CPE 453: Operating Systems
Assignment 4: Minix Secret Driver
Daniel Gutierrez (jguti151)

Executive Summary: In this assignment, I implemented a driver to interface with a Secret device. The Secret device can only hold one secret at a time.

Architecture
The driver implemented is intended to interact with the Secret device, dev/Secret. The Secret device can only hold one secret at a time. The Secret device is a character device.

Here is some nomenclature is used to help me better understand:
Empty - There is nothing in the buffer.
Full - The buffer is not empty.
Free - The secret is not owned by any user
Owned - The secret is owned by at least 1 user

The device's designed behavior is as follows:
Opening the Device:
If Free:

- Any process may open /dev/Secret for reading or writing (but not both).
    - Opening for read-write access will result in a permission denied error (EACCES).
- The owner of that process that opened it will then become the owner of the secret.
- Open for writing can only succeed if the secret is free.
    - This means it may only be opened for writing once.
- The secret held is of fixed size.
    - The default size is 8 KB, but can be reconfigured.
    - Attempts to write more into the device than will fit will result in an ENOSPC response.

If Owned:

- /dev/Secret may not be opened for writing once it is holding a secret.
    - Attempting to write to a full device results in an ENOSPC error.
- /dev/Secret may be opened for reading by a process owned by the owner of the secret.
    - Attempts to open a full secret for writing result in a device full error (ENOSPC).
    - Attempts to read a secret belonging to another user result in a permission denied error (EACCES).
- The secret persists as long as there are open file descriptors associated with it.

Closing the Device:

- The number of open file descriptors is decremented.
    - Once the last file descriptor is closed after a read, the device becomes empty and free.
    - The device is then available for new writes.

IOCTL Behavior:

- Changing ownership
    - The desired user to grant ownership to is determined and the new owner of the secret is set to the desired user.

Live Update behavior:

- The secret owner, the current buffer data, the writing and reading positions within the buffer, and the open counter are saved.
- These variables are then restored to the driver.

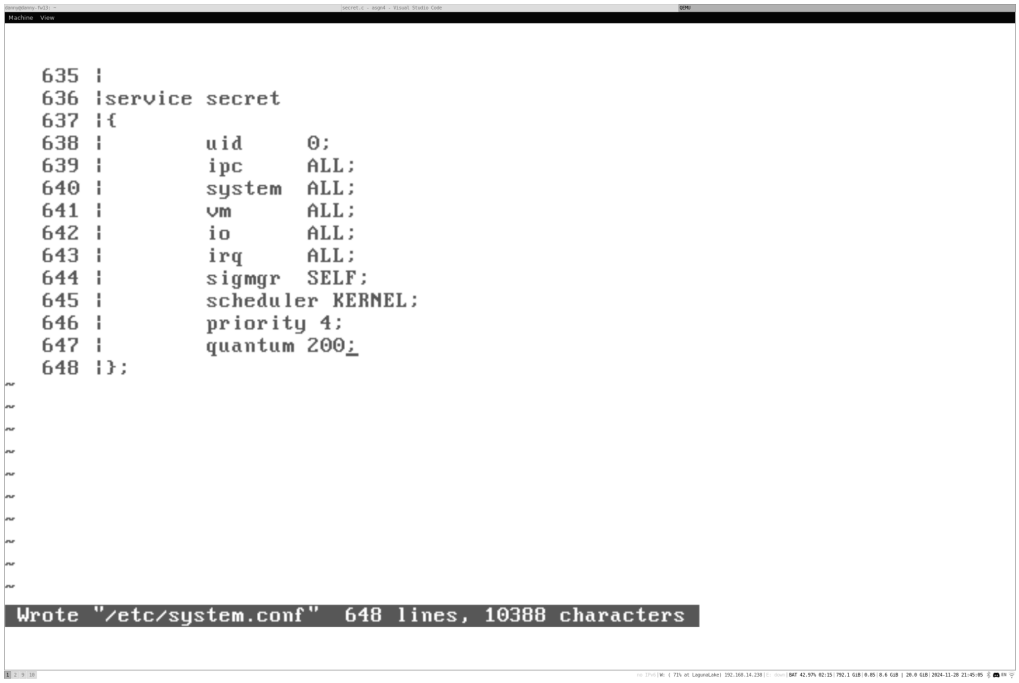
Driver Description

Dev Environment

I developed on MINIX 3.1.8 running on QEMU on a machine running Ubuntu 20.04.

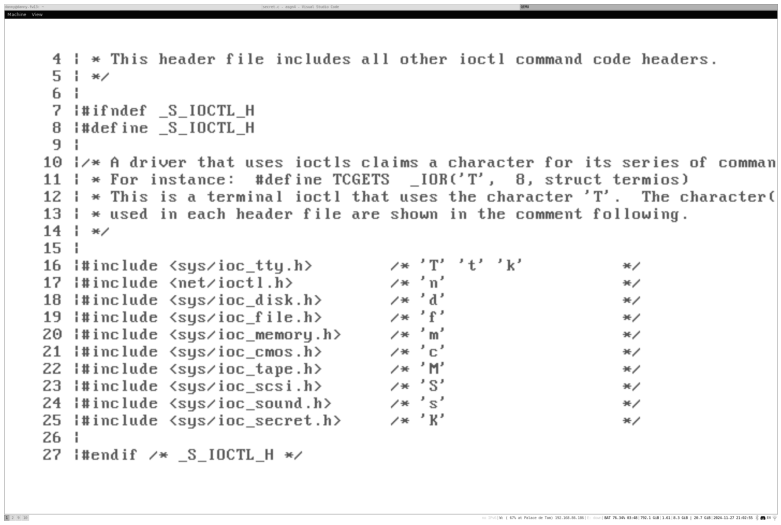
Files Modified

/etc/system.conf



I added the above configuration for the system to recognize the service I was adding. I decided to make the service all powerful to make it as simple as possible. In hindsight, this is a very poor decision, but for the sake of the lab it works for what I need it to do even if it is a security risk. I chose a priority of 4 to designate it as a user-level process. The quantum was set to 200ms, based on information from *Individual Programming Assignment User Mode Scheduling in MINIX 3*[1] indicating this as the default value.

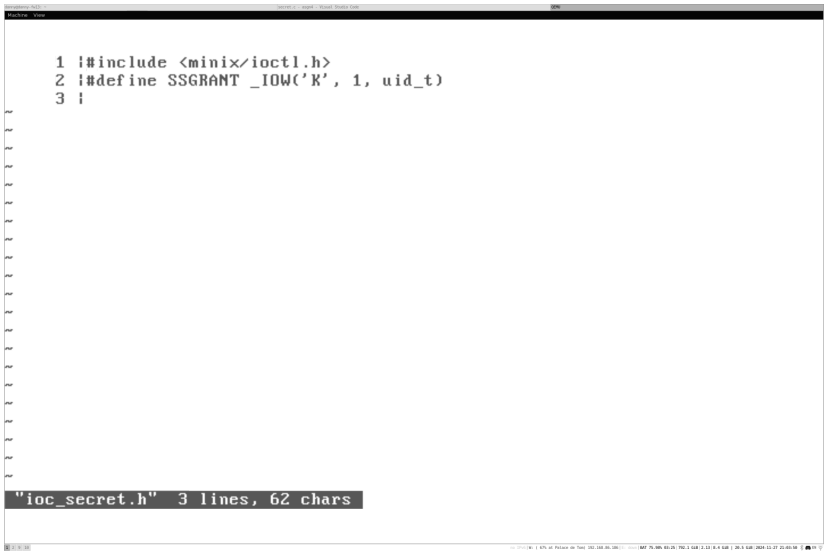
/usr/include/sys/ioctl.h



```
 4 ¦ * This header file includes all other ioctl command code headers.
 5 ¦ */
 6 ¦
 7 ¦#ifndef _S_IOCTL_H
 8 ¦#define _S_IOCTL_H
 9 ¦
10 ¦/* A driver that uses ioctls claims a character for its series of comman
11 ¦ * For instance:  #define TCGETS  _IOR('T',  8, struct termios)
12 ¦ * This is a terminal ioctl that uses the character 'T'.  The character(
13 ¦ * used in each header file are shown in the comment following.
14 ¦ */
15 ¦
16 ¦#include <sys/ioc_tty.h>        /* 'T' 't' 'k'          */
17 ¦#include <net/ioctl.h>          /* 'n'                  */
18 ¦#include <sys/ioc_disk.h>       /* 'd'                  */
19 ¦#include <sys/ioc_file.h>       /* 'f'                  */
20 ¦#include <sys/ioc_memory.h>     /* 'm'                  */
21 ¦#include <sys/ioc_cmos.h>       /* 'c'                  */
22 ¦#include <sys/ioc_tape.h>       /* 'M'                  */
23 ¦#include <sys/ioc_scsi.h>       /* 'S'                  */
24 ¦#include <sys/ioc_sound.h>      /* 's'                  */
25 ¦#include <sys/ioc_secret.h>     /* 'K'                  */
26 ¦
27 ¦#endif /* _S_IOCTL_H */
```

The next thing I did was change the ioctl.h file to include my new, smaller header file called ioc_secret.h. This was important because ioctl is special set of instructions that lets us talk to the device driver in ways other than just reading or writing. SSGRANT allows us to give ownership of the device to another user. The letter K was used as a tag to tell the system which driver.

/usr/include/sys/ioc_secret.h



```
 1 ¦#include <minix/ioctl.h>
 2 ¦#define SSGRANT _IOW('K', 1, uid_t)
 3 ¦
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
"ioc_secret.h"  3 lines, 62 chars
```

Courtesy of Dr. Nico this given to us. When a process uses the `SSGRANT` command, the UID passed becomes the new owner of the secret.

I broke this assignment down into smaller parts to make it easier to complete: copying the hello example, modifying it to take in input, modifying it to hold the one secret, modifying it to be locked to 1 user, transferring users, then saving state.

Src Code

secret.c

```
#include "secret.h"


/* Function prototypes for secret driver */
PRIVATE char* secret_name(void);
PRIVATE int secret_open(struct driver* d, message* m);
PRIVATE int secret_close(struct driver* d, message* m);
PRIVATE struct device* secret_prepare(int device);
PRIVATE int secret_transfer(int procnr, int opcode,
                  u64_t position, iovec_t* iov, unsigned nr_req);
PRIVATE void secret_geometry(struct partition* entry);
PRIVATE int secret_ioctl(struct driver* d, message* m);

/* SEF functions */
PRIVATE void sef_local_startup(void);
PRIVATE int sef_cb_init(int type, sef_init_info_t* info);
PRIVATE int sef_cb_lu_state_save(int);
PRIVATE int lu_state_restore(void);

/* Entry points to the hello driver */
PRIVATE struct driver secret_tab = {
   secret_name,
   secret_open,
   secret_close,
   secret_ioctl,
   secret_prepare,
   secret_transfer,
   nop_cleanup,
   secret_geometry,
   nop_alarm,
   nop_cancel,
   nop_select,
   do_nop,
};

/* the device */
PRIVATE struct device secret_device;

/* state variables */
PRIVATE int fd_open_counter;
PRIVATE int is_readable;
PRIVATE int owned;
PRIVATE uid_t owner_uid;
PRIVATE int read_pos;
PRIVATE int write_pos;
PRIVATE char buffer[SECRET_SIZE];
```

```
/* gets the name of the device */
PRIVATE char* secret_name(void) {
   return "secret";
}


/* opens the device if free */
PRIVATE int secret_open(struct driver* d, message* m) {
   struct ucred caller_process;
   int reading, writing;
   /*
   In brightest code, in darkest compile, no relic of C89 shall beguile.
   Let those who cling to its ancient style, beware
   the future—it's worth the trial.

   ( courtesy of chatgpt for this poem but seriously i hate c89,
       c99 for life. let me declare variables anywhere)
   */

   getnucred(m->IO_ENDPT, &caller_process);
   reading = m->COUNT & R_BIT;
   writing = m->COUNT & W_BIT;

   if (!owned) {
       if (reading && writing) {
           printf("[OPEN] tring to RW while free\n");
           return EACCES;
       }
       else if (reading) {
           is_readable = FALSE;
       }
       else { /* writing to it so it becomes readable" */
           is_readable = TRUE;
       }
       owner_uid = caller_process.uid;
       owned = TRUE;
   }
   else {
       if (writing) { /* can only write once */
           printf("[OPEN] trying to write while owned\n");
           return EACCES;
       }
       else {
           if (owner_uid != caller_process.uid) {
               printf("[OPEN] trying to read while owned by someone else\n");
               return EACCES;
           }
           is_readable = FALSE;
       }
```

```
    }

    fd_open_counter++;
    return OK;
}

/* closes the device */
PRIVATE int secret_close(struct driver* d, message* m) {
    struct ucred caller_process;

    getnucred(m->IO_ENDPT, &caller_process);

    fd_open_counter--;

    if (fd_open_counter == 0 && !is_readable) {
        owner_uid = NO_OWNER_ID;
        owned = FALSE;
        memset(buffer, 0, SECRET_SIZE);
        write_pos = 0;
        read_pos = 0;
    }

    return OK;
}


/* switch owners */
PRIVATE int secret_ioctl(struct driver* d, message* m) {
    uid_t new_owner_uid;
    int ret;

    if (m->REQUEST != SSGRANT) {
        return ENOTTY;
    }

    ret = sys_safecopyfrom(m->IO_ENDPT, (vir_bytes)m->IO_GRANT, 0,
                                (vir_bytes)&new_owner_uid, sizeof(uid_t), D);
    owner_uid = new_owner_uid;

    return ret;
}


/* prepare the device (stolen from hello) */
PRIVATE struct device* secret_prepare(int device) {
    secret_device.dv_base.lo = 0;
    secret_device.dv_base.hi = 0;
    secret_device.dv_size.lo = 0;
    secret_device.dv_size.hi = 0;
```

```
   return &secret_device;
}


/* transfer data to/from the device */
PRIVATE int secret_transfer(int procnr, int opcode,
                  u64_t position, iovec_t* iov, unsigned nr_req) {
   int bytes;
   int ret;

   switch (opcode) {
       case DEV_GATHER_S: /* read from device */
           bytes = write_pos - read_pos < iov->iov_size ?
                   write_pos - read_pos : iov->iov_size;
           if (bytes <= 0) { /* unsuccessful read but continue on */
               return OK;
           }

           if (ret = sys_safecopyto(procnr, iov->iov_addr, 0,
                                    (vir_bytes)(buffer + read_pos),
                                    bytes, D) != OK) {
               return ret;
           }

           iov->iov_size -= bytes;
           read_pos += bytes;
           break;

       case DEV_SCATTER_S: /* write to device */
           bytes = SECRET_SIZE - write_pos < iov->iov_size ?
                   SECRET_SIZE - write_pos : iov->iov_size;
           if (bytes <= 0) { /* unsuccessful write but continue on */
               return ENOSPC;
           }

           if (ret = sys_safecopyfrom(procnr, iov->iov_addr, 0,
                                    (vir_bytes)(buffer + write_pos),
                                    bytes, D) != OK) {
               return ret;
           }

           iov->iov_size -= bytes;
           write_pos += bytes;
           break;

       default:
           return EINVAL;
   }
```

```
   return ret;
}


/* stolen from hello */
PRIVATE void secret_geometry(struct partition* entry) {
   printf("hello_geometry()\n");
   entry->cylinders = 0;
   entry->heads = 0;
   entry->sectors = 0;

   return;
}


/* save all the variables to be restored later */
PRIVATE int sef_cb_lu_state_save(int state) {
   SecretState_t cur_secret;
   cur_secret.fd_open_counter = fd_open_counter;
   cur_secret.owner_uid = owner_uid;
   cur_secret.owned = owned;
   cur_secret.write_pos = write_pos;
   cur_secret.read_pos = read_pos;
   memcpy(cur_secret.buffer, buffer, SECRET_SIZE);

   ds_publish_mem(SECRET_STATE_NAME, (char*)&cur_secret,
                  sizeof(SecretState_t), DSF_OVERWRITE);

   return OK;
}


PRIVATE int lu_state_restore() {
   SecretState_t restored_secret;
   size_t len;

   ds_retrieve_mem(SECRET_STATE_NAME, (char*)&restored_secret, &len);

   fd_open_counter = restored_secret.fd_open_counter;
   owner_uid = restored_secret.owner_uid;
   owned = restored_secret.owned;
   write_pos = restored_secret.write_pos;
   read_pos = restored_secret.read_pos;
   memcpy(buffer, restored_secret.buffer, SECRET_SIZE);

   ds_delete_mem(SECRET_STATE_NAME);

   return OK;
}
```

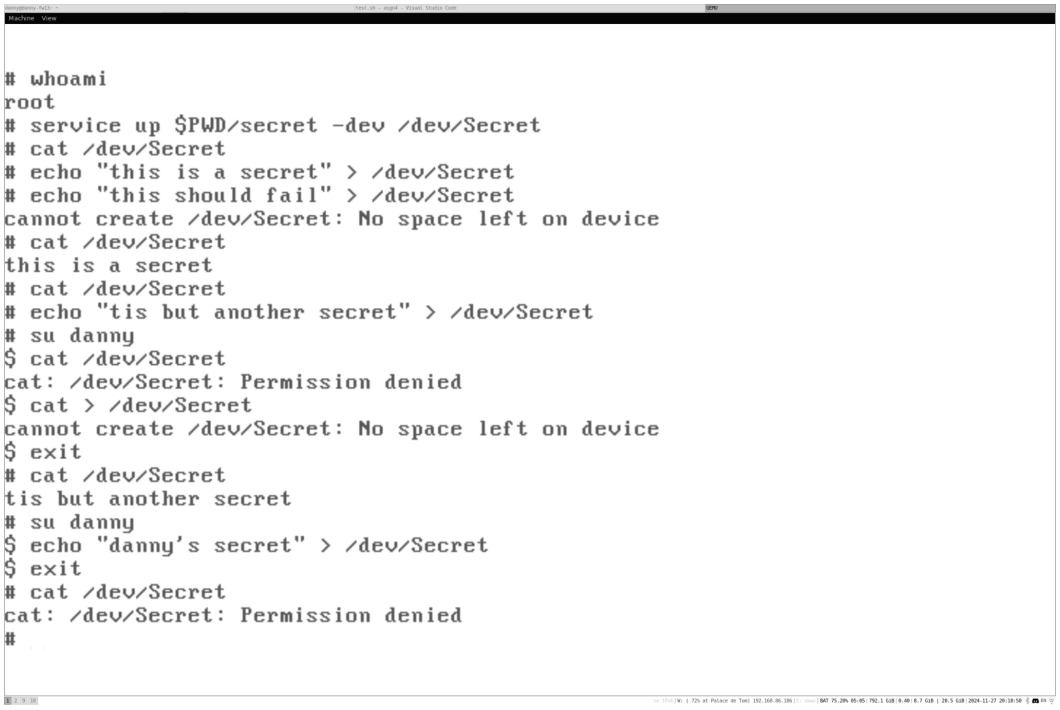
```
/* stolen from hello */
PRIVATE void sef_local_startup() {
   /* Register init callbacks. Use the same function for all event types */
   sef_setcb_init_fresh(sef_cb_init);
   sef_setcb_init_lu(sef_cb_init);
   sef_setcb_init_restart(sef_cb_init);

   /* Register live update callbacks */
   /* - Agree to update immediately when LU is requested in a valid state. */
   sef_setcb_lu_prepare(sef_cb_lu_prepare_always_ready);
   /* - Support live update starting from any standard state. */
   sef_setcb_lu_state_isvalid(sef_cb_lu_state_isvalid_standard);
   /* - Register a custom routine to save the state. */
   sef_setcb_lu_state_save(sef_cb_lu_state_save);
   /* Let SEF perform startup. */
   sef_startup();

   return;
}


/* initialize the driver */
PRIVATE int sef_cb_init(int type, sef_init_info_t *info) {
   int do_announce_driver = TRUE;

   switch(type) {
       case SEF_INIT_FRESH:
           fd_open_counter = 0;
           is_readable = FALSE;
           owned = FALSE;
           owner_uid = NO_OWNER_ID;
           read_pos = 0;
           write_pos = 0;
           memset(buffer, 0, SECRET_SIZE);
           break;
       case SEF_INIT_LU:
           lu_state_restore();
           do_announce_driver = FALSE;
           printf("Secret driver: live update\n");
           break;
       case SEF_INIT_RESTART:
           printf("Secret driver: restart\n");
           break;
   }

   /* announce when up */
   if (do_announce_driver) {
```

```
        driver_announce();
    }
    /* been initialized properly */
    return OK;
}



/* stolen from hello */
PUBLIC int main(int argc, char **argv) {
    /*
     * Perform initialization.
     */
    sef_local_startup();

    /*
     * Run the main loop.
     */
    driver_task(&secret_tab, DRIVER_STD);

    return OK;
}
```

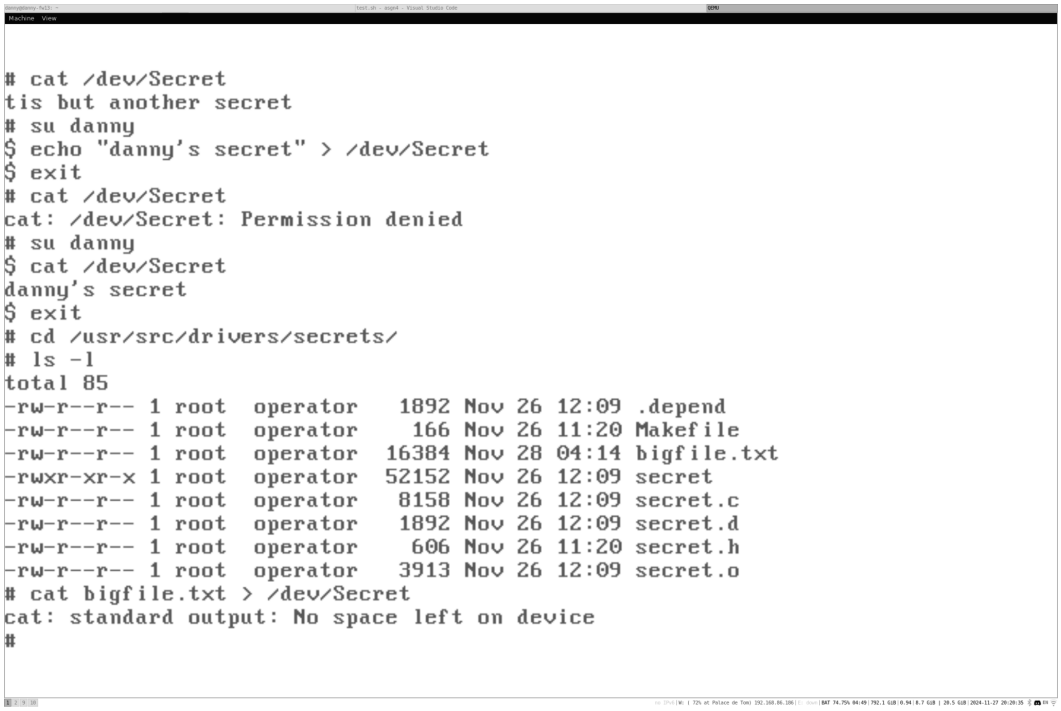
Secret.h

```
#ifndef SECRET_H
#define SECRET_H

#include <minix/drivers.h>
#include <minix/driver.h>
#include <stdio.h>
#include <stdlib.h>
#include <minix/ds.h>
#include <sys/ioc_secret.h>

#define SECRET_MSG ("I am a secret hehe")
#define NO_OWNER_ID  (-1)

#ifndef SECRET_SIZE
#define SECRET_SIZE (8192)
#endif

/* struct for simplifing saving state */
typedef struct SecretState_t {
   int fd_open_counter;
   int is_readable;
   int owned;
   uid_t owner_uid;
   int read_pos;
   int write_pos;
   char buffer[SECRET_SIZE];
} SecretState_t;
#define SECRET_STATE_NAME ("secret_state")

#endif /* secret.h */
```

Running Behavior



```
# whoami
root
# service up $PWD/secret -dev /dev/Secret
# cat /dev/Secret
# echo "this is a secret" > /dev/Secret
# echo "this should fail" > /dev/Secret
cannot create /dev/Secret: No space left on device
# cat /dev/Secret
this is a secret
# cat /dev/Secret
# echo "tis but another secret" > /dev/Secret
# su danny
$ cat /dev/Secret
cat: /dev/Secret: Permission denied
$ cat > /dev/Secret
cannot create /dev/Secret: No space left on device
$ exit
# cat /dev/Secret
tis but another secret
# su danny
$ echo "danny's secret" > /dev/Secret
$ exit
# cat /dev/Secret
cat: /dev/Secret: Permission denied
#
```



```
# cat /dev/Secret
tis but another secret
# su danny
$ echo "danny's secret" > /dev/Secret
$ exit
# cat /dev/Secret
cat: /dev/Secret: Permission denied
# su danny
$ cat /dev/Secret
danny's secret
$ exit
# cd /usr/src/drivers/secrets/
# ls -l
total 85
-rw-r--r-- 1 root  operator   1892 Nov 26 12:09 .depend
-rw-r--r-- 1 root  operator    166 Nov 26 11:20 Makefile
-rw-r--r-- 1 root  operator  16384 Nov 28 04:14 bigfile.txt
-rwxr-xr-x 1 root  operator  52152 Nov 26 12:09 secret
-rw-r--r-- 1 root  operator   8158 Nov 26 12:09 secret.c
-rw-r--r-- 1 root  operator   1892 Nov 26 12:09 secret.d
-rw-r--r-- 1 root  operator    606 Nov 26 11:20 secret.h
-rw-r--r-- 1 root  operator   3913 Nov 26 12:09 secret.o
# cat bigfile.txt > /dev/Secret
cat: standard output: No space left on device
#
```

The devicer is working as intended verified by running the same run as in the assignment spec. As I got the same result as in the sample run and passed all the test on test suite I deemed the driver as working.

Problems Encountered

The only problem I encountered was where to place the ioctl include for ioc_secret. I initially edited `/usr/src/include/sys/ioctl.h` and when trying to compile my driver I encountered `error: unable to open ioc_secret.h`. After running `find / -name "ioctl.h"` and placing it in `usr/src/include/sys/ioctl.h` it worked.