Heriberto Rodriguez
Fernando Diaz
CPE 453 - 03

Asgn. 4 Secret Device Driver

Approach

Before diving into coding our device driver, we looked over the device driver tutorial and the reading assigned in the assignment description. The tutorial showed us how to start up, update, and shutdown the hello device. After we had a basic understanding of how to interface with a device we began modifying the hello device driver until we had a working secret device driver.

Architecture

Our driver is expected to function very much like the hello driver. Since all drivers have the same list of functions, we simply had to copy the hello driver and change it's individual function behaviors. Obviously our driver had a lot more restrictions than the hello driver, as well as allowing the ability to write to the device.

Implementation

Development Environment:

Version of MINIX kernel is 3.1.8. It was installed as a VM via Virtual Box.

Files Modified:

All of the files in the hello driver directory were copied and modified to be of use for our secret device. Most of the files, including the Makefile, the .conf file, and the .d file simply involved us changing variable names to 'secret'. The header file was modified to only define our max secret length. Hello.c was greatly changed to assure behavior was exactly as described in the specifications. A few global variables as well as function prototypes were declared for use throughout the file. hello_open() and hello_transfer() were the two functions changed the most. Inside these functions we do multiple checks for things such as message type and message sender.

Problems Encountered

1. One of the road blocks we ran into is not being able to get the uid of the current user logged in. We tried using the *m_source field* of a message structure as the *endpoint_t* to the *getnucred* function, but that always set the real user id to 0.

2. When we were testing if a user that owns the device can write and read from a buffer and then another user can write to the buffer we were getting a permission error. The buffer was being emptied, but another user could not write to it.

3. Another problem encountered was still keeping the secret after the initial write fd had closed. Our initial approach involved just clearing the secret once our open counter had reached zero. So we had the problem of having the initial write increment the counter and decrement when done, which essentially just cause a write and immediate erase.

Heriberto Rodriguez
Fernando Diaz
CPE 453 - 03

4. Writing a message whose size was larger than the size of the buffer was not resulting in an error. Instead, the maximum amount of characters that fit into the buffer were being written to it.

5. For error handling we weren't sure how to set the correct error. Whenever there was incorrect usage or permission errors we were setting errno to the appropriate error code and then printing using perror(). This resulted in strange output that made no sense.

Solutions

1. The solution to getting the real uid of the user currently logged in was to use the IO_ENDPT field of the message structure in *secret_open().*

2. The reason why we were getting a permission error when a new user was attempting to write to an empty buffer was because we forgot to reset the permissions of the device when the buffer was being emptied.

3. In order to make sure we didn't clear the secret after the initial write fd had closed, we had to make sure we only cleared the secrets after reads were performed. Therefore, since the initial fd was only performing a write we didn't bother checking the open count. The counter would still be set to zero after the write, but the secret would still remain this way.

4. When checking for the size of the message typed by the user we were using the bytes local variable in *secret_transfer()*. This variable was set to the the max size of the buffer when the message was longer than the size of the buffer. We fixed this by using *iov->iov_size* instead to get the size of the message that the user typed in.

5. We realized that when handling errors, we just needed to return the appropriate error to the calling function, in this case driver_task(). The calling function would then appropriately handle the error that was returned from our function. We also noticed that driver_task() would print the error message in certain situations, but not others, so for those cases we output our own error messages.

Lessons Learned

In this assignment we learned how to interface with a device file through a device driver and how to make a character device file. In terms of writing a device driver, we learned that MINIX separates device driver independent software with device driver dependent software. This makes MINIX more modular because we only need to provide functions for our specific device through a driver structure. All devices are controlled by the driver_task() which calls the functions that we provided to the driver structure.

```c
#ifndef __SECRET_H
#define __SECRET_H

/** The Hello, World! message. */
#define HELLO_MESSAGE "Hello, World!\n"
#define SECRET_SIZE 8192

#endif /* __SECRET_H */
```

```c
#include <minix/drivers.h>
#include <minix/driver.h>
#include <sys/types.h>
#include <sys/ioctl.h>
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <minix/ds.h>
#include "secret.h"

#define O_RDWR 6
#define O_WRONLY 2

char secret[SECRET_SIZE];
uid_t owner = -1;
struct ucred e;
int wr_counter = 0;

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

/* SEF functions and variables. */
FORWARD _PROTOTYPE( void sef_local_startup, (void) );
FORWARD _PROTOTYPE( int sef_cb_init, (int type, sef_init_info_t *info) );
FORWARD _PROTOTYPE( int sef_cb_lu_state_save, (int) );
FORWARD _PROTOTYPE( 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,
```

```
46      secret_transfer,
47      nop_cleanup,
48      secret_geometry,
49      nop_alarm,
50      nop_cancel,
51      nop_select,
52      nop_ioctl,
53      do_nop,
54  };
55
56  /** Represents the /dev/hello device. */
57  PRIVATE struct device secret_device;
58
59  /** State variable to count the number of times the device has been opened. *
60  PRIVATE int open_counter;
61
62  PRIVATE char * secret_name(void)
63  {
64      printf("secret_name()\n");
65      return "hello";
66  }
67
68  PRIVATE int secret_ioctl(d, m)
69      struct driver *d;
70      message *m;
71  {
72      uid_t grantee;
73
74      if (m->REQUEST == SSGRANT)
75      {
76         sys_safecopyfrom(m->IO_ENDPT, (vir_bytes)m->IO_GRANT,
77           0, (vir_bytes)&grantee, sizeof(grantee), D);
78         owner = grantee;
79      }
80      else
81      {
82         printf("invalid ioctl request\n");
83         return ENOTTY;
84      }
85
86      return OK;
87  }
88
89  PRIVATE int secret_open(d, m)
90      struct driver *d;
```

```
91      message *m;
92  {
93      getnucred(m->IO_ENDPT, &e);
94
95      /*printf("secret_open(). Called %d time(s).\n", ++open_counter);*/
96      if (m->COUNT == O_RDWR)
97      {
98         printf("Can't open device with read write access\n");
99         return EACCES;
100     }
101     else if (m->COUNT & W_BIT)
102     {
103        if (e.uid != owner && owner != -1)
104          return EACCES;
105        if (strlen(secret) != 0)
106        {
107           /*printf("cannot create /dev/Secret: No space left on device\n");*/
108           return ENOSPC;
109        }
110     }
111     ++open_counter;
112
113     if (owner == -1)
114        owner = e.uid;
115     return OK;
116 }
117
118 PRIVATE int secret_close(d, m)
119     struct driver *d;
120     message *m;
121 {
122     /*printf("secret_close()\n");*/
123     return OK;
124 }
125
126 PRIVATE struct device * secret_prepare(dev)
127     int dev;
128 {
129     secret_device.dv_base.lo = 0;
130     secret_device.dv_base.hi = 0;
131     secret_device.dv_size.lo = SECRET_SIZE;
132     secret_device.dv_size.hi = 0;
133     return &secret_device;
134 }
135
```

```
136 PRIVATE int secret_transfer(proc_nr, opcode, position, iov, nr_req)
137     int proc_nr;
138     int opcode;
139     u64_t position;
140     iovec_t *iov;
141     unsigned nr_req;
142 {
143     int bytes, ret;
144 
145     /*printf("secret_transfer()\n");*/
146 
147     bytes = SECRET_SIZE - position.lo < iov->iov_size ?
148             SECRET_SIZE - position.lo : iov->iov_size;
149 
150     if (bytes <= 0)
151     {
152         return OK;
153     }
154     switch (opcode)
155     {
156         case DEV_GATHER_S:
157             if (owner == -1 || owner == e.uid)
158             {
159                open_counter--;
160                ret = sys_safecopyto(proc_nr, iov->iov_addr, 0,
161                                     (vir_bytes) (secret + position.lo),
162                                      bytes, D);
163                iov->iov_size -= bytes;
164                if (open_counter == 0)
165                {
166                   memset(secret, 0, strlen(secret));
167                   owner = -1;
168                }
169             }
170             else
171             {
172                --open_counter;
173                ret = EACCES;
174             }
175             break;
176         case DEV_SCATTER_S:
177             --open_counter;
178             if (iov->iov_size > SECRET_SIZE)
179             {
180                 printf("Input too big\n");
```

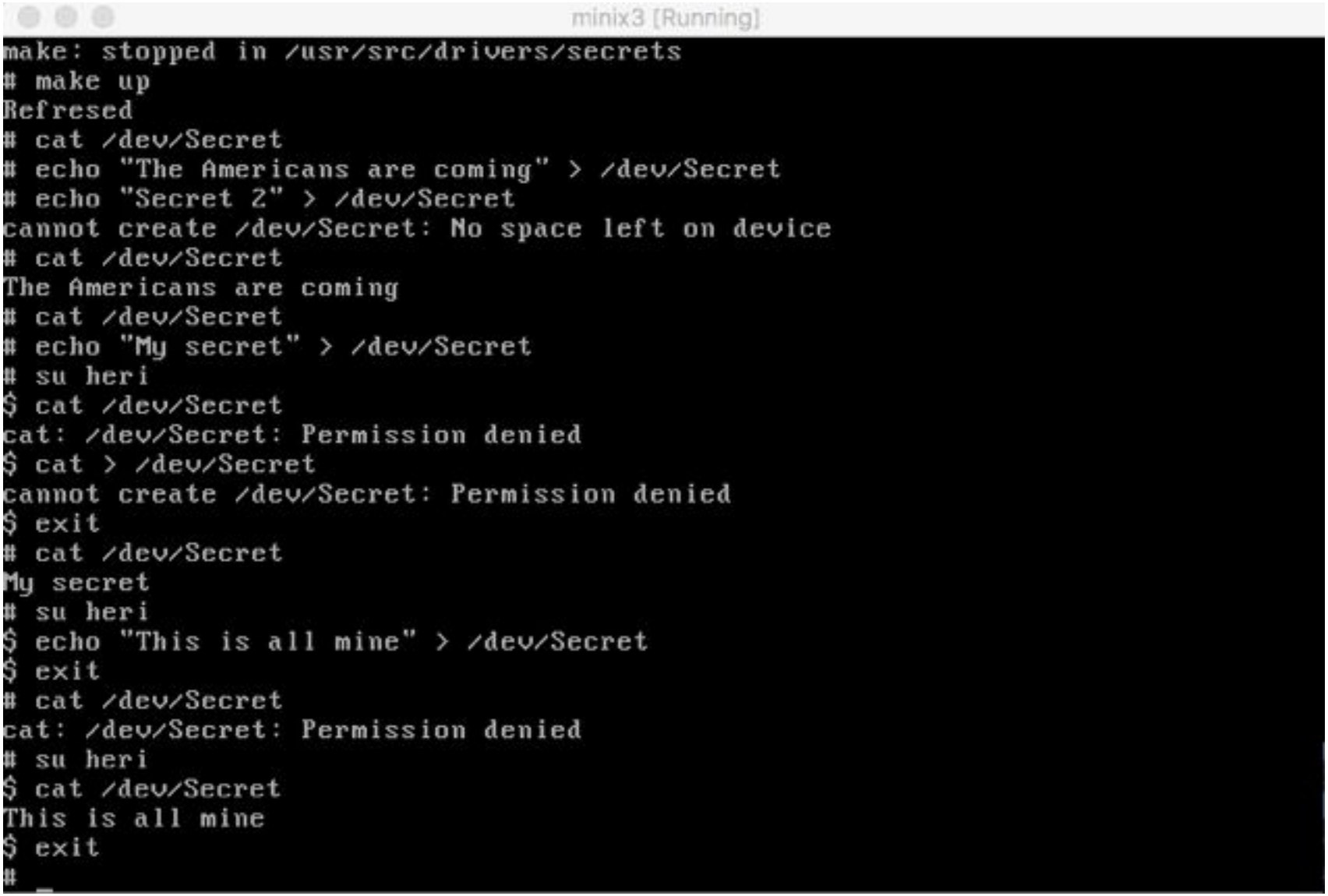
```
181                 return ENOSPC;
182             }
183             ret = sys_safecopyfrom(proc_nr, iov->iov_addr, 0,
184                                    (vir_bytes) (secret + position.lo),
185                                     bytes, D);
186 
187             iov->iov_size -= bytes;
188             break;
189         default:
190             return EINVAL;
191     }
192     return ret;
193 }
194 
195 PRIVATE void secret_geometry(entry)
196     struct partition *entry;
197 {
198     printf("secret_geometry()\n");
199     entry->cylinders = 0;
200     entry->heads     = 0;
201     entry->sectors   = 0;
202 }
203 
204 PRIVATE int sef_cb_lu_state_save(int state) {
205 /* Save the state. */
206     ds_publish_u32("open_counter", open_counter, DSF_OVERWRITE);
207 
208     return OK;
209 }
210 
211 PRIVATE int lu_state_restore() {
212 /* Restore the state. */
213     u32_t value;
214 
215     ds_retrieve_u32("open_counter", &value);
216     ds_delete_u32("open_counter");
217     open_counter = (int) value;
218 
219     return OK;
220 }
221 
222 PRIVATE void sef_local_startup()
223 {
224     /*
225      * Register init callbacks. Use the same function for all event types
```

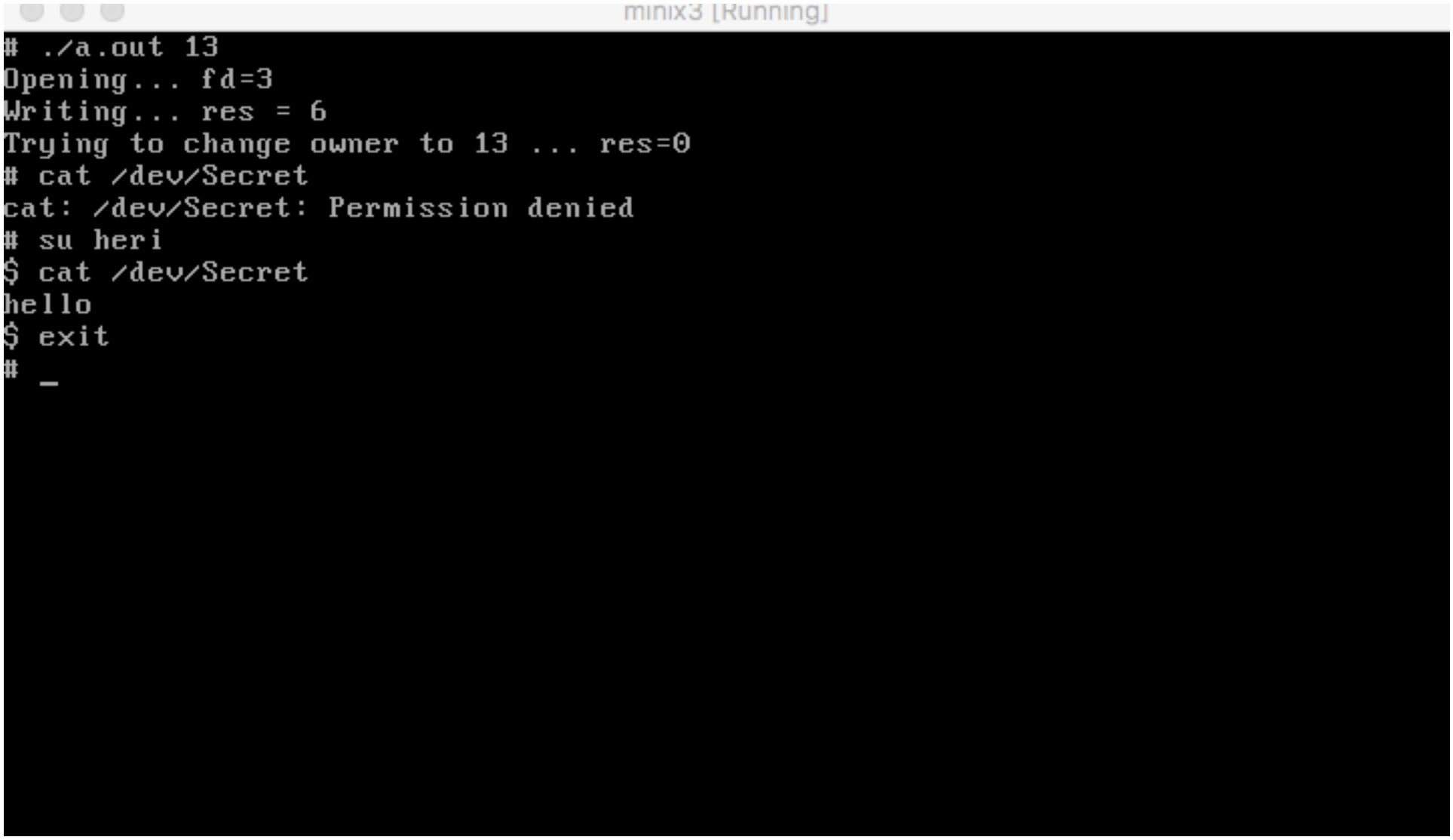
```
226      */
227     sef_setcb_init_fresh(sef_cb_init);
228     sef_setcb_init_lu(sef_cb_init);
229     sef_setcb_init_restart(sef_cb_init);
230 
231     /*
232      * Register live update callbacks.
233      */
234     /* - Agree to update immediately when LU is requested in a valid state. *
235     sef_setcb_lu_prepare(sef_cb_lu_prepare_always_ready);
236     /* - Support live update starting from any standard state. */
237     sef_setcb_lu_state_isvalid(sef_cb_lu_state_isvalid_standard);
238     /* - Register a custom routine to save the state. */
239     sef_setcb_lu_state_save(sef_cb_lu_state_save);
240 
241     /* Let SEF perform startup. */
242     sef_startup();
243 }
244 
245 PRIVATE int sef_cb_init(int type, sef_init_info_t *info)
246 {
247 /* Initialize the hello driver. */
248     int do_announce_driver = TRUE;
249 
250     open_counter = 0;
251     switch(type) {
252         case SEF_INIT_FRESH:
253             printf("Refresed\n");
254         break;
255 
256         case SEF_INIT_LU:
257             /* Restore the state. */
258             lu_state_restore();
259             do_announce_driver = FALSE;
260 
261             printf("Hey, I'm a new version!\n");
262         break;
263 
264         case SEF_INIT_RESTART:
265             printf("Hey, I've just been restarted!\n");
266         break;
267     }
268 
269     /* Announce we are up when necessary. */
270     if (do_announce_driver) {
```

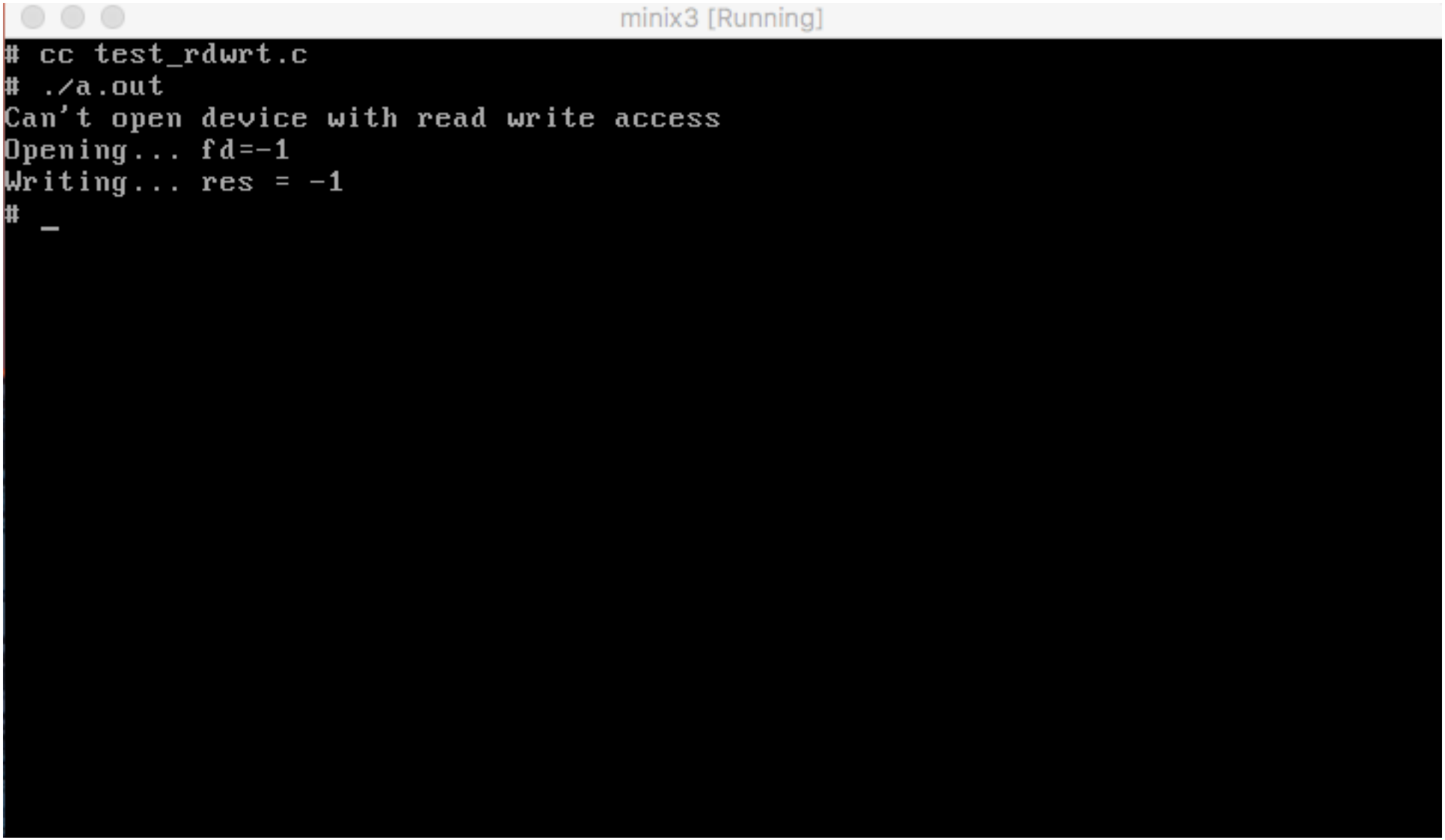
```
271         driver_announce();
272     }
273 
274     /* Initialization completed successfully. */
275     return OK;
276 }
277 
278 PUBLIC int main(int argc, char **argv)
279 {
280     /*
281      * Perform initialization.
282      */
283     sef_local_startup();
284 
285     /*
286      * Run the main loop.
287      */
288     driver_task(&secret_tab, DRIVER_STD);
289     return OK;
290 }
291 
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ioctl.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>

int main(int argc, char *argv[])
{
   int fd, res;
   char *msg = "hello\n";
   uid_t uid;

   fd = open("/dev/Secret", O_WRONLY);
   printf("Opening... fd=%d\n", fd);
   res = write(fd, msg, strlen(msg));
   printf("Writing... res = %d\n", res);

   /*try grant*/
   if (argc > 1 && 0 != (uid=atoi(argv[1]))) {
      if (res = ioctl(fd, SSGRANT, &uid))
         perror("ioctl");
      printf("Trying to change owner to %d ... res=%d\n", uid, res);
   }
   return 0;
}

```

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ioctl.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>

int main(int argc, char *argv[])
{
   int fd, res;
   char *msg = "hello";
   uid_t uid;

   fd = open("/dev/Secret", O_RDWR);
   printf("Opening... fd=%d\n", fd);
   res = write(fd, msg, strlen(msg));
   printf("Writing... res = %d\n", res);

   return 0;
}

```


minix3 [Running]
make: stopped in /usr/src/drivers/secrets
make up
Refresed
cat /dev/Secret
echo "The Americans are coming" > /dev/Secret
echo "Secret 2" > /dev/Secret
cannot create /dev/Secret: No space left on device
cat /dev/Secret
The Americans are coming
# cat /dev/Secret
# echo "My secret" > /dev/Secret
# su heri
$ cat /dev/Secret
cat: /dev/Secret: Permission denied
$ cat > /dev/Secret
cannot create /dev/Secret: Permission denied
$ exit
# cat /dev/Secret
My secret
# su heri
$ echo "This is all mine" > /dev/Secret
$ exit
# cat /dev/Secret
cat: /dev/Secret: Permission denied
# su heri
$ cat /dev/Secret
This is all mine
$ exit
#


minix3 [Running]
./a.out 13
Opening... fd=3
Writing... res = 6
Trying to change owner to 13 ... res=0
cat /dev/Secret
cat: /dev/Secret: Permission denied
# su heri
$ cat /dev/Secret
hello
$ exit
# _


minix3 [Running]
cc test_rdwrt.c
./a.out
Can't open device with read write access
Opening... fd=-1
Writing... res = -1
_