Operating Systems (234123) - Spring 2016
Home Assignment 4 - Wet

Due Date:	Tuesday, 21.6.2016, 12:30p.m.
Teaching assistant in charge:	Matthias Bonne

Postponements can be authorized **only** by Arthur, the TA in charge. If you need a postponement, please contact him directly.

Important: The Q&A for the exercise will take place at a public forum Piazza only. Critical updates about the HW will be published in pinned notes in the Piazza forum. These notes are mandatory and it is your responsibility to be updated. A number of guidelines to use the forum:

- Read previous Q&A carefully before asking the question; repeated questions will probably go without answers.
- Be polite, remember that course staff does this as a service for the students.
- You are not allowed to post any kind of solution and/or source code in the forum as a hint for other students; in case you feel that you have to discuss such a matter, please come to the reception hour.
- When posting questions regarding hw4, put them in the hw4 folder.

Introduction

In this assignment you will implement a simplified version of `/dev/random`. This is a character special file that provides user-space applications with cryptographically secure random numbers. It is used for various purposes, such as key generation, creating one-time pads, and others.

Note that the simplified version presented here has several flaws which make it unsuitable for cryptographic use. However, the driver you will implement can quite easily be extended in order to fix these problems, and become useful for user-space applications which require high-quality random numbers.

Basic Concepts

Your driver will use an array of a fixed size (512 bytes). This array is called the "entropy pool" and is used to hold random data. When the driver is loaded, the entropy pool should be initialized to all-zeroes[1].

We use the term "entropy count" to denote the number of "unknown", ie. random bits in the entropy pool. Initially the entropy count is zero, since the contents of the entropy pool is completely predicatble (all-zeroes). When nothing is known about the contents of the entropy pool, the entropy count is $512 \cdot 8 = 4096$. When approximately half of the contents is known, the entropy count is 2048. And so on.

Adding data to the pool is done by calling the following function:

```
void mix (const void *in, size_t nbytes, void *pooldata);
```

This function can be found in the file `mix.c` which is attached to this assignment. The purpose of this function is to distribute the new data evenly across the entropy pool.

Reading data from the pool is done by taking a `SHA-1` hash[2] over the contents of the entropy pool. This is done by calling:

```
void hash_pool (const void *pooldata, void *out);
```

Where `out` is a buffer of 20 bytes. This function hashes the data in the entropy pool and returns a 20-byte hash value in `out`. It is defined in the file `sha1.c`, which is attached to this assignment.

Technical Information

Your task is to write a kernel module named "srandom". Your module will support a character device with major number 62 and minor number 0. `open()`ing and `close()`ing the device has no effect on the state of the pool. Other than that, the device should support the following operations.

write()

Writing a buffer `buf` of size `n` to the device is done as follows:

- Divide `buf` into 64-byte chunks.
- For each chunk, call:

  ```
  mix (chunk, chunksize, pooldata);
  ```

  Where `chunk` is the address of the chunk, `chunksize` is the size of the chunk, and `pooldata` is the address of the entropy pool. Note that `chunksize` may be less than 64 if `n` is not a multiple of 64.

If some of the buffer cannot be read (`copy_from_user()` returns nonzero), `write()` should return `-EFAULT`. Otherwise, it should return `n`.

Note that `write()` does not change the entropy count (why?).

[1] Note that this is only for testing purposes and should not be done in a real random number generator (why?).

[2] `SHA-1` is a standard cryptographic hash function. For this driver, its most important property is that it is considered infeasible to derive information on its input from its output. Thus, returning a hash of the contents of the pool does not reveal any useful information on its internal state. You can read more about cryptographic hash functions and their applications on Wikipedia.

read()

Reading a buffer `buf` of size `n` is done as follows:

- If `n` is zero, return 0 immediately.
- Wait until the entropy count is at least 8 (so that we have at least one random byte to return).
- Let $E = \left\lfloor \frac{\text{entropy count}}{8} \right\rfloor$. Note that E is now at least 1.
- If `n` $> E$, set `n` to E. In the next steps, we consider only the first `n` bytes of the given buffer and assume its size is `n`.
- Subtract $8 \cdot$ `n` from the entropy count.
- Divide `buf` into 20-byte chunks.
- For each chunk do:
  - Call:

    ```
    hash_pool (pooldata, tmp);
    ```

    Where `pooldata` is the address of the entropy pool and `tmp` is a temporary buffer. Note that `tmp` must be at least 20 bytes long.
  - Call:

    ```
    mix (tmp, 20, pooldata);
    ```

    This modifies the contents of the pool so that the next chunk will return different data.
  - Copy `tmp` to the current chunk. If the size of the current chunk is less than 20 bytes, copy the first bytes of `tmp` only.

If a signal is received while waiting in the second step above, your `read()` method should return `-ERESTARTSYS`. Note that this is an interval value used by the kernel; in user-space you will see a return value of `-EINTR` in this case.

If any part of the user-space buffer cannot be written to (`copy_to_user()` returns nonzero), `read()` returns `-EFAULT`. Otherwise, `read()` returns the number of bytes actually read. If this is less than the original size of the buffer, the contents of the last part of the buffer are undefined[3]. If an error is returned, the contents of the whole buffer are undefined.

ioctl()

The device should support the following `ioctl` commands. Use `#include <linux/random.h>` to get their definitions. Calling `ioctl()` with any other command should return `-EINVAL`. The name and type of the optional argument follows each command in parenthesis. `void` means no argument is used.

`RNDGETENTCNT (int *p)`

> Copy the current entropy count to `*p`. If `*p` is not a valid user-space address, return `-EFAULT`. Otherwise, return 0.

[3] "undefined" in this context means there are no requirements on what you return in that part of the buffer.

`RNDCLEARPOOL (void)`

Set the entropy count to zero.
This operation requires privilege. To check whether the calling process is allowed to do this, call `capable(CAP_SYS_ADMIN)`. If it returns 0, the operation should fail with `-EPERM`. Otherwise, the entropy count is set to zero and 0 is returned.

`RNDADDENTROPY (struct rand_pool_info *p)`

This command is used to add random data to the entropy pool and increase the entropy count accordingly. The structure `struct rand_pool_info` is defined in the aforementioned header file, `<linux/random.h>`.
This operation requires privilege. You should do the same test described above for `RNDCLEARPOOL`, and return `-EPERM` if it fails. Otherwise, this command is similar to `write(p->buf, p->buf_size)`, with the following additional actions:

- Add `p->entropy_count` to the current entropy count.
- If the new value exceeds 4096, set the entropy count to 4096.

If any part of the structure cannot be read (including `p->buf`), `-EFAULT` is returned. If `p->entropy_count` is negative, `-EINVAL` is returned. Otherwise zero is returned.

Notes

- In order to implement the semantics of `read()` you need to define a waitqueue:

  ```
  DECLARE_WAIT_QUEUE_HEAD(my_waitqueue);
  ```

  where `my_waitqueue` is the name of your waitqueue. Then:
  - call `wait_event_interruptible(my_waitqueue, cond)` to wait for the condition `cond` to become true.
  - when the respective condition becomes true, call `wake_up_interruptible(&my_waitqueue)` to wake up possible waiters.
- To check if there are signals pending for the current process, call `signal_pending(current)`. This function returns non-zero if (and only if) there are pending signals for the current process. You should always check for pending signals after `wait_event_interruptible()` returns (see `read()` above).
- Use `get_user()`, `put_user()`, `copy_from_user()` and `copy_to_user()` to read./write data from/to user-space addresses.
- You may assume your module will run on a single-core machine. Therefore, no form of kernel locking is required.
- You may want to run some tests without root privileges, for example to see whether the permission checks for `ioctl()` work as expected. To temporarily drop root privileges, call `seteuid(n)`, where `n` is nonzero. To regain root privileges, call `seteuid(0)`.

  Alternatively, you can create a new user account by running the following commands:

  ```
  groupadd username
  useradd -s /bin/bash -g username -m username
  ```

  where *username* is the name of the new user. Then, copy your test program to the new user's home directory (`/home/`*username*`/`) and run it with the following command:

  ```
  su - username -c ./mytest
  ```

Submission

You should submit a `zip` file which contains the following files:

- The source files of your module.
- A Makefile that compiles your module. Running "make" should create a module object file called "`srandom.o`".
- A file named `submitters.txt` which includes the ID, name and email of the participating students. The following format should be used:

  ```
  Linus Torvalds linus@gmail.com 234567890
  Steve Jobs jobs@os_is_best.com 345678901
  ```

Important Note: Make the outlined `zip` structure exactly. In particular, the `zip` should contain only the following files (no directories):

```
zipfile
|
+--- your source file(s)
+--- Makefile
+--- submitters.txt
```

Good Luck!
The Course Staff