

Computer Science 161: Operating Systems

Section Handout: Week 2

CS161 Course Staff / Stephen Dawson-Haggerty
cs161@fas.harvard.edu

0.1 Things to Think About TM

- Assignment 1 will be the first coding you will undertake for this class. Remember the coding/commenting hints/suggestions/demands that we have given you at this point.
- Remember also that some of this code (most importantly, the implementation of locks and CVs) may well become part of later assignments. All things being equal, the chance of this is about 50% given that the only other choice is using your partner's implementation.
- Measure twice, cut once!

1 Threads

- The *unit of CPU scheduling*.
- What is a process, then?
- User v. Kernel: where to implement threads? What happens to blocking system calls and scheduling policies? How do you implement preemptive threads in userspace?
- Pieces of threads and processes: `tcb`, `pcb`, `stack`, `kstack`, *etc.* Where does all this stuff go?

1.1 Interrupts

Interrupts are the way some part of the machine gets the attention of the processor as it is running some thread of execution. There are times when you will not want this to happen (notably in implementing synchronization primitives) when you do not want this to

happen, and hence you will need to disable interrupts. In general, however, you want to leave interrupts off for as short a time as possible. (Why?)

- `splhigh()`: Turn off interrupts
- `spl0()`: Turn on interrupts

Where should this be used? The pitfalls of `spl` synchronization...

2 Synchronization

2.1 So what's the problem?

Let's say there is a global variable `num_monkeys` to which both threads have access. Both threads want to add 3 to it, then multiply by 5. Let's begin with `num_monkeys = 2`. So after both threads have run, `num_monkeys` should be 140, right? But then you run your program and you are shocked and dismayed to see that `num_monkeys` has ended up as 200!! That's 60 too many monkeys!

Thread A	Thread B	Value of <code>num_monkeys</code>
Initial State	Initial State	2
Add 3		5
Multiply by 5		25
	Add 3	28
	Multiply by 5	140

What actually happened:

Thread A	Thread B	Value of <code>num_monkeys</code>
Initial State	Initial State	2
	Multiply by 5	200

Doh!

2.2 Critical Sections

Mutual Exclusion: At most one thread is currently executing inside the critical section

Progress: If a thread T1 is outside the critical section, then T1 cannot prevent T2 from entering.

Bounded waiting: If thread T1 is waiting to get into the critical section, then T1 will eventually be allowed to enter.

Performance: The overhead of entering and exiting the critical section is small with respect to the work being done.

```
char sleep_addr_t1, sleep_addr_t2;
```

```
void t1(void) {  
    thread_sleep(&sleep_addr_t1);  
    kprintf("t1 says hello\n");  
    thread_wakeup(&sleep_addr_t2);  
}
```

```
void t2(void) {  
    kprintf("t2 says hello\n");  
    thread_wakeup(&sleep_addr_t2);  
    thread_sleep(&sleep_addr_t1);  
}
```

And another one.

```
int t = 0;
```

```
void t1(void) {  
    while (++t == t) ;  
    kprintf("Hmm? t1 exits\n");  
}
```

```
void t2(void) {  
    while (++t == t) ;  
    kprintf("Hmm? t2 exits\n");  
}
```

2.3 Account Transfers

To get started, let's consider a bank. There is a large table of accounts, and you want to design a function to transfer money from one account to another. The function will be given references to two different accounts, and an amount to move from the first to the second. Suppose each account has a “balance lock” associated with it, so that in order to read or write the account balance, you must hold that lock. How do you lock the accounts to prevent deadlock? Assume that once you lock an account, it must remain locked until the transfer is complete.

3 Comments, Style, Random Tips

We talked a little about this last time, but keep a keen eye to commenting style. A couple of tips:

- group related items together.
- use lots of one-line whitespace, as necessary. Don't use huge chunks of whitespace.
- lots of comments – clear and concise.
- descriptive variable names.
- consistent variable and function naming.
- what *do* `static` and `volatile` mean? How about `goto`?

One other interesting note. One line of C code is not necessarily atomic:

```
c = (a > b);
```

Is that atomic? Nope. Really? Well, probably not. (`sgt`). A better example is something like `c = a + b + c;`. No chance of that being atomic.

Appendix

Files and Functions

- `kern/thread/synch.c`
 - Things Implemented for You

```

* struct semaphore * sem_create(const char *namearg, int initial_count)
* void sem_destroy(struct semaphore *sem)
* void P(struct semaphore *sem)
* void V(struct semaphore *sem)

```

– Things You Have to Implement

```

* struct lock * lock_create(const char *name)
* void lock_destroy(struct lock *lock)
* void lock_acquire(struct lock *lock)
* void lock_release(struct lock *lock)
* int lock_do_i_hold(struct lock *lock)
* struct cv * cv_create(const char *name)
* void cv_destroy(struct cv *cv)
* void cv_wait(struct cv *cv, struct lock *lock)
* void cv_signal(struct cv *cv, struct lock *lock)
* void cv_broadcast(struct cv *cv, struct lock *lock)

```

- kern/thread/thread.c

– Things Implemented for You

```

* void thread_yield(void) - it yields the processor to another thread. The
  thread stays runnable! Maybe it just wanted a rest ...
* void thread_sleep(const void *addr) - it yields the processor to another
  thread. A subsequent call to thread_wakeup with the same addr will wake
  up the thread again. addr can be anything that you want it to be. Interrupts
  must be off before calling this function.
* void thread_wakeup(const void *addr) - it wakes up any and all threads
  sleeping on addr. It does not necessarily cause them to run immediately, it
  merely makes them runnable.

```

- kern/thread/hardclock.c

– void clocksleep(int num_secs) - yields the processor for num_secs seconds.

Synchronization Primitives

What they are

- Semaphores:
 - Types:

- * Binary - 0 or 1. Generally protects a single resource.
- * Counting - any number ≥ 0 . Often used to allocate a pool of resources.
- Operations:
 - * P() - decrement semaphore, or if 0, wait until semaphore is available, then decrement. Used to acquire a resource.
 - * V() - increment semaphore. Used to signal the freeing of a resource.
- Notes:
 - * These are different from locks!
 - * No thread “owns” the semaphore. One thread may V() it while another P()’s it, or vice versa.
- Locks
 - Operations:
 - * Acquire - claim ownership of a lock, or if unavailable, wait until current owner releases and then claim it.
 - * Release - release ownership of a lock so that another thread can claim it.
 - Notes:
 - * locks are owned by a particular thread.
 - * only the owner of a lock may release it.
 - * who gets woken up when a lock is released?
- Condition Variables
 - Operations:
 - * Wait - wait for another thread to signal/broadcast on this condition variable
 - * Signal - signal exactly one waiting thread.
 - * Broadcast - signal all waiting threads.
 - Notes:
 - * generally, a lock is associated with a given condition variable.
 - * sometimes multiple condition variables will be associated with a given lock.
 - * you must hold the lock to perform any of these operations.

Semantics

Semantics (n) The meaning or the interpretation of a word, sentence, or other language form.

You may hear computer scientists or your TFs bandying this word around; it will usually sound something like “well sure, but that doesn’t reflect the correct semantics.” What do they mean?

The idea is that when we make call `lock_acquire()`, we want to have a precise concept of what we expect that function to do. The issue of semantics comes up a lot in interface design, which it turns out has a lot to do with kernels: system calls provide the interface into the kernel, and we want their behavior to be as well-defined and deterministic as possible. In practice, this means doing a lot of testing on the arguments and having a million failure cases. See `errno.h` for more...

We do tell you that you should implement *Mesa Semantics* for condition variables; however there are other subtleties even in something as simple as this assignment. What do you do if a thread attempts to acquire a lock it already holds?