Locking

Concurrency Aspects of Project 2

- Multiple producers, one consumer
  - Multiple users can issue system calls at the same time
- Need to protect all shared data
- Examples
  - Passengers may appear on a floor at the same time the elevator does
  - /proc/elevator might be read at the same time that you're updating the backing data
- How do you guarantee correctness?

Global vs Local

- Global data is
  - Declared outside of the functions
  - Before any function that uses it
  - Often required for kernel programming
  - Very sensitive to concurrency issues

Global vs Local

- Local data is
  - Declared within a functions
  - Sensitive to concurrency when
    - It depends on global data
    - Parallel access to the function is possible
  - Carefully consider whether they need to be synchronized

Synchronization Primitives

- Atomic functions
- Spin locks
- Semaphores
- Mutexes

Mutexes

- MUTual Exclusion
- Based on semaphores
- States
  - Locked
  - Unlocked
- Only one thread may hold the lock at a given time

Structure

- #include <linux/mutex.h>
- struct mutex my_mutex
- mutex_init(&my_mutex)

Locking

- mutex_lock (&my_mutex)
  - Waits indefinitely
- mutex_lock_interruptible(&my_mutex)
  - Locks so long as it is not interrupted
  - Returns 0 if succeeded
  - Returns <0 if interrupted
  - Preferred

Unlocking

- mutex_unlock(&my_mutex)
  - Guarantees that the mutex is unlocked

Example

- Compete module
  - Uses counter module as a template
- Two kthreads compete for a single variable
- /proc/compete shows which thread last wrote to it

Headers

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/proc_fs.h>
#include <linux/slab.h>
#include <linux/string.h>
#include <linux/kthread.h>
#include <linux/mutex.h>
#include <linux/delay.h>
#include <asm-generic/uaccess.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Britton");
MODULE_DESCRIPTION("Simple module featuring proc read");
```

Globals

```
#define ENTRY_NAME "compete"
#define PERMS 0644
#define PARENT NULL
static struct file_operations fops;

#define BUFFER_SIZE 20
#define KTHREAD_STRING_1 "kthread 1"
#define KTHREAD_STRING_2 "kthread 2"
static struct task_struct *kthread1;
static struct task_struct *kthread2;

static struct mutex shared_data_mutex;
static char *shared_data;

static char *message;
static int read_p;
```

Kthread Run

```
int my_run(void *data) {
    char *name = (char*)data;
    mutex_lock_interruptible(&shared_data_mutex);
    while (!kthread_should_stop()) {
        strcpy(shared_data, name);
        strcat(shared_data, "\n");

        mutex_unlock(&shared_data_mutex);
        schedule();
        mutex_lock_interruptible(&shared_data_mutex);
    }
    mutex_unlock(&shared_data_mutex);
    printk("The %s has terminated\n", name);
    return 0;
}
```

Proc Open

```
int compete_proc_open(struct inode *sp_inode, struct file *sp_file) {
    printk("proc called open\n");

    read_p = 1;
    message = kmalloc(sizeof(char) * BUFFER_SIZE, __GFP_WAIT | __GFP_IO |
   __GFP_FS);
    if (message == NULL) {
        printk("ERROR, counter_proc_open");
        return -ENOMEM;
    }

    mutex_lock_interruptible(&shared_data_mutex);
    strcpy(message, shared_data);
    mutex_unlock(&shared_data_mutex);
    return 0;
}
```

Proc Read

```
ssize_t compete_proc_read(struct file *sp_file, char __user
*buf, size_t size, loff_t *offset) {
    int len = strlen(message);

    read_p = !read_p;
    if (read_p) {
        return 0;
    }

    printk("proc called read\n");
    copy_to_user(buf, message, len);
    return len;
}
```

Proc Close

```
int compete_proc_release(struct inode *sp_inode, struct file *sp_file) {
    printk("proc called release\n");
    kfree(message);
    return 0;
}
```

Proc Init

```
static int compete_init(void) {
    printk("/proc/%s create\n", ENTRY_NAME);
    mutex_init(&shared_data_mutex);
    shared_data = kmalloc(sizeof(char) * BUFFER_SIZE, __GFP_WAIT | __GFP_IO |
   __GFP_FS);

    kthread1 = kthread_run(my_run, (void*)KTHREAD_STRING_1, KTHREAD_STRING_1);
    if (IS_ERR(kthread1)) {
        printk("ERROR! kthread_run, %s\n", KTHREAD_STRING_1);
        return PTR_ERR(kthread1);
    }

    kthread2 = kthread_run(my_run, (void*)KTHREAD_STRING_2, KTHREAD_STRING_2);
    if (IS_ERR(kthread2)) {
        printk("ERROR! kthread_run, %s\n", KTHREAD_STRING_2);
        return PTR_ERR(kthread2);
    }
```

Proc Init

```
    fops.open = compete_proc_open;
    fops.read = compete_proc_read;
    fops.release = compete_proc_release;

    if (!proc_create(ENTRY_NAME, PERMS, NULL, &fops)) {
        printk("ERROR! proc_create\n");
        remove_proc_entry(ENTRY_NAME, NULL);
        return -ENOMEM;
    }
    return 0;
}
module_init(compete_init);
```

Proc Exit

```
static void compete_exit(void) {
    int ret;
    ret = kthread_stop(kthread1);
    if (ret != -EINTR)
        printk("%s has stopped\n", KTHREAD_STRING_1);

    ret = kthread_stop(kthread2);
    if (ret != -EINTR)
        printk("%s has stopped\n", KTHREAD_STRING_2);

    kfree(shared_data);

    remove_proc_entry(ENTRY_NAME, NULL);
    printk("Removing /proc/%s.\n", ENTRY_NAME);
}
module_exit(compete_exit);
```