Assignment 4

cpe 453 Winter 2023

```
Three may keep a secret, if two of them are dead.
-- Benjamin Franklin
```

— /usr/games/fortune

Due by 11:59:59pm, Monday, February 27th.
This assignment is to be done individually.

A new device: `/dev/Secret`

Every so often it is necessary to store secret information or to pass a secret securely to another process. In this assignment, your job is to create a new device, `/dev/Secret`, to do just this.

The device will be a character-special device. The major/minor numbers don't matter much other than that they need to not be currently in use. I recommend major number 20 and minor number 0. Create the device special file (once) using `mknod(8)`:

```
mknod /dev/Secret c 20 0
```

Now that we have a device, the driver's behavior can be described as follows:

- Opening: `/dev/Secret` can hold only one secret at a time. How it behaves will depend on whether the device is empty or full.

  If empty (owned by nobody):

  - Any process may open `/dev/Secret` for reading or writing.
  - That owner of that process will then become the owner of the secret. (determined via `getnucred(2)`)
  - Open for writing can *only* succeed if the secret is not owned by anybody. This means it may only be opened for writing once.
  - The device may not be opened for read-write access (because it makes no sense). This results in a permission denied error (`EACCES`).

  If full (owned by somebody):

  - `/dev/Secret` may not be opened for writing once it is holding a secret.
  - `/dev/Secret` may be opened for reading by a process owned by the owner of the secret. You must keep track of how many open file descriptors there are, however, because the secret resets when the last file descriptor closes after a read file descriptor has been opened[1].
  - 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`).

[1] That is, the secret can persist after the initial write fd has been closed, but once anybody has opened it for reading, when the open count goes to zero the secret goes back to being empty.

- Closing: when the last file descriptor is closed after any read file descriptor has been opened, `/dev/Secret` reverts to being empty.
- The secret held by `/dev/Secret` may be of fixed size. Exactly how big doesn't matter, but it should be settable by defining the macro `SECRET_SIZE` in your driver's source. Attempts to write more into the device than will fit will result in an `ENOSPC` response.

  The test harness will expect your buffer size to be 8192 (8KB), but, of course, this should be configurable by changing a `SECRET_SIZE`. To make it reconfigureable while compiling (do), define it like:

```
#ifndef SECRET_SIZE    /* only define it if not already defined */
#define SECRET_SIZE 8192
#endif
```

- `/dev/Secret` supports a single `ioctl(2)` call, `SSGRANT`, which allows the owner of a secret to change the ownership to another user. E.g.:

```
ioctl(fd, SSGRANT, &other_uid);
```

  Any `ioctl(2)` requests other than `SSGRANT` get a `ENOTTY` response.
- `/dev/Secret` should preserve its state over live update events.

Your Task

Your task is to create a Secret Keeper device for MINIX that demonstrates the behavior above. (See §2.6.6 (among others) for information about how the various system tasks and device drivers get started.) Creating this driver will require modifications to various portions of the MINIX system.

To Do List

This is not necessarily complete

1. Get over any residual fear you may have of reading kernel source and/or system header files.
2. Know, love, and become one with `http://wiki.minix3.org/doku.php?id=developersguide:driverprogramming`[2] (linked from the class web page.) This is for version 3.3, but there's a hello driver appropriate to your version in `/usr/src/drivers/hello` on your system.
3. Add `secretkeeper` to `/etc/system.conf`
4. Create your device file, `/dev/Secret`
5. Create your driver source directory in `/usr/src/drivers/secrets`, (or whatever you want to call it) by copying and gutting the `hello` driver.
6. Add `SSGRANT` to `/usr/src/include/sys/ioctl.h`. The meaning of the ioctl request encoding is described in `/usr/src/include/minix/ioctl.h` if you're interested, but what you'll want to do is to add the following line to `<sys/ioctl.h>`:

```
#include <sys/ioc_secret.h> /* 'K' */
```

[2] The version for 3.1.8 is availlable at `https://wiki.minix3.org/doku.php?id=developersguide:driverprogramming&rev=1425574251`. Go version control.

and then create <`sys/ioc_secret.h`> containing the magic lines:

```
#include <minix/ioctl.h>
#define SSGRANT _IOW('K', 1, uid_t)
```

Remember to copy these files over to `/usr/include/sys/` where other programs will be able to see them.

7. Write your device driver and test program(s).

8. Test it until satisfied or out of time.

9. Write your report, including:
   - Source for driver
   - Program(s) demonstrating driver's proper functioning
   - A screenshot or typescript of it working

A note on memory addressing

All of the memory addresses for buffers that are passed around in messages are virtual addresses in the address space of the requesting process that originally requested the IO. In order to read from or write to data in another process, your driver will need the help of the system task. The system task exists in kernel space and can read or write any portion of anybody's address space. To do this, you'll use the functions `sys_safecopyfrom()` and `sys_safecopyto()`:

```
int sys_safecopyfrom (
     endpoint_t         source,        /* source process                               */
     cp_grant_id_t      grant,         /* source buffer                                */
     vir_bytes          grant_offset,  /* offset in source buffer (for block devs)     */
     vir_bytes          my_address,    /* virtual address of destination buffer        */
     size_t             bytes,         /* bytes to copy                                */
     int                my_seg         /* memory segment (It's 'D' :-)                 */
);


int sys_safecopyto (
     endpoint_t         source,        /* destination process                          */
     cp_grant_id_t      grant,         /* destination buffer                           */
     vir_bytes          grant_offset,  /* offset in destination buffer (for block devs) */
     vir_bytes          my_address,    /* virtual address of source buffer             */
     size_t             bytes,         /* bytes to copy                                */
     int                my_seg         /* memory segment (It's 'D' :-)                 */
);
```

Tricks and Tools

Most of what you need to know is embedded in the "How MINIX works" portions of the Tanenbaum and Woodhull book. In particular, read §3.4.2–3.5.3 about the architecture of MINIX device drivers and §5.7.7 about how devices interact with the filesystem. It probably wouldn't hurt to look into the man page for `mknod(8)` so you know how to create the `Secret` device.

Some good advice: You might want to try and work out some of your driver behavior in user-space before diving in to write the device driver.

Some extremely good advice: **Be sure you understand how the kernel is working now, before you modify it!** This includes its makefiles and other support structures.

In no particular order, useful things to know:

- The entire System Event Framework (SEF) is based on callbacks enumerated in the `struct driver` you provide to it when you call `driver_task()` in `main()`. The SEF handles all the general device driver activities like receiving messages, and responding, but calls you when you have to do something specific to your device.

- Preserving state over an update event is demonstrated in the `/dev/hello` device tutorial in the `sef_cb_lu_state_save()` and `sef_cb_init()` callbacks. In particular, you will use the following three functions (defined in `ds.h`) to save anything you care about to a named store and then retrieve it:

```
int ds_publish_mem, (const char *ds_name, void *vaddr, size_t length, int flags);
int ds_retrieve_mem, (const char *ds_name, char *vaddr, size_t *length);
int ds_delete_mem, (const char *ds_name);
```

- Data transfer. `Hello` only demonstrates transfer out of the device, but transfer in is analogous. The functions you're interested in are `sys_safecopyfrom()` and `sys_safecopyto()` to copy from and to another process respectively.

  As seen in the hello driver, the opcode for reading is `DEV_GATHER_S`. The opcode for writing is `DEV_SCATTER_S`.

  Because this is a character device, feel free to ignore the `position` parameter. `/dev/Secret` isn't seekable and the reader/writer gets whatever's next.

  Do be careful not to allow a process to write beyond the end of the secret buffer, nor to read beyond what has been written. Be aware that a process may read or write many times so you will have to keep track of where the last read or write occurred.

- `xxx_prepare()`: This reports the geometry of the device back to the filesystem. They're 64-bit numbers set as .lo and .hi. Nobody's going to use it anyhow.

- adding a functional `ioctl()`: The `ioctl` callback has the same prototype as `xxx_open()` or `xxx_close()`. The request is in the `REQUEST` field of the message, and the parameter's location is encoded in the `IO_GRANT` field. You can get the parameter with;

```
uid_t grantee;   /* the uid of the new owner of the secret */
res = sys_safecopyfrom(m->IO_ENDPT, (vir_bytes)m->IO_GRANT,
                       0, (vir_bytes)&grantee, sizeof(grantee), D);
```

- The flags given to `open()` are passed along in the DEV_OPEN message in the `COUNT` field. These flags are not the same as the ones defined in `fcntl.h`. They have been re-mapped by the filesystem to be the same as the bits used in the file permissions mode. These values are defined in `<minix/const.h>`:

```
#define R_BIT 0000004 /* Rwx protection bit */
#define W_BIT 0000002 /* rWx protection bit */
```

  This means that our usual flag sets will have the following values:

`O_WRONLY`	2
`O_RDONLY`	4
`O_RDWR`	6

  Of course there may be other flags as well. This is a bitfield that encodes all the flags passed to `open(2)`.

- To determine the owner of the process calling `open(2)` (the only place you care about ownership) you can use `getnucred(2)` to populate a `struct ucred`, defined in `include/sys/ucred.h` to be:

```
struct ucred {
    pid_t pid;
    uid_t uid;
    gid_t gid;
};
```

- Note: Nothing says that the secret is a string. Beware any of libc's string functions. They may not do what you want.

- Some possibly useful man pages are include in figure 1.

usage(8)	MINIX configuration and usage guide. This is also included on the CD-ROM in `MINIX/INSTALL.TXT` so you can read the installation instructions *before* installing the system.
monitor(8)	describes the MINIX boot monitor process
boot(8)	describes the MINIX boot procedure
init(8)	describes how programs get started. Esp. about `/etc/rc` and `/usr/etc/rc.local`.
service(8)	interface to the reincarnation server for starting and stopping system services.
mknod(8)	how to create a device special file.
getnucred(2)	determine credentials from an `endpoint_t`

Figure 1: Possibly useful man pages

While you're doing this, remember that each device driver is providing the back side of the IO system calls. That is, when you get puzzled about what you should be responding, think about what the caller will expect. (e.g. what do you expect `read()` and `write()` to return to you?)

Note: Device tasks are *below* the filesystem. This means that making normal filesystem calls would be distinctly weird. There is a driver library that provides rudimentary IO services, including a `printf(3)` that writes to the console (and to `/usr/log/messages`). That said, be aware that there is not a `fprintf(3)` in that library. Attempts to use `fprintf(3)` will result in a message being sent to the filesystem which will report a "Strange reply from..." error message. This is not what you want.

What to turn in

The deliverable for this assignment includes the source for your Secret Keeper driver, and a complete report—in the style of the labs—describing the implementation of the device under MINIX.

The report should include:

- An overall architecture of the driver: How is it expected to function?

- A detailed description of the driver implementation including:

- Your development environment: version of the MINIX kernel and type of installation (platform, native vs. bochs, qemu, or VMware).
- A list of all files modified, and the modifications (along with why such modification was necessary),
- The complete code of your driver.

- A description of the driver's behavior when running in the system. If possible, include a typescript or screenshot[3] of the driver in action. (The screenshot is easy if you are running a simulator, not easy if you are running a native installation.)
- A section listing problems encountered, solutions attempted, results obtained, and lessons learned as previous labs.
- Other things as necessary: Remember, the meta-goal here is to convince the reader of the report that you successfully implemented a Secret Keeper device, or, failing that, to convey what you did do and that you learned something useful.

What to turn in (and when to turn it in)

Since this is a weird quarter, both the report and the driver source are to be submitted via `handin` to the `asgn4` directory of the `pn-cs453` account by the deadline.

Sample Runs

Below are a number of sample runs using hte device. Notice the interaction between the two users (root and pnico) as well as what happens when the device fills up.

```
root# mknod /dev/Secret c 20 0
root# chmod 666 /dev/Secret
root# ls -l /dev/Secret
crw-rw-rw- 1 root  operator  20,   0 Nov  1 17:54 /dev/Secret
root# service up `pwd`/secretsafe -dev /dev/Secret
root# cat /dev/Secret
root# echo "The British are coming" > /dev/Secret
root# echo "Another secret" > /dev/Secret
cannot create /dev/Secret: No space left on device
root# cat /dev/Secret
The British are coming
root# cat /dev/Secret
root# echo "This secret is just for me" > /dev/Secret
root# su pnico
pnico$ cat /dev/Secret
cat: /dev/Secret: Permission denied
pnico$ cat > /dev/Secret
cannot create /dev/Secret: No space left on device
pnico$ exit
root# cat /dev/Secret
```

[3] If you do manage to include a screenshot, for goodness' sake be kind to your printer and video-reverse it so it's black on white. (e.g., `mogrify -negate pic.jpg`).

```
This secret is just for me
root# su pnico
pnico$ echo "It's all mine now" > /dev/Secret
pnico$ exit
root# cat /dev/Secret
cat: /dev/Secret: Permission denied
root# su pnico
pnico$ cat /dev/Secret
It's all mine now
pnico$ exit
root# ls -l mys.c
-rw------- 1 pnico  100  7359 Nov  1 19:16 mys.c
root# cat mys.c > /dev/Secret
root# cat /dev/Secret > a
root# diff a mys.c
root# ls -l BigFile
-rw------- 1 root  operator  12102 Nov  1 19:35 BigFile
root# cat BigFile > /dev/Secret
cat: standard output: No space left on device
root# cat /dev/Secret > out
root# ls -l out
-rw-r--r-- 1 root  operator  8192 Nov  1 19:36 out
root# service down secretsafe
```

Since you can only do the `ioctl()` call while holding an open file descriptor, granting another user access requires a program. `a.out` contains the fragment of code shown in Figure 2.

```
fd = open(FILENAME, 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);
}
res=close(fd);
```

Figure 2: Entering a secret and granting ownership to someone else

```
root# ./a.out 13
Opening... fd=4
Writing... res=13
Trying to change owner to 13...res=0
Closing... res=0
root# cat /dev/Secret
```

```
cat: /dev/Secret: Permission denied
root# su pnico
pnico$ cat /dev/Secret
Hello, world
pnico$ exit
```