CS61: Advanced concurrency
December 3, 2013

Announcements:

- Final: Wednesday December 18 at 9am Science Center C(?)
- Fill out Q Guide!
- Eddie will be having office hours during reading period
- If you want to implement extra credit for problem sets, won't count against late days
- Everything due December 11. No work will be graded afterwards

Implementing Concurrency

**How can we implement a mutual-exclusion lock?**

One possible way is to use pipes, and reading/writing, but the issue is that pipes are expensive

**How can we implement a mutual-exclusion lock without kernel support (using only basic memory functions)?**

Idea: Associate with the mutex some chunk of memory, and look at memory and decide what to do based on its contents

```
pthread_mutex_t {
      int islocked;
}

pthread_mutex_init(pthread_mutex_t *m) {
      // mutex starts out unlocked
      m->islocked = 0;
}

// v1 NOT CORRECT Because use up all the memory in the while loop! (see below
for //correct code)
pthread_mutex_lock(pthread_mutex_t *m) {
      // any lock will have a while statement in it because it should block
      for an //amount of time determined by this thread and other threds
      while(m->islocked) /* do nothing*/ ;
      m->islocked = 1;
}

pthread_mutex_unlock(pthread_mutex_t *m) {
      m->islocked = 0;
}
```

| | |
|---|---|
| spin lock | lock that uses CPU while in the "attempting to lock" state |

| | |
|---|---|
| blocking lock | doesn't use CPU |

In general, prefer operations that block to operations that spin. But, would prefer spin if you want to find out as soon as possible when a lock becomes unlocked. With spin, lower latency between time that you unlock and the time you run. For now, will implement spin lock. Implementing a blocking lock inherently uses the operating system

Instructions with atomic effects/without

-loading value from memory is atomic

-storing a value into memory is generally atomic (not when data is stored in unaligned way so that the data overlaps more than one cache line -> cache lines are the unit of atomicity!)

- registers

- `incl(%eax)` - increment isn't necessarily atomic because three steps are involved, and the hidden register can be changed:

```
incl (%eax)
      (%eax)++
      Involves read [HIDDEN]<-(%eax)
            [HIDDEN]++
            (%eax)<- [HIDDEN]
```

`lock incl(%eax)` -- atomic increment!

^Intel architects gave us a command called lock

Another instruction:

```
// exchange. takes two arguments, register and memory (or register and
register)
// atomically exchanges values of SRC and DST, and returns the SRC
lock xchgl SRC, DST
      // in C terms:
      tmp = src;
      src = dst;
      dst = tmp;
      // but executed in one atomic step
```

Testing whether a lock is locked should not change its lockedness.

Back to implementing pthread_mutex_lock

```
// v2 CORRECT to fix v1, need to make checking for the lock and setting the
lock //atomic without using the kernel
pthread_mutex_lock(pthread_mutex_t *m) {
      // any lock will have a while statement in it because it should block
      for an //amount of time determined by this thread and other threads
      while (xchgl(&m->islocked, 1)) == 1) /*do nothing*/;
```

```
      m ->islocked = 1;
}
```

Lock Advantages relative to pipe:
A lock takes up 4 bytes of memory
A pipe takes up 64 KB (buffer)

Disadvantages:
This lock spins, it doesn't block

**How do we implement a condition variable?**

`wait(cond *c, mutex *m)` -> block until a signal releases m
`signal(cond *c)` -> wake oldest waiting thread (if any waiting)

```
// struct for linked list of threads that asked for wake, in order. On stack
p_c {
      // add condition variable to deal with race conditions
      pthread_mutex_t m;
      cond_waiter *first;
}

// struct for node of linked list of every thread that's waiting, so we have
an order //guarantee
cond_waiter {
      // whether or not watier is currently waiting
      int waiting;
      cond_waiter *next;
}

p_c_wait(c, m) {
      unlock(m);
      // putting lock here and unlock at the end of this function introduces
      //DEADLOCK. Because it is holding the lock as it is waiting, so it never
      //unwaits -> circular wait
      // every thread that waits will have a waiter
      cond_waiter w;
      w.waiting = 1;
      w.next = NULL;
      // lock(&c->m) can also go here because previous code only modifies the
stack
      // this is a better place for the lock because then you hold the lock
      for a //shorter period of time -> shorter critical section
      lock(&c->m);
      // unlock can also go after putting &w at the end of the list
      unlock(&c->m);
```

```
        put &w at the end of the c->first list;
        while (w.waiting) /* spin */;s
        lock(m);
}

p_c_signal(c) {
        lock(&c->m);
        cond_waiter *w = c->first;
        if (w) {
                // you ain't waiting anymore
                w->waiting = 0;
                c->first = w->next;
        }
        unlock(&c->m)
}
```