Lab 2

Quick links: Useful kernel functions / Getting kernel source
Download skeleton code at CourseWeb [http://courseweb.seas.ucla.edu/classAssignment.php?term=11W&srs=187336200]

Overview

In Lab 2, you'll write a Linux *kernel module [http://en.wikipedia.org/wiki/Kernel_module]* that implements a **ramdisk**: an in-memory block device. (A block device is basically a device that acts like a disk, supporting *random-access* read and write operations of fixed-sized units called *sectors*. Random-access means the disk can read or write sectors in any order -- unlike a pipe, for example.)

Your ramdisk will support the usual **read** and **write** operations. You can read and write the ramdisk by reading and writing a file, for instance one named `/dev/osprda`. (You could also initialize your ramdisk to contain a Linux file system!) Your ramdisk will also support a special **read/write locking** feature, where a process can gain exclusive access to the ramdisk. This, of course, leads to some interesting synchronization issues.

Like most device drivers, your module can support several of these devices simultaneously; in particular, it supports four simultaneous ramdisks.

This lab has a handful of purposes, including:

- To introduce you to kernel programming.
- To give you experience in writing a device driver for a real operating system. (Although you don't have to implement any interaction with real hardware controllers.)
- To give you some interesting synchronization problems to handle.

This lab *requires* that you use the QEMU virtual machine environment running a special Linux 2.6.18 image. (While you can do the lab on a real Linux 2.6.18 computer, we don't recommend it.) The SEASnet Linux servers, Linux lab, and CS 111 Ubuntu distribution have been set up with QEMU.

Lab materials

The skeleton code you will get for this lab consists of:

- a `Makefile`
- two module source files (`osprd.c` and `osprd.h`)
- the source code for a utility to read and write a ramdisk (optionally locked) (`osprdaccess.c`)
- a header file implementing the instrumented locking code described below (`spinlock.h`)
- a script for running in Qemu on Linux (`run-qemu`) and a helper script that is run in the Qemu environment (`setup-in-qemu`)
- a simple tester for some (NOT ALL) Lab 2 functionality (`lab2-tester.pl`)

When you extract `lab2.tar.gz` you will get a directory `lab2` which contains these files. Since you will be using Linux for this lab, the command `tar xvzf lab2.tar.gz` will suffice to extract the archive.

Writing kernel code

Writing kernel code isn't actually that scary. The Linux kernel is a very large piece of software, but you only have to use a small subset for your module. Remember that the kernel is providing the abstractions we usually take for granted in userspace, so often it cannot take advantage of them itself (sort of a chicken-and-egg thing).

Each process in Linux has its own *kernel stack*: While the kernel is executing on behalf of a process, it uses a stack which is specific to that process. This is convenient, because kernel code running on behalf of a process can "block" similar to user code. (Contrast this with WeensyOS 1, the MiniprocOS, where processes could not block in the kernel, and there was only one kernel stack.) Linux kernel code must be prepared to handle interrupts and symmetric multiprocessors without breaking. Opportunities for race conditions are rampant. It is very important in the kernel to use proper synchronization mechanisms like locks to avoid unpredictable behavior and deadlocks. Otherwise the whole system can crash!

In the lab skeleton code, we have instrumented the kernel's locks to help you detect incorrect lock usage even on machines where incorrect use of the locks would go undetected. However, you won't be able to detect missing locks, so you'll have to think carefully about where they will be necessary.

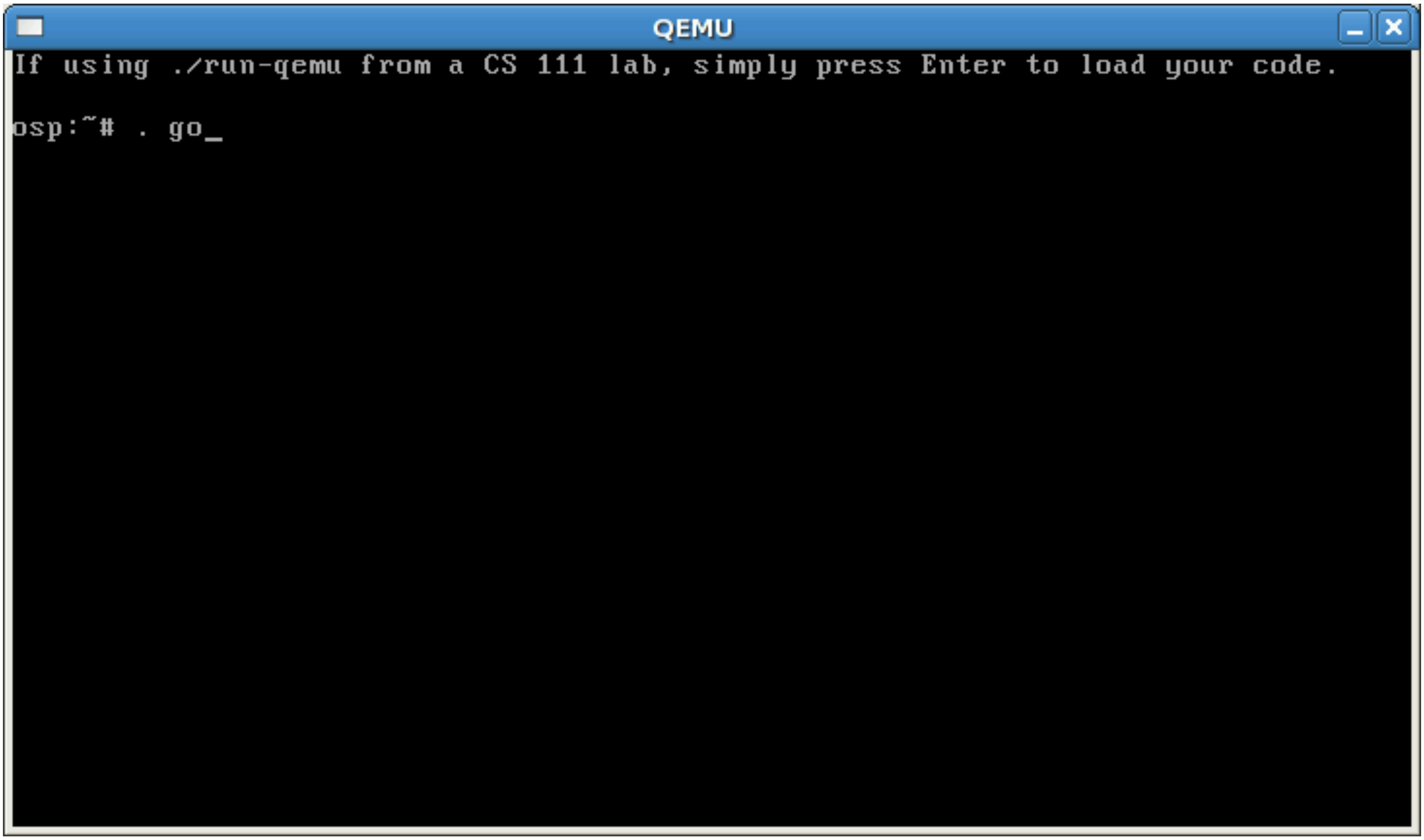
Memory protection is also not the same as it is in userspace. In userspace, if you follow a bad pointer, your program may be killed, but that's about it. In the kernel, the kernel itself will likely crash. Even worse, userspace programs can pass invalid pointers into the kernel through system calls. If the kernel dereferences such a pointer, the machine is in trouble! To help the kernel protect itself against such abuses, there are special kernel functions for dealing with userspace-supplied pointers in a safe way. You won't need to worry about these functions for this lab, but they will become important in Lab 3. You will find them, and other useful kernel functions, documented here.

Finally, you can't call system calls like `open`, `read`, or `write` from inside the kernel. You are *implementing* these functions for your block device; only user code can call them.

Running the Lab

The hardest thing about real-life kernel programming is that bugs have very serious consequences -- often a reboot. Virtual machines and emulators can make this much much easier: a bug will stop the emulator, but not crash the whole computer.

We expect you to complete this lab using a special Linux image we've setup for the QEMU emulator. The lab skeleton code comes with a helpful script named `run-qemu` that will help you get started. On the Linux lab machines and on SEASnet's Linux servers (or your own Linux machine, once you've set it up according to our instructions), simply type `./run-qemu`. This will start the emulator and copy your current directory into the emulated environment. When the emulator comes up, it will say:



```
If using ./run-qemu from a CS 111 lab, simply press Enter to load your code.

osp:~# . go_
```

Simply press Enter. This will copy your lab2 directory into the emulator, unpack it, compile the module, and install it. Then you can test it, using `./osprdaccess` commands as described below. To run the `lab2-tester.pl` script we provide (which tests some, but not all, functionality), run

```
./lab2-tester.pl
```

(or, equivalently, `make check`). To run a particular test case (rather than all of the test cases), give the test case number:

```
./lab2-tester.pl 15
```

Your task

To complete the lab you need to complete all `EXERCISE` sections in `osprd.c`. This section of the lab manual gives an overview of the exercises and related background knowledge.

Your ramdisk will be a *block device*, meaning it will implement the Linux block device interface. Concretely:

- Block devices **store data**. They support read and write operations.
- That data is stored in **fixed-size units**, called **sectors**, which the system can access in any order.

Even though the ramdisk could be more flexible, it conforms to the block device interface so that it can be used everywhere a physical block device can be used. The Linux block device interface also includes an `ioctl` function to perform special per-device operations. You will use the `ioctl` interface to implement the locking features of your block device.

Reading and writing

We suggest you first implement read and write support in your block device. This is the `osprd_process_request` function. When you're done, you'll be able to read and write data from each ramdisk by reading and writing the `/dev/osprdX` files! To make this easier, we've provided a program called `osprdaccess` that can read or write the block device.

To write data into the block device, use a command like

```
echo DataForRamDisk | ./osprdaccess -w
```

The `osprdaccess -w` command reads data from standard input and writes it to the ramdisk. It writes to `/dev/osprda` by default, but you can also supply a ramdisk name:

```
echo DataForRamDisk | ./osprdaccess -w /dev/osprdb
```

You can also write the contents of a file into the ramdisk.

```
./osprdaccess -w < whatever.txt
```

You can limit the amount of data written; this writes at most 128 bytes:

```
./osprdaccess -w 128 < whatever.txt
```

And you can write starting at an offset into the file. This writes the third sector (bytes 1024-1535).

```
./osprdaccess -o 1024 -w 512 < whatever.txt
```

The `./osprdaccess -r` command reads data from the ramdisk and writes it to standard output. It takes very similar arguments. For instance, this command reads bytes 12-15 of `/dev/osprdc` and prints them to standard output.

```
./osprdaccess -o 12 -r 4 /dev/osprdc
```

Here are some more examples.

```
% ./osprdaccess -r 512 | hexdump -C
00000000  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000200
% ./osprdaccess -w 512 < osprd.c
% ./osprdaccess -r 512 | hexdump -C
.......
00000000  23 69 6e 63 6c 75 64 65  20 3c 6c 69 6e 75 78 2f  |#include <linux/|
00000010  76 65 72 73 69 6f 6e 2e  68 3e 0a 23 69 6e 63 6c  |version.h>.#incl|
.......
```

Locking

Next, you will implement support for locking. Once a user is granted a read lock. A write lock guarantees exclusive access to the ramdisk -- no other readers or writers have access to the ramdisk. We've designed the interface as follows. Assume that process P has a file descriptor `fd` that points to a ramdisk.

- `ioctl(fd, OSPRDIOCACQUIRE);`
  - This system call acquires a lock on the ramdisk file. If `fd` was opened for reading (`O_RDONLY`), then the system call requests a read lock. If `fd` was opened for writing (`O_WRONLY` or `O_RDWR`), then the system call requests a write lock.
- `ioctl(fd, OSPRDIOCRELEASE);`

    - This system call releases any lock on the ramdisk file.
    - In addition, closing `fd` will release any lock on the ramdisk file.
- `ioctl(fd, OSPRDIOCTRYACQUIRE);`
    - This system call *attempts* to acquire a lock on the ramdisk file. This is just like OSPRDIOCACQUIRE, except if the acquire operation would block. `OSPRDIOCTRYACQUIRE` never blocks; it returns the `EBUSY` error instead.

The interface works as follows.

- A request for a read lock on a ramdisk file will block until no other files on that ramdisk have a write lock.
- A request for a write lock on a ramdisk file will block until no other files on that ramdisk have a read **or** write lock.
- Lock requests are served in the order in which they are received. Thus, for example, if P1 requests the write lock, and then P2 requests the read lock for the same ramdisk, then P1's request is served first and P2's request will be blocked until P1 has released the write lock.

To implement these semantics, you will need to maintain a wait queue per ramdisk, so that processes that are blocked can be placed on the wait queue. You also keep track of how many files are open for a given ramdisk for reading and for writing.

The `osprdaccess` program's `-l` option will open the ramdisk and then request a lock before performing the read or write. The `-l` flag optionally takes a parameter that indicates a delay to wait after opening and before locking the disk. The `-d` option allows you to specify a delay to wait after opening (and possibly locking) the disk, but before performing the read or write and closing the disk. The `-L` option is like `-l`, but uses TRYACQUIRE instead of ACQUIRE. You can use these functions to test the locking feature.

```
    # First, zero out the ramdisk.  We do this after each set of commands.
% ./osprdaccess -w -z
```

```
    # This command writes "foo\n" into the first four bytes of the ramdisk,
    # as we can see when we read those bytes back out.
% echo foo | ./osprdaccess -w
% ./osprdaccess -r 4
foo
```

```
    # This command has the same effect, but the "-d 5" option tells the first
    # osprdaccess to wait 5 seconds after opening the ramdisk, but before
    # writing.
% echo foo | ./osprdaccess -w -d 5
[[WAITS 5 SECONDS]]
% ./osprdaccess -r 4
foo
```

```
    # Now let's play with some locks!  Run the first osprdaccess
    # in the background, then try to run the second.  The second should wait
    # for the first to complete.  It is easiest to get this to work if you "tee up"
    # both commands in adjacent terminals, then press return on the two
    # terminals in the correct order.  Or you can make the delay bigger so you
    # have time to type (or paste) the second command.
% echo foo | ./osprdaccess -w -l -d 5 &
[1]
% ./osprdaccess -r 4 -l
[[WAITS 5 SECONDS]]
foo
```

```
    # If we don't use locking, the read and write proceed in parallel.
% ./osprdaccess -w -z
% echo foo | ./osprdaccess -w -d 5 &
% ./osprdaccess -r 4
[[returns four 0 characters right away]]
```

```
    # Note that two locking reads can proceed in parallel.
% echo foo | ./osprdaccess -w
% ./osprdaccess -r 4 -l -d 5 &
[1]
% ./osprdaccess -r 4 -l
foo      [[returns right away, then after 5 seconds:]]
foo
```

```
    # But a locking read will not allow a locking write.
% ./osprdaccess -r 4 -l -d 5 &
[1]
% echo bar | ./osprdaccess -w -l
[[WAITS 5 SECONDS]]
```

```
   # As we can see when we use TRYACQUIRE.
% ./osprdaccess -r 4 -l -d 5 &
[1]
% echo bar | ./osprdaccess -w -L
ioctl OSPRDIOCTRYACQUIRE: Device or resource busy

   # Locking the same ramdisk twice would cause deadlock!
% echo foo | ./osprdaccess -w -l /dev/osprda /dev/osprda
ioctl OSPRDIOCACQUIRE: Resource deadlock avoided
```

The examples above are not exhaustive. Make sure to come up with your own examples to thoroughly test your locking implementation.

Make sure you protect modifications to shared state with the `osprd_info_t`'s `mutex` member. This is a *spin lock*, or polling mutex. Remember that it is illegal in Linux for a process to block while holding a spin lock! (This is because another thread could then attempt to lock the spin lock, and spin forever.) You will have to arrange your code in later steps so that you always release the spin lock before blocking and re-acquire it afterwards.

You can either implement locking support all at once, or you can go more gradually. To go gradually, first change osprd_ioctl to implement mutual exclusion: make sure that lock requests don't complete until the conditions described in the locking rules are met. You can do this without blocking. Instead, call schedule() if a lock request cannot currently complete and then re-check the condition when you wake up (when schedule returns). Use the `ticket` members of `osprd_info_t` to serve lock requests in FIFO order. (`schedule` and other important kernel functions are defined here. **MAKE SURE YOU READ THAT PAGE!** The `ticket_head` and `ticket_tail` members are effectively an eventcount and a sequencer, respectively, as defined in your textbook, section 5.6.)

Second, support *blocking*, using the `wait_queue` functions. To implement blocking, you need code to begin blocking in the OSPRDIOCACQUIRE case of osprd_ioctl (this is a trivial change if you use `wait_event_interruptible`), and you need code to wake up blocking processes in osprd_release and in the OSPRDIOCRELEASE case of osprd_ioctl.

Signals are a tricky aspect of this interface. Remember that signals are a way to send a running process asynchronous events, or to kill a running process. If the process is blocked in the kernel -- for instance, if it is blocked in `osprd_ioctl` -- then the kernel must wake up so the process can process the signal. Since we want to kill blocked processes, they had better be able to handle signals. This means:

- When blocked, processes should use an *interruptible wait*. This means that signal delivery takes place as usual. (In an uninterruptable wait, you would not be able to kill a waiting processes except by causing the event it was waiting for -- and this would, in turn, prevent your module from being unloaded because it is currently blocked by code in your module. You'd have to reboot a lot!)
- When your process receives a signal, it will wake up. If a signal was received (see the `signal_pending` function, or `-ERESTARTSYS` return value), `osprd_ioctl` should **immediately return** to its caller.

Test your code using test cases like those above (**AND OTHERS you come up with yourself**).

Deadlock

So far, so good! But there's a problem: An ACQUIRE operation might cause a deadlock. (Note that TRYACQUIRE could never cause deadlock -- why not?) Your job is to detect these deadlocks. Instead of blocking forever (or until a signal), your code should return `-EDEADLK`, a special error code that indicates a deadlock was avoided.

One easy case of deadlock is indicated above. For full credit, you should detect **any** possible deadlock condition, but we will put most grading weight on simple deadlock cases where a process is waiting directly for itself.

The `lab2-tester.pl` **DOES NOT TEST DEADLOCK AVOIDANCE**. Come up with your own test cases.

Compiling your module for real

This section applies **only** if you are installing your module for real, in a running kernel. You might enjoy reading it over anyway, so you see how modules are loaded and unloaded.

On the lab machines in 4405 Boelter, we have already installed the Linux kernel source code necessary for building kernel modules for the revision of the kernel they are running. All you have to do to compile your code into a loadable kernel module (a `.ko` file) is run `make`. However, if you are using your own machine, you may need to get the kernel source code first. Try compiling the skeleton module; if it does not work, go here for instructions on how to download and install the kernel source code.

Loading and unloading your module

Ordinarily, loading or unloading Linux kernel modules requires that you be the administrator (the `root` user). If you are using your own machine, you can just change to the root user with `su`.

After changing to the `root` user, you can use the `insmod` program to load `.ko` files: just run `insmod` with the name of the `.ko` file to load as an argument. The name of the module is the part before the `.ko`, thus, for this lab, the module will be named `osprd` and the module file will be `osprd.ko`. To unload a module, use the `rmmod` command with the name of the module as an argument. Note that when you try to load a module, if there is already a module by the same name loaded, it will fail -- so you will have to unload it first.

Once your module is loaded, it becomes a part of the currently executing kernel. Your code executes as part of the kernel, and can easily crash the machine if you dereference bad pointers or do other things that would only kill your process in userspace. **Save your files and run `sync` before you load your module to avoid losing your work!** Even after unloading your module, there may be residual effects of its having been loaded if there were serious bugs in it, so you should save your work frequently.

Getting debugging output

Messages you print with `printk` or `eprintk` from your module may not be shown on the screen by default, which can make debugging a little difficult. If you find that this is the case, you can get a list of the messages by running the `dmesg` program. On the lab machines, you may need to run it via `sudo` as described above for `insmod` and `rmmod`; if so, you may also need to specify its full path: `/bin/dmesg`. We found that `eprintk` messages were actually sent to all open terminals on the lab machines, so you may want to use `printk` and `dmesg` instead.

Using your module

After loading your module, you'll probably want to try using it. To do this, you will have to access the *block special* *files* in `/dev` which represent the devices it controls. If you are using your own machine, these files may not exist yet (depending on a variety of things). Look in `/dev` to see if you have files named `osprda`, `osprdb`, `osprdc`, and `osprdd`. If so, you're all set. If not, run the `create-devs` script as `root` to create them.

These files are not like other files in your filesystem. They do not actually contain any data; they just "represent" the devices your driver controls. The only reason they are there is so programs can call `open` on them to get file descriptors which the kernel knows are connected to particular devices. However, this property means that many programs can use them just as if they were normal files. This is part of the UNIX design philosophy that "everything is a file."

So, to use your devices and thus your module, you just need a program which will open the block special files and read from or write to them. We have included the `osprdaccess.c` code for this purpose, but you can also write your own program to do it. If you write your own program, you may want to look into the `read` (`man 2 read`) and `write` (`man 2 write`) system calls.

Design problems

Design problems info

Disk Emulation

Since ramdisks are just blocks of memory, they behave much nicer than physical disks in lots of ways. Real disks have advantages (persistence, cheap per-byte cost), but they are also **slow**, disk requests can cause **contention** (multiple outstanding disk requests are often executed in some order, rather than in parallel), and disks tend to **fail** (I/O errors, bad sectors).

In this design problem you will design support for **emulating real disk characteristics on your ramdisk**. Your ramdisk should be able to run slowly (requests will block for some time before they complete), and/or report errors (where ramdisk sectors "go bad": all writes and reads to a bad sector should fail with an `EIO` error), and/or support for other characteristics of a real disk as you see fit.

This might seem like the least useful thing ever, but in fact it's quite useful for testing file system implementations. To see whether a file system can handle disk errors, a programmer might put a file system image on an error-simulating ramdisk and see how it behaves. This is a lot more convenient (and cheaper) compared to putting the file system on a real hard drive and shaking it until an error occurs! Similarly, you can test how a file system behaves for different disk speeds.

Implementing this design problem will require more hacking into the `osprd_process_request` function, plus designing a way to specify disk characteristics (speed/errors -- the `ioctl` function may be useful here).

Change Notification

Design a mechanism by which a process can request notification whenever data on a ramdisk changes. Specifically, a process P1 can open a ramdisk and request notification on any change. When another process writes to the ramdisk, P1 should receive notification.

We are being purposefully vague about how "notification" should be implemented: it is your job to decide. But a waiting process P1 should be able to block on something -- such as a system call like `ioctl` -- until the notification arrives.

The waiting process should be able to wait for changes to specific areas of the disk. For instance, maybe there will be separate notifications for each sector of the disk, or for ranges of sectors, all the way down to individual bytes. Again, you should decide what type of notification to implement, and justify your choice. Think about time and space overheads.

Encrypted Ramdisk

Recent operating systems have allowed users to store data encrypted on the filesystem. The filesystem is readable and writable only if the user provides the right password when they log in.

Implement an encrypted ramdisk, where data is stored on the ramdisk in encrypted format. If a user opens the ramdisk normally, they should see encrypted gobbledegook. But if a user provides the right password at open time, then `read` operations on that open file should transparently decrypt the disk's data. Furthermore, if the user writes to the file, the data they write should be encrypted before it is sent to the ramdisk.

You will need to implement ramdisk-specific `read` and `write` operations for this challenge problem; we can help you set this up. (You must change the `osprd_blk_fops` structure's `read` and `write` operations to point to your code.) You should decide which forms of encryption to support. One choice might be to support any form of encryption Linux can handle. These are described in `include/linux/crypto.h` and `Documentation/crypto/*`.

Hand-in

When you are finished, edit the file named `LAB` and follow the instructions at the top of the file to fill in your name(s), student ID(s), email address(es), short descriptions of any extra credit or challenge problems you did, any known limitations of your code (including known bugs), and any other information you'd like us to have.

Then run "`make tarball`" which will generate a file `lab2-`*`yourusername`*`.tar.gz` inside the lab2 directory. Upload this file to CourseWeb using a web browser to turn in the project. Remember to upload it only once if you are working in a team – the `LAB` file will allow us to give both team members credit.

Good luck!