58925fb951846977d.
* by Chauncey Yan, Wangxiaomei, Jacky.
* Ported to kernel v2.6.31 by casualfish. 2010.7.20
*
* Crypto testing/Hexdump reference code: http://www.logix.cz/michal/devel/cryptodev/cryptoap
*/

#include <linux/module.h>
#include <linux/moduleparam.h>
#include <linux/init.h>

#include <linux/sched.h>
#include <linux/kernel.h>        /* printk() */
#include <linux/slab.h>                /* kmalloc() */
#include <linux/fs.h>               /* everything... */
#include <linux/errno.h>        /* error codes */
#include <linux/timer.h>
#include <linux/types.h>        /* size_t */
#include <linux/fcntl.h>        /* O_ACCMODE */
#include <linux/hdreg.h>        /* HDIO_GETGEO */
#include <linux/kdev_t.h>
#include <linux/vmalloc.h>
#include <linux/genhd.h>
#include <linux/blkdev.h>
#include <linux/buffer_head.h>        /* invalidate_bdev */
#include <linux/bio.h>
#include <linux/crypto.h>        /* For crypto (Duh) */

MODULE_LICENSE("Dual BSD/GPL");

/*
 * Global variables and stuff
```

```
 */
static char *key = "someweakkey"; //Crypto key
struct crypto_cipher *tfm; //Crypto cipher struct. Critical for crypto.

module_param(key, charp, 0000); //Allow user to set key as param in module
static int osurd_major = 0;
/* Passing 0 for the major number means the kernel will allocate a new number for it.
*/
module_param(osurd_major, int, 0);
static int hardsect_size = 512;
module_param(hardsect_size, int, 0);
static int nsectors = 1024;        //Size of the drive
module_param(nsectors, int, 0);
static int ndevices = 4; //Number of RAM disks we want
module_param(ndevices, int, 0);

/*
* The different "request modes" we can use.
*/
enum {
        RM_SIMPLE = 0,                /* The extra-simple request function */
        RM_FULL = 1,                /* The full-blown version */
        RM_NOQUEUE = 2,                /* Use make_request */
};
static int request_mode = RM_SIMPLE;
module_param(request_mode, int, 0);

/*
* Minor number and partition management.
*/
#define OSURD_MINORS 16
#define MINOR_SHIFT 4
#define DEVNUM(kdevnum) (MINOR(kdev_t_to_nr(kdevnum)) >> MINOR_SHIFT
#define OSU_CIPHER "aes" //Cipher algorithm to use
```

```
/*
* We can tweak our hardware sector size, but the kernel talks to us
* in terms of small sectors, always.
*/
#define KERNEL_SECTOR_SIZE 512

/*
* After this much idle time, the driver will simulate a media change.
*/
#define INVALIDATE_DELAY 30*HZ

/*
* The internal representation of our device.
*/
struct osurd_dev {
        int size;               /* Device size in sectors */
        u8 *data;               /* The data array */
        short users;               /* How many users */
        short media_change;        /* Flag a media change? */
        spinlock_t lock;        /* For mutual exclusion */
        struct request_queue *queue;        /* The device request queue */
        struct gendisk *gd;        /* The gendisk structure */
        struct timer_list timer;        /* For simulated media changes */
};
static struct osurd_dev *Devices = NULL;

/*
 * For debugging crypto.
 */
static void hexdump(unsigned char *buf, unsigned int len)
{
        while (len--)
                printk("%02x", *buf++);

        printk("\n");
```

```
}

/*
* Handle an I/O request, in sectors.
*/
static void
osurd_transfer(struct osurd_dev *dev, unsigned long sector,
               unsigned long nsect, char *buffer, int write)
{
        unsigned long offset = sector *KERNEL_SECTOR_SIZE;
        unsigned long nbytes = nsect *KERNEL_SECTOR_SIZE;
        int i;
        if ((offset + nbytes) > dev->size) {
                printk(KERN_NOTICE "Beyond-end write (%ld %ld)\n", offset,
                       nbytes);
                return;
        }

        crypto_cipher_clear_flags(tfm, ~0);
        crypto_cipher_setkey(tfm, key, strlen(key)); //Set the key to be our crypto key

        if (write){
                printk("Writing to RAMdisk\n");
                printk("Pre-encrypted data: ");
                hexdump(buffer, nbytes); //For debugging
                for (i = 0; i < nbytes; i += crypto_cipher_blocksize(tfm)) {
                        memset(dev->data + offset + i, 0,
                               crypto_cipher_blocksize(tfm));
                        crypto_cipher_encrypt_one(tfm, dev->data + offset + i,
                                                  buffer + i);
                }
                printk("Encrypted data:");
                hexdump(dev->data +offset, nbytes); //For debugging
        } else {
                printk("Reading from RAMdisk\n");
```

```
                printk("Encrypted data:");
                hexdump(dev->data +offset, nbytes); //For debugging
                for (i = 0; i < nbytes; i += crypto_cipher_blocksize(tfm)) {
                        crypto_cipher_decrypt_one(tfm, buffer + i,
                                                  dev->data + offset + i);
                }
                printk("Decrypted data: ");
                hexdump(buffer, nbytes); //For debugging
        }
}

/*
* The simple form of the request function.
* Same.
*/
static void
osurd_request(struct request_queue *q)
{
        struct request *req;

        req = blk_fetch_request(q);
        while (req != NULL) {
                struct osurd_dev *dev = req->rq_disk->private_data;
                if (req->cmd_type != REQ_TYPE_FS) {
                        printk(KERN_NOTICE "Skip non-fs request\n");
                        __blk_end_request_all(req, -EIO);
                        continue;
                }
                osurd_transfer(dev, blk_rq_pos(req),
                               blk_rq_cur_sectors(req), req->buffer,
                               rq_data_dir(req));
                /* end_request(req, 1); */
                if (!__blk_end_request_cur(req, 0)) {
                        req = blk_fetch_request(q);
                }
```

```
        }
}

/*
* Transfer a single BIO.
* Also same.
*/
static int
osurd_xfer_bio(struct osurd_dev *dev, struct bio *bio)
{
        int i;
        struct bio_vec *bvec;
        sector_t sector = bio->bi_sector;

        /* Do each segment independently. */
        bio_for_each_segment(bvec, bio, i) {
                char *buffer = __bio_kmap_atomic(bio, i, KM_USER0);
                osurd_transfer(dev, sector, bio_cur_bytes(bio) >> 9        /* in sectors */
                               , buffer, bio_data_dir(bio) == WRITE);
                sector += bio_cur_bytes(bio) >> 9;        /* in sectors */
                __bio_kunmap_atomic(bio, KM_USER0);
        }
        return 0;               /* Always "succeed" */
}

/*
 * Transfer a full request.
 * Same
 */
static int
osurd_xfer_request(struct osurd_dev *dev, struct request *req)
{
        struct bio *bio;
        int nsect = 0;
```

```
        __rq_for_each_bio(bio, req) {
                osurd_xfer_bio(dev, bio);
                nsect += bio->bi_size / KERNEL_SECTOR_SIZE;
        }
        return nsect;
}

/*
* Smarter request function that "handles clustering".
* Same.
*/
static void
osurd_full_request(struct request_queue *q)
{
        struct request *req;
        int sectors_xferred;
        struct osurd_dev *dev = q->queuedata;

        req = blk_fetch_request(q);
        while (req != NULL) {
                if (req->cmd_type != REQ_TYPE_FS) {
                        printk(KERN_NOTICE "Skip non-fs request\n");
                        __blk_end_request_all(req, -EIO);
                        continue;
                }
                sectors_xferred = osurd_xfer_request(dev, req);
                if (!__blk_end_request_cur(req, 0)) {
                        blk_fetch_request(q);
                }
        }
}

/*
* The direct make request version.
* Same.
```

```
*/
static int
osurd_make_request(struct request_queue *q, struct bio *bio)
{
        struct osurd_dev *dev = q->queuedata;
        int status;

        status = osurd_xfer_bio(dev, bio);
        bio_endio(bio, status);
        return 0;
}

/*
* Open and close.
* Simulates removable media.
* Same.
*/
static int
osurd_open(struct block_device *device, fmode_t mode)
{
        struct osurd_dev *dev = device->bd_disk->private_data;

        del_timer_sync(&dev->timer);
        /* filp->private_data = dev; */
        spin_lock(&dev->lock);
        if (!dev->users)
                check_disk_change(device);
        dev->users++;
        spin_unlock(&dev->lock);
        return 0;
}

/*
 * Same
 */
```

```
static int
osurd_release(struct gendisk *disk, fmode_t mode)
{
        struct osurd_dev *dev = disk->private_data;

        spin_lock(&dev->lock);
        dev->users--;

        if (!dev->users) {
                dev->timer.expires = jiffies + INVALIDATE_DELAY;
                add_timer(&dev->timer);
        }
        spin_unlock(&dev->lock);

        return 0;
}

/*
* Look for a (simulated) media change.
* Same.
*/
int
osurd_media_changed(struct gendisk *gd)
{
        struct osurd_dev *dev = gd->private_data;

        return dev->media_change;
}

/*
* Revalidate.  WE DO NOT TAKE THE LOCK HERE, for fear of deadlocking
* with open.  That needs to be reevaluated.
* This function is called after a media change.
* Same.
*/
```

```
int
osurd_revalidate(struct gendisk *gd)
{
        struct osurd_dev *dev = gd->private_data;

        if (dev->media_change) {
                dev->media_change = 0;
                memset(dev->data, 0, dev->size);
        }
        return 0;
}

/*
* The "invalidate" function runs out of the device timer; it sets
* a flag to simulate the removal of the media.
* Same
*/
void
osurd_invalidate(unsigned long ldev)
{
        struct osurd_dev *dev = (struct osurd_dev *) ldev;

        spin_lock(&dev->lock);
        if (dev->users || !dev->data)
                printk(KERN_WARNING "osurd: timer sanity check failed\n");
        else
                dev->media_change = 1;
        spin_unlock(&dev->lock);
}

/*
 * Added getgeo() function
*/
static int
osurd_getgeo(struct block_device *device, struct hd_geometry *geo)
```

```
{
        struct osurd_dev *dev = device->bd_disk->private_data;
        long size = dev->size * (hardsect_size / KERNEL_SECTOR_SIZE);
        geo->cylinders = (size & ~0x3f) >> 6;
        geo->heads = 4;
        geo->sectors = 16;
        geo->start = 0;                 //Changed this to 0 from 4
        return 0;
}

/*
* The device operations structure.
*/
static struct block_device_operations osurd_ops = {
        .owner = THIS_MODULE,
        .open = osurd_open,
        .release = osurd_release,
        .media_changed = osurd_media_changed,
        .revalidate_disk = osurd_revalidate,
        .getgeo = osurd_getgeo
};

/*
* Set up our internal device.
* Called by the init function.
* Same.
*/
static void
setup_device(struct osurd_dev *dev, int which)
{
        /*
        * Get some memory.
        */
        memset(dev, 0, sizeof (struct osurd_dev));
        dev->size = nsectors * hardsect_size; //Set the device size
```

```
dev->data = vmalloc(dev->size); //Allocate virtually contiguous memory for the RAM d
if (dev->data == NULL) {
        printk(KERN_NOTICE "vmalloc failure.\n");
        return;
}
/* Allocate a spinlock for mutual exclusion */
spin_lock_init(&dev->lock);

/*
 * The timer which "invalidates" the device.
 * This is a 30-second timer used to simulate
 * behavior of a removable device.
 */
init_timer(&dev->timer);
dev->timer.data = (unsigned long) dev;
dev->timer.function = osurd_invalidate;

/*
 * The I/O queue, depending on whether we are using our own
 * make_request function or not.
 */
switch (request_mode) {
case RM_NOQUEUE:
        dev->queue = blk_alloc_queue(GFP_KERNEL);
        if (dev->queue == NULL)
                goto out_vfree;
        blk_queue_make_request(dev->queue, osurd_make_request);
        break;
case RM_FULL:
        /* blk_init_queue() allocates the request queue. */
        dev->queue = blk_init_queue(osurd_full_request, &dev->lock);
        if (dev->queue == NULL)
                goto out_vfree;
        break;
default:
```

```
        printk(KERN_NOTICE
               "Bad request mode %d, using simple\n", request_mode);
        /* fall into.. */
case RM_SIMPLE:
        dev->queue = blk_init_queue(osurd_request, &dev->lock);
        if (dev->queue == NULL)
                goto out_vfree;
        break;
}
blk_queue_logical_block_size(dev->queue, hardsect_size);
dev->queue->queuedata = dev;

/*
 * And the gendisk structure.
 * gendisk is the kernel's representation of an individual disk device.
 */
dev->gd = alloc_disk(OSURD_MINORS);
if (!dev->gd) {
        printk(KERN_NOTICE "alloc_disk failure\n");
        goto out_vfree;
}
dev->gd->major = osurd_major;
dev->gd->first_minor = which * OSURD_MINORS;
dev->gd->fops = &osurd_ops;
dev->gd->queue = dev->queue;
dev->gd->private_data = dev;
/*
 * Set the device names to osurda, osurdb, etc.
 */
snprintf(dev->gd->disk_name, 32, "osurd%c", which + 'a');
set_capacity(dev->gd, nsectors * (hardsect_size / KERNEL_SECTOR_SIZE));
add_disk(dev->gd);
return;

out_vfree:
```

```
        if (dev->data)
                vfree(dev->data);
}

/*
 * Module initialization function.
 */
static int __init
osurd_init(void)
{
        int i;
        /* Initialize the cipher with AES encryption */
        tfm = crypto_alloc_cipher(OSU_CIPHER, 0, 0);
        /* Error checking for crypto */
        if (IS_ERR(tfm)) {
                printk(KERN_ERR "osurd: cipher allocation failed");
                return PTR_ERR(tfm);
        }

        /*
        * Get registered.
        * Register a block device called "osurd."
        */
        osurd_major = register_blkdev(osurd_major, "osurd");
        if (osurd_major <= 0) {
                printk(KERN_WARNING "osurd: unable to get major number\n");
                return -EBUSY;
        }

        /*
        * Allocate the device array, and initialize each one.
        */
        Devices = kmalloc(ndevices * sizeof (struct osurd_dev), GFP_KERNEL);
        if (Devices == NULL)
                goto out_unregister;
```

```
        /* Setup each device */
        for (i = 0; i < ndevices; i++)
                setup_device(Devices + i, i);

        return 0;
      out_unregister:
                unregister_blkdev(osurd_major, "osurd");
                return -ENOMEM;
}

static void
osurd_exit(void)
{
        int i;
        for (i = 0; i < ndevices; i++) {
                struct osurd_dev *dev = Devices + i;

                del_timer_sync(&dev->timer);
                if (dev->gd) {
                        del_gendisk(dev->gd);
                        put_disk(dev->gd);
                }
                if (dev->queue)
                        /* Destroy the request queue by releasing the request_queue_t */
                        blk_cleanup_queue(dev->queue);
                if (dev->data)
                        /* Free the virtual memory allocated earlier for the disk */
                        vfree(dev->data);
        }
        /* Unregister the osurd block device */
        unregister_blkdev(osurd_major, "osurd");
        /* Free the crypto cipher struct from memory */
        crypto_free_cipher(tfm);
        kfree(Devices);
}
```

```
module_init(osurd_init);
module_exit(osurd_exit);
```