Project 3

The swapper module
(debugfs, kobjects, spinlocks, mutexes, and a misc device)

Overview

In this assignment you will combine everything we have covered regarding debugfs, kobjects, spinlocks, mutexes, and misc char devices to write a kernel module called **swapper**.

The swapper module attempts to emulate the behavior of a simple, yet somewhat strange, piece of hardware that supports reading and writing to the multiple drives it manages. These drives are called **swapstores**. Since this drive isn't real, it doesn't have any buttons we can press or slots we can insert swapstores into. So, we are going to use debugfs to emulate all that stuff. You will be able to insert new swapstores, eject swapstores, and set the active swapstore by writing to different files in debugfs.

The swapper module exposes a single misc device at **`/dev/swapper`**. Writing to **`/dev/swapper`** writes to the currently active swapstore, and reading from **/dev/swapper** reads from the currently active swapstore. Again, the currently active swapstore is set through debugfs and swapstores can be inserted and ejected through debugfs. Only one swapstore can be active at a time, so think of the swapper device as being like one of those multi-CD changers you would find in a car.

As you may have guessed, a swapstore is built around a kobject. Each swapstore can store up to **`PAGE_SIZE`** bytes (probably 4096 on your system) and has two (2) attributes of interest that are exposed to userspace: **`readonly`** and **`removable`**.

- The **`readonly`** attribute is read/write, but it can only take on the values of **`0`** or **`1`** – trying to write anything else to this attribute produces **`-EINVAL`**. If set to **`0`**, then the swapstore can be read from and written to; otherwise, if it is set to **`1`** the swapstore can only be read from. Attempting to write to **`/dev/swapper`** when the active swapstore has **`readonly`** set to **`1`** will produce **`-EPERM`**.
- The **`removable`** attribute can only be read. All swapstores that are inserted using debugfs will have this attribute set to **`1`** upon creation. The only swapstore that will have this set to **`0`** is the **`"default"`** swapstore. This is a permanent swapstore that is built into the hardware, so it can't be ejected. It is created when the swapper module is loaded and only gets destroyed when the swapper module is unloaded.

We will look at how the debugfs interface should work briefly. At this point it would be a good idea to watch the **expected behavior video** before continuing further.

This project has several systems that work together to produce the final behavior. I recommend the 3 step approach that follows, but you can attempt to do it in another order if you would like.

Step 1 – Build the swapstore object

A swapstore object looks like this:

```
struct swapstore {
      struct kobject kobj;
      char data[PAGE_SIZE];
      int readonly;
      int removable;
};
```

I recommend first building a **kobj_type** that exposes the members **removable** and **readonly** as attributes to userspace first. These should function as described in the **Overview** on the previous page. There's no need to put these attributes into a named group. Naturally, you will also need to write a **release** function for your **kobj_type**, so don't forget.

All swapstore objects should belong to a **kset** that shows up in userspace at:
/sys/kernel/swapstore/

Create a few swapstores in your init function and make sure you can see them in sysfs under the **kset**. Test them, make sure you can read and write values to them, as appropriate. Remember, you should only be able to read from the **removable** attribute, and writing to it should produce **-EPERM**. Also, make sure that writing anything other than **0** or **1** to the **readonly** attribute produces **-EINVAL** before continuing. Only the **root** user should be able to access these attributes.

All swapstore objects should free themselves when the module unloads. It shouldn't matter how many you have or what their names are. We talked about how to do this in lecture. This is desirable behavior because you will be creating swapstore objects dynamically at runtime in the next step when you add the debugfs interface.

Now would also be a good time to create the **"default"** swapstore described in the **Overview**. Again, this can be thought of as a "built in" swapstore than can't be ejected. It should be created when your module loads and be destroyed only when your module unloads. It should have **removable** set to **0** and **readonly** set to **0** upon creation.

Got all your **kobject** and **kset** stuff solid as a rock? Good. Time to move on to Step 2.

Step 2 – Build the debugfs interface

Your debugfs interface should be located at:
`/sys/kernel/debug/swapper/`

In this directory, you will add three (3) files:

- **`insert`** – (writeable by root only)
  Writing a **name** to this file will result in a swapstore object being created with the specified name. The new swapstore object should have **`removable`** set to **`1`** and **`readonly`** set to **`0`** upon creation. If a swapstore by that name already exists, **`-EINVAL`** should be produced.

- **`swapstore`** – (read/write by root only)
  This one is special. Implement this after Step 3 (but read now!)

  Writing the **name** of a swapstore to this file will result in the corresponding swapstore being "attached" to the misc char device **`/dev/swapper`**. The previously attached swapstore will be detached. Subsequent reads and writes to **`/dev/swapper`** will be stored in this swapstore. However, if the char device **`/dev/swapper`** is currently held open by at least once userspace process, no detachment/attachment should occur and instead **`-EBUSY`** will be produced.

  Reading from this file simply produces the name of the currently active (i.e. "attached") swapstore.

- **`eject`** – (writeable by root only)
  This one is only half-special. Implement parts involving /dev/swapper after Step 3

  Writing a **name** to this file will result in the corresponding swapstore being queued for ejection. If the swapstore is not currently attached to **`/dev/swapper`**, it will be removed from the system immediately, generating a corresponding removal uevent. However, if the swapstore is currently attached to **`/dev/swapper`** it will instead be removed immediately after becoming detached from **`/dev/swapper`**, again generating the appropriate removal uevent. If **name** does not correspond to an existing swapstore, **`-EINVAL`** should be produced.

Once you can insert and eject swapstores by writing their names to the **`insert`** and **`eject`** files, move on to the next step. Be sure to first make sure all the sysfs attributes are still working, though! Always test old functionality after you build something on top of it.

Step 3 – Add the misc device /dev/swapper

This part should seem relatively straightforward at first. All you need to do now is create a misc char device and write its **`struct file_operations`**, right?

Start off by creating the misc device **`/dev/swapper`** and attaching it to the **`"default"`** swapstore. Make sure you can read and write to the **`"default"`** swapstore before continuing. For this assignment, you can zero out the active swapstore just buffer before each write to keep things simple and looking nice.

Working? Great. Now here's the tricky part.

Users are going to try their hardest to break your swapper device. Even the **`root`** user ...well, especially the **`root`** user. Specifically, you need to worry about the active swapstore being ejected while **`/dev/swapper`** is attached to it. This could happen at any time since the **root** user is able to write the name of the active swapstore to **`/sys/kernel/debug/swapper/eject`** whenever they feel like it with reckless disregard. If the root user's request for ejection were granted immediately, any poor process that attempted to write to **`/dev/swapper`** would crash the entire system -- **`/dev/swapper`**'s write fops would try store data into kernel memory it no longer owned. You can't allow this to happen! The active swapstore can't be freed from memory as long as it is attached to **`/dev/swapper`**. Period. However, **root** will expect the specified swapstore to be ejected at some later point in time once it is no longer attached – this should happen without root requesting the eject again.

Additionally, that pesky **root** user can attempt to change the active swapstore by writing to **`/sys/kernel/debug/swapper/swapstore`** whenever they fancy the winds! It would be a huge problem if this was allowed to happen while some userspace process was in the process of reading or writing to the currently attached swapstore. In other words, the active swapstore can't be detached (and a new one reattached) while the **`/dev/swapper`** file is being held open by even just 1 process. So, it will be important for your module to keep track of how many times **`/dev/swapper`** has been opened and closed. Again, you have to be careful here as well. Users are sneeky and out to get you. Many users could be opening/closing **`/dev/swapper`** at the same time, and their processes doing that could all be running on different CPU cores. You'll have to take extra care when updating your count of how many processes currently have **`/dev/swapper`** open!

Keeping these types of issues in mind, it's time to go back and finally implement **`swapstore`** as well as the missing bits of **`eject`** from Step 2. Be careful! Don't forget, the spinlock, mutex, and referencing counting rules, and you'll be alright. I know you can do it.