CS 111 Discussion Notes—Lab 2
Updated 26 Apr 2013

What you need to do

1. **Read fully** the lab 2 assignment: http://www.read.cs.ucla.edu/111/lab2
2. **Read fully** the useful kernel functions page: http://www.read.cs.ucla.edu/111/lab2functions
    1. **Note:** copy_from_user() and copy_to_user() are for lab 3. Use memcpy() instead.
3. **Review** the struct request structure in <linux/blkdev.h>. It's too big to put here.
    1. http://lxr.linux.no/#linux+v2.6.18/include/linux/blkdev.h is a useful page.
    2. You may also find <linux/wait.h> and <linux/spinlock.h> to be useful.
4. **Solve** the six exercises in osprd.c:
    1. Ex. 1: should be really easy
    2. Ex. 2: reading and writing
    3. Ex. 4-6: synchronization
    4. Ex. 3: cleanup

Some global data definitions in osprd.c

```
#define SECTOR_SIZE     512       // The size of each sector.
#define F_OSPRD_LOCKED  0x80000   // A flag indicating whether the file (disk) is locked.
#define NOSPRD          4         // Number of disks.

static int nsectors = 32;         // Number of sectors.
module_param(nsectors, int, 0);   // Can change this when starting the module.

static osprd_info_t osprds[NOSPRD];   // One osprd_info_t per disk. Since we're in a kernel
                                      // module, this is shared memory across all processes.
```

The osprd_info data type

```
typedef struct osprd_info {
        uint8_t *data;              // The actual disk data.
        osp_spinlock_t mutex;
        unsigned ticket_head;
        unsigned ticket_tail;
        wait_queue_head_t blockq;
        // (Additional internal elements)
} osprd_info_t;
```

Notes on the slides—part one

Slide 1: Critical sections (CS) are pieces of code in which you want only one process running at a time, usually because the code accesses shared resources that aren't themselves protected from multiple processes. Critical sections usually only require before-and-after (atomic) behavior (as opposed to all-or-nothing behavior). An example of where before-and-after behavior is needed is in a printer. In a multiprocessor system, it's best to keep critical sections as fast as possible.

Slide 2: In lab 2 you will have access to mutexes, which there are called spinlocks because a process trying to acquire a spinlock will spin until it receives a lock. Mutexes are simple, but from mutexes you can build more complex locks that have types and semantics associated with them. In lab 2, you will be implementing read and write locks (the behavior is specified in the assignment guidelines) that unlock processes in the order in which the processes called acquire(). The lab 2 kernel only uses one processor, so in the kernel there's never confusion as to which process called acquire() first. In Linux, each process has its own kernel stack, which implies that if one of them blocks while in the kernel, its stack won't be overwritten by another process.

Slide 3: Mutexes have two operations: acquire() and release(). Remember to always release a mutex after completing a critical section it protects.

Slide 4: A few things to know about CS and the locks that protect them.

Slide 5: As an example of building a complex lock from mutexes, we build a counterlock. This lock counts the number of processes waiting to get a lock on a CS. The reason we might want such a lock is that if there are lots of other processes waiting to get a lock, and the CS is slow, then we might prefer to not acquire the lock.

A counterlock has three methods. The last, nwaiting(), returns the number of processes that are currently waiting to acquire the lock. A counterlock has three data items: the counter itself, a lock to protect access to the counter, and a lock that protects the CS.

Slide 6: nwaiting() just returns the value of the counter, and release() releases the lock protecting the CS. acquire() first increments the counter, then acquires the CS lock, and then decrements the counter. So if the counter is initialized to 0, then at any time the counter has the number of processes waiting to get a lock on the CS.

Question 1: What assumptions were made that have `L._nwaiting++` in a CS but not return `L._nwaiting`?

Notes on the code—osprd.c

The first exercise is at lines 34–37:

```
MODULE_LICENSE("Dual BSD/GPL");
MODULE_DESCRIPTION("CS 111 RAM Disk");
// EXERCISE: Pass your names into the kernel as the module's authors.
MODULE_AUTHOR("Skeletor");
```

Hopefully you should be able to figure out how to solve this one.

Lines 65–66:

```
/* HINT: You may want to add additional fields to help
         in detecting deadlock. */
```

You're allowed to edit the code in the file however you want.

Lines 68–73: Other data elements. You can see how these are used at the end of the file. You won't need these to do lab 2.

Lines 80–100: Useful helper functions. They are defined later in the file. You don't need to understand the internals, just the interface, which is explained here.

Lines 115-121:

```
// EXERCISE: Perform the read or write request by copying data between
// our data array and the request's buffer.
// Hint: The 'struct request' argument tells you what kind of request
// this is, and which sectors are being read or written.
// Read about 'struct request' in <linux/blkdev.h>.
// Consider the 'req->sector', 'req->current_nr_sectors', and
// 'req->buffer' members, and the rq_data_dir() function.
```

Make sure to look over the definition of 'struct request'. It's big, but you'll need to use the elements in it. The rq_data_dir() function is explained on the useful kernel functions page.

Line 144: osprd_close_last() just makes sure that you shut down ramdisks cleanly so that processes don't block forever or keep locks that would hold over to the next time the disk is opened.

Line 173: osprd_ioctl() handles the locking and synchronization. All locking and synchronization is done through the ioctl() interface. You can read about ioctl() with man 2 ioctl.

Lines 189–222: Comments for exercise 4, which is to implement locking. The comments explain how read and write locks need to interact with each other, how to notify user code that the file is locked, and error codes.

Processes that can't get a read or write lock should block until they can. Waiting and waking is discussed in section 5 of the useful kernel functions page.

The ticketing works like a bakery queue. The last two lines of the comments refer to the ticketing.

Lines 230-235: Exercise 5 is basically the same as exercise 4, except that if you can't acquire a lock immediately, you return, don't block, and don't join the wait queue.

Lines 243–248: Exercise 6 is unlocking the ramdisk. This is mostly just accounting.

Lines 261–268: This initializes your disks and is called when the disk is first mounted. You may need to add code.

Notes on the slides—part two

Slide 8: An example of deadlock. Deadlock depends heavily on the order of execution; for example, if P_1 ran in its entirety, then P_2, and then P_3, everything would be fine. It's only when the processor chooses processes in a bad order that deadlock occurs. In this particular example, we don't know for sure that deadlock has happened until step 6; a different P_3' might first release its read lock on C before write-locking A.

Question 2: Suppose P_3 did release C (at step 6) before locking A. Could deadlock ever occur?

Slide 9: The rest of the deadlock discussion will be about detecting deadlock. First, we need to model the situation, and then we need an algorithm for our model.

Our model will be a dependency graph, which we'll call a "wait graph" because it's easier to think of the relation "p waits on q" with an arrow from $(p \rightarrow q)$ than from q to p. Unlike the dependency graph in lab 1.c, we want this graph to be dynamic because edges will be added and removed based on locks, which occur during runtime.

Slide 10: We show that this graph does actually model deadlock. Considering the example for slide 8, we have three processes. We add an edge from P_1 to P_2 in step 4 because in step 4, P_1 is forced to wait, and it won't stop waiting until P_2 release the lock. The necessary conditions for the proposition are:

1. All unblocked processes eventually terminate
2. All processes release any locks they're holding when they terminate (if not earlier)
3. No external events interfere

The argument is that if there isn't a cycle, then there's some process that's not blocked. So we can run it until it terminates; when it does, it releases its locks, and then there's a new process that's not blocked. This argument will form the basis for the cycle checking algorithm in slide 12.

Slide 11: To keep our model consistent with the state of the processes, a process that can't immediately get a lock adds an edge to the process that holds it, and a process that releases the lock releases all of the processes that are currently waiting for the lock. A slight difficulty with our model so far is that if a process has two locks, we have no way to differentiate between processes waiting for the first lock and those waiting for the second. To remedy this, we just note on each edge to which lock it corresponds.

Slide 12: The algorithm for checking for directed cycles in a graph. Using our argument from before, we'll have all nodes be ON initially, and have NEXT stand for a node that isn't waiting on anyone. We run the process to completion, at which point its node is turned OFF because the process isn't running anymore. At the end, if there are any processes that are still ON, it must be that they are all waiting (otherwise they would be NEXT), which is the definition of deadlock. The diagrams on the right show two very similar situations, one with deadlock and one without.

Slide 13: In slide 11 we needed to add a tag to each edge to keep track of different locks. Since read and write locks have special semantics, you'll need to modify this model to handle them.