1 Mutex, semaphore and the proc file system

A device is often accessed by multiple processes, therefore critical section of the code, that intends to access shared data atomically, must be protected by mechanisms such as mutex, semaphore, or spinlocks. "Critical section" is a section of code that can be executed by one process at a time. Mutex and semaphore are discussed in this lab, and spinlock, a lock that busy-waits and expected to be held for only a very short time, is discussed later in this course.

1.1 Mutex

A mutex is a basic kind of sleepable locking mechanism, used for protecting critical section of the code. A process must lock the mutex before entering the critical section and release the mutex after leaving the critical section.

Mutexes are initialized in an unlocked state with:

```
DEFINE_MUTEX(name);
```

at compile time, or:

```
void mutex_init(struct mutex *lock);
```

Locking primitives come with interruptible and uninterruptible forms:

```
void mutex_lock(struct mutex *lock);
int mutex_lock_interruptible(struct mutex *lock);
int mutex_lock_killable(struct mutex *lock);
```

The last two functions return 0 if the lock is acquired.

There is only one unlocking primitive:

```
void mutex_unlock(struct mutex *lock);
```

Any signals will break a lock taken out with mutex_lock_interruptible() while only fatal signals can break a lock taken out with mutex_lock_killable(). Both functions return -EINTR if they are interrupted by signals.

Locks taken out by mutex_lock() are not affected by signals (i.e. unkillable), and in most cases, it is a bad idea to use this unkillable version because the only way to terminate a blocked process is a system reboot.

```
int mutex_trylock(struct mutex *lock);
```

is the non-blocking version of mutex_lock(), which always returns immediately with 1 on success and 0 on contention.

Here are some rules on the use of mutexes:

1. The mutex must be released by the original owner.
2. The mutex cannot be applied recursively (i.e. a process cannot acquire the mutes again without releasing the mutex first.
3. The mutex cannot be locked or unlocked from interrupt context.

For example:

```
DEFINE_MUTEX(my_mutex);

static ssize_t
mycdrv_read(struct file *file, char __user * buf, size_t lbuf, loff_t * ppos)
{
        printk(KERN_INFO "process %i (%s) going to sleep\n", current->pid,
               current->comm);
        if (mutex_lock_interruptible(&my_mutex)) {
                printk(KERN_INFO "process %i woken up by a signal\n",
                       current->pid);
                return -ERESTARTSYS;
        }
        printk(KERN_INFO "process %i (%s) awakening\n", current->pid,
               current->comm);
        return mycdrv_generic_read(file, buf, lbuf, ppos);
}

static ssize_t
mycdrv_write(struct file *file, const char __user * buf, size_t lbuf,
             loff_t * ppos)
{
        int nbytes = mycdrv_generic_write(file, buf, lbuf, ppos);

        printk(KERN_INFO "process %i (%s) awakening the readers...\n",
               current->pid, current->comm);
        mutex_unlock(&my_mutex);
        return nbytes;
}
```

1.1.1 Exercise: Mutex

Write three simple modules where the second and the third one use a variable exported from the first one. The second and third one can be identical, but having different module names.

Hint: You can use the macro __stringify(KBUILD_MODNAME) to print out the module name.

The exported variable should be a mutex. Have the first module initialize it in the unlocked state.

A variable from a module can be exported using the macro:

```
EXPORT_SYMBOL(my_variable);
```

so in module 1, if you want to declare, initialize and export the mutex, you would do this in the global scope of module 1:

```
DEFINE_MUTEX(my_mutex);
EXPORT_SYMBOL(my_mutex);
```

The second and third modules should attempt to lock the mutex during initialization. If the mutex is already locked, the module will not be loaded because the initialization function should return with an appropriate value for the module to be loaded.

Make sure the mutex is released in the cleanup (module_exit) function in the module.

Test by trying to load both modules simultaneously, and see if it is possible. Make sure you can load one of the modules after the other has been unloaded, to make sure you released the mutex properly.

1.2 Semaphore

Semaphores are also used in protecting access to critical sections in kernel code. There are two types of semaphore structures: struct semaphore, and struct rw_semaphore, the latter of which supports single-writer multiple-readers (SWMR) policy.

Semaphores can be initialized with:

```
DEFINE_SEMAPHORE(name);
```

and rw_semaphore can be initialized with:

```
DECLARE_RWSEM(name);
```

Here are the semaphore primitives:

```
#include <linux/semaphore.h>

void down(struct semaphore *sem);
void down_interruptible(struct semaphore *sem);
void down_killable(struct semaphore *sem);
int down_trylock(struct semaphore *sem);        /* non-block */

void up(struct semaphore *sem);
```

Here are the read-write semaphore primitives:

```
#include <linux/rwsem.h>

void down_read(struct rw_semaphore *sem);
void down_write(struct rw_semaphore *sem);
void up_read(struct rw_semaphore *sem);
void up_write(struct rw_semaphore *sem);
```

The down() function checks to see if someone else has already entered the critical code section; if the value of the semaphore is greater than zero, it decrements it and returns. If it is already zero, it will sleep and try again later.

The down_interruptible() function differs in that it can be interrupted by a signal; the other form blocks any signals to the process, and should be used only with great caution. However you will now have to check ti see if a signal arrived if you use this form, so you-ll have code like:

```
if (down_interruptible(&sem)) return -ERESTARTSYS;
```

which tells the system to either retry the system call or return -EINTR to the application.

The down_killable() function can only be interrupted by fatal signals, so it will not be interrupted unless the signal is intended to terminate the program. Therefore in most cases, this version should be used to ensure the caller user-space programme is killable, otherwise the only way to terminate a deadlocked proces is to reboot the system.

The down_trylock() form checks if the semaphore is available, and if not, returns a non-zero value immediately without blocking (which is why it does not need an interruptible form) and if the semaphore is available, return 0. For instance, a typical read entry from a driver may contain:

```
...
if (file->f_flags & O_NONBLOCK) {
  if (down_trylock(&iosem)) return -EAGAIN;
} else {
  if (down_interruptible(&iosem)) return -ERESTARTSYS;
}
```

The up() function increments the semaphore value, waking up any processes waiting on the semaphore. It doesn't require any _interruptible form.

Here is an example showing the use of semaphores in a kernel module:

```
DEFINE_SEMAPHORE(name);

static ssize_t
mycdrv_read(struct file *file, char __user * buf, size_t lbuf, loff_t * ppos)
{
        printk(KERN_INFO "process %i (%s) going to sleep\n", current->pid,
               current->comm);
        if (down_interruptible(&my_sem)) {
                printk(KERN_INFO "process %i woken up by a signal\n",
                       current->pid);
                return -ERESTARTSYS;
        }
        printk(KERN_INFO "process %i (%s) awakening\n", current->pid,
               current->comm);
        return mycdrv_generic_read(file, buf, lbuf, ppos);
}

static ssize_t
mycdrv_write(struct file *file, const char __user * buf, size_t lbuf,
             loff_t * ppos)
{
        int nbytes = mycdrv_generic_write(file, buf, lbuf, ppos);

        printk(KERN_INFO "process %i (%s) awakening the readers...\n",
               current->pid, current->comm);
        up(&my_sem);
        return nbytes;
}
```

1.2.1 Exercise: Semaphore

Replace the mutex in the last lab with semaphores.

Please note that the macro used to statically declare and initialize a semaphore in an unlocked state is:

```
DEFINE_SEMAPHORE(name);
```

1.2.2 Exercise: Device-private data

In struct file, there is a field:

```
void *private_data;
```

which can point to a piece of data that is private to the process that opens the device.

Now you should convert the results of Exercise 3.1.3 to use private data. Each process that opens the device now has its own private ramdisk, so the ramdisk structure should be allocated in open() and pointed to by file→private_data and freed in close().

1.3 /proc entry

/proc is a virtual filesystem allowing devices to display their information. Each device needs to create an entry under /proc to make use of the /proc filesystem to display its information. In addition, writing to a /proc entry can set system parameters and modify device functionality.

1.3.1 Creating entries

Creating, managing and removing entries in the proc filesystem is done with these functions:

```
#include <linux/proc_fs.h>

struct proc_dir_entry *proc_create (const char *name, mode_t mode,
       struct proc_dir_entry *parent, struct proc_ops *fops);

struct proc_dir_entry *proc_create_data (const char *name, mode_t mode,
       struct proc_dir_entry *parent, struct proc_ops *fops, void *data);

void remove_proc_entry (const char *name, struct proc_dir_entry *parent);

struct proc_dir_entry *proc_symlink (const char *name, struct proc_dir_entry *parent,
                                     const char *dest);

struct proc_dir_entry *proc_mkdir (const char *name, struct proc_dir_entry *parent);
```

The **name** argument gives the name of the directory entry, which will be created with the permissions contained in the mode argument. **parent** is the proc entry of the parent subdirectory which the current proc entry resides in. If the parent argument is NULL, the entry will go in the /proc main directory. **mode** sets the permission of the proc entry (please see lab 2 for details on file **mode**).

The function proc_create() creates a proc entry with **name** under **parent** with file operations pointed by **fops**. New Linux kernels above 3.11.x treat all proc entries as normal device files so all functions in **fops** need to be implemented as you implement the functions of a char device, except the proc entries usually have small data sets which are read sequentially. A simple way to implement such proc entries is to use **seq_file** which is implemented in Linux kernel. Details will be given shortly.

The function proc_create_data() uses an additional parameter **data** which allows the file operations in **fops** to access later. You may use the same set of file operations for different proc entries but treat them differently in the file operations according to the different content passed via **data**.

The function proc_symlink() creates a symbolic link; it is equivalent to doing:

```
$ ln -s <dest> <name>
```

The function proc_mkdir() creates directory name under parent.

The parent directory can be something you created with proc_mkdir(), or if you want to put it in an already-created subdirectory of /proc, such as /proc/driver, one can do:

```
my_proc = proc_create("driver/my_proc", 0, NULL, &my_fops);
```

1.3.2 Reading entries

When a process tries to read an entry in the proc filesystem, it causes invocation of the read callback function passed via file operations such as **fops**. It is quite some challenge to get the read function work correctly in all conditions, especially when the data to be read is long.

Linux kernels (above 2.6) contain a set of functions that are meant to make the job of creating virtual files like /proc entries easy. They are collectively called the seq_file interface, which is available by including linux/seq_file.h in your device driver program. The detail of the interface can be found at Documentation/filesystems/seq_file.txt.

The seq_file interface has an iterator that can step through a number of objects you want a process to read.

```
struct seq_operations my_seq_ops = {
    .start = my_seq_start,
    .next = my_seq_next,
    .stop = my_seq_stop,
    .show = my_seq_show
};
```

You need to implement each of the above callback functions just like you implement the file operations. When a proc entry implemented by the seq_file interface is read, .start function will be called, then .show is called, then .next and .show are called repeatedly until .next returns NULL. Finally .stop will be called. To make the read operation to stop, conditions should be set for both .start and .next to return NULL. Below is an example of these functions

```
void *my_seq_start(struct seq_file *s, loff_t *pos)
{
if(*pos >= 1) return NULL;
else return &asgn2_dev_count + *pos;
}

void *my_seq_next(struct seq_file *s, void *v, loff_t *pos)
{
(*pos)++;
if(*pos >= 1) return NULL;
else return &asgn2_dev_count + *pos;
}

void my_seq_stop(struct seq_file *s, void *v)
{
/* There's nothing to do here! */
}

int my_seq_show(struct seq_file *s, void *v) {
 seq_printf(s,
                   "Pages: %i\nMemory: %i\n",
                   asgn2_device.num_pages,
                   asgn2_device.data_size);
  return 0;
}
```

The above example allows the proc entry to read one object which only shows the number of pages and data size. **seq_printf** is part of the seq_file interface which is similar to **printf**. Note that **v** is not used in **my_seq_show** as we only print for one object. For printing messages of multiple objects, you need to use **v** to tell which object you are going to print for. In case you may wonder where **v** is from: **v** is the pointer returned by **.start** or **.next** which should return a different **v** each time they are invoked. For more advanced use, refer to Documentation/filesystems/seq_file.txt and https://github.com/jesstess/ldd4/blob/master/scull/-main.c

To complete the rest of the code using the seq_file interface, see the code below.

```
int my_proc_open(struct inode *inode, struct file *filp)
{
    return seq_open(filp, &my_seq_ops);
}

struct proc_ops asgn2_proc_ops = {
.proc_open = my_proc_open,
.proc_lseek = seq_lseek,
.proc_read = seq_read,
.proc_release = seq_release,
};
```

The code uses most of the file operations provided by seq_file except that .open is instantiated by **my_proc_open** which calls **seq_open** with the defined iterator functions my_seq_ops passed to seq_file.

Finally, to complete the creation of the proc entry, the following function should be called in the module init function:

```
my_proc_entry = proc_create(MYDEV_NAME, 0, NULL, &asgn2_proc_ops);
/* should test if my_proc_entry is NULL here */
```

If you want to create a both readable and writable proc entry, you cannot use the seq_file interface. As mentioned before, you will need to create read/write callback functions in file operations yourself. Below is an example code snippet that handles a limited message buffer of 100 bytes.

```
static char msg[101];
static int dlen=0, wptr=0, rptr=0;

int test_read_proc(struct file *filp,char *buf,size_t count,loff_t *offp )
{
if(rptr == dlen) return 0;
if(count>dlen-rptr) count=dlen-rptr;
copy_to_user(buf+rptr, msg, count);
rptr = rptr + count;
return count;
}

int test_write_proc(struct file *filp,const char *buf,size_t count,loff_t *offp)
{
if(wptr >= 100) return count;
if(count > 100 - wptr) count = 100 - wptr;
copy_from_user(msg+wptr, buf, count);
dlen = dlen + count;
wptr = wptr + count;
return count;
}

int test_open_proc(struct inode *inode, struct file *filp)
{
rptr = 0;
wptr = 0;

  // if not opened in read-only mode, clean message buffer
  if ((filp->f_flags & O_ACCMODE) != O_RDONLY){
    memset(msg, 101, 0);
    dlen = 0;
  }

return 0;
```

```
}

int test_close_proc(struct inode *inode, struct file *filp)
{
  // if not opened in read-only mode, process the message buffer
  if ((filp->f_flags & O_ACCMODE) != O_RDONLY){
    parse_msg(msg);
  }

return 0;
}

struct proc_ops test_proc_ops = {
.proc_open = test_open_proc,
.proc_read = test_read_proc,
.proc_write = test_write_proc,
.proc_release = test_close_proc,
};
```

With the above file operations defined, we can create a proc entry:

```
proc_create("test",0,NULL,&test_proc_ops);
```

Note that usually /proc entries are text, not binary. This means to convert user-space input into usable form, you need to parse the message buffer according to the format of the message defined. The following functions defined in /usr/src/linux/lib/vsprintf.c are useful for the purpose of parsing:

```
long simple_strtol (const char *cp, char **endp, unsigned int base);
unsigned long simple_strtoul (...);
unsigned long long simple_strtoull (...);
long long simple_strtoull (...);
```

all of which have the same arguments. The first argument is a pointer to the string to convert, the second is a pointer to the end of the parsed string, and the third is the number base to use; giving 0 is the same as giving 10. The following statements are equivalent:

```
long j = simple_strtol ("-1000", NULL, 10);
```

```
long j = simple_strtol ("-1000", 0, 0);
```

You can also do the format conversion using the kernel implementation of sscanf()

Here is a code snippet showing how a write function processes the user input.

```
        char *str;
        str = kmalloc((size_t) count, GFP_KERNEL);
        if (copy_from_user(str, buffer, count)) {
                kfree(str);
                return -EFAULT;
        }
        sscanf(str, "%d", &param);
        printk(KERN_INFO "param has been set to %d\n", param);
        kfree(str);
```

1.3.3 Exercise: using the proc filesystem

Write a module that creates a /proc filesystem entry and can read and write to it.

When you read from the entry, you should obtain the value of some parameter set in your module.

When you write to the entry, you should modify that value, which should then be reflected in a subsequent read.

Make sure you remove the entry when you unload the module. What happens if you don't and you try to access the entry after the module has been removed?

The solution shows how to create the entry in the /proc directory and also in the /proc/driver directory.

1.3.4 Exercise: Making your own subdirectory in /proc

Write a module that creates your own proc filesystem subdirectory and creates at least two entries under it.

As in the first exercise, reading an entry should obtain a parameter value, and writing it should reset it.