CMSC421: Principles of Operating Systems

Nilanjan Banerjee

Assistant Professor, University of Maryland
Baltimore County
nilanb@umbc.edu
http://www.csee.umbc.edu/~nilanb/teaching/421/

Acknowledgments: Some of the slides are adapted from Prof. Mark Corner and Prof. Emery Berger's OS course at Umass Amherst

Announcements

- Project 1 due on Oct 7^{th}
- Homework 2 is out (due Oct 13^{th})
- Readings from Silberchatz [6^{th} chapter]

Test and Set semantics

- What's the effect of **testAndset(value)** when:
  - value = 0? ("unlocked")
  - value = 1? ("locked")

```
int testAndset (int* v) {
  int old = *v;
  *v = 1;
  return old;
}
```

Blocking Locks

- Suspend thread immediately
  - Lets scheduler execute another thread
  - Go to back of run queue or wait to be woken
- Minimizes time spent waiting
- But: always causes context switch

```
void blockinglock (Lock* l) {
  while (testAndSet(l.v) == 1) {
    sched_yield();
  }
}
```

Spin Locks

- Instead of blocking, loop until released

```
void spinlock (Lock* l) {
  while (testAndSet(l.v) == 1) {
    ;
  }
}
```

Other variants

- Spin-then-yield:
  - Spin for some time, then yield
    - Fixed spin time
    - Exponential backoff

Safety

- Locks can enforce mutual exclusion, but notorious source of errors
  - Failure to unlock
  - Double locking
  - Deadlock (its own lecture)
  - Priority inversion
    - not an “error” per se

```
pthread_mutex_t l;
void square (void) {
  pthread_mutex_lock (&l);
  // acquires lock
  // do stuff
  if (x == 0) {
    return;
  } else {
    x = x * x;
  }
  pthread_mutex_unlock (&l);
}
```

Bounded Buffer or Producer Consumer Problem

- Suppose we have a thread-safe queue
  - insert(item), remove(), empty()
  - must protect access with locks
- Consumer
  - Consumes items in the queue
  - Only if queue has items
- Producer:
  - Produces items
  - Adds them only if the queue is not full

A simple case: max size of queue is 1

Solution (sleep?)

- Sleep =
  - "don't run me until something happens"
- What about this?

```
Dequeue(){
   lock();
   if (queue empty) {
      sleep();
   }
    take one item;
   unlock();
}
```

```
Enqueue(){
   lock();
   insert item;
   if (thread waiting)
      wake up dequeuer();
   unlock();
}
```

Another solution

- Does this work?

```
Dequeue(){
   lock();
   if (queue empty){
      unlock();
      sleep();
      remove item;
   }
   else unlock;
}
```

```
Enqueue(){
    lock();
    insert item;
    if (thread waiting)
       wake up dequeuer();
    unlock();
}
```

Conditional variables

- Make it possible/easy to go to sleep
  - Atomically:
    - release lock
    - put thread on wait queue
    - go to sleep
- Each cv has a queue of waiting threads
- Worry about threads that have been put on the wait queue but have NOT gone to sleep yet?
  - no, because those two actions are atomic
- Each condition variable associated with one lock

Conditional variables

- Wait for 1 event, atomically release lock
  - **`wait(Lock& l, CV& c)`**
    - If queue is empty, wait
      - Atomically releases lock, goes to sleep
      - You must be holding lock!
      - May reacquire lock when awakened (pthreads do)
  - **`signal(CV& c)`**
    - Insert item in queue
      - Wakes up one waiting thread, if any
  - **`broadcast(CV& c)`**
    - Wakes up all waiting threads
- Monitors = locks + condition variables
  - Sometimes combined with data structures

Lets take a look at a demo

Producer-consumer problem using pthread conditionals

```
void * consumer (void *){
  while (true) {
   pthread_mutex_lock(&l);
   while (q.empty()){
      pthread_cond_wait(&nempty, &l);
   }
   cout << q.pop_back() << endl;
   pthread_mutex_unlock(&l);
  }
}
```

```
void * producer(void *){
  while (true) {
   pthread_mutex_lock(&l);
   q.push_front (1);
   pthread_cond_signal(&nempty);
   pthread_mutex_unlock(&l);
  }
}
```

Semaphores

- Computer science: Dijkstra (1965)

A non-negative integer counter with atomic increment & decrement. Blocks rather than going negative.

Semaphore Operations

- P(sem), a.k.a. wait = decrement counter
  - If sem = 0, block until greater than zero
  - P = "prolagen" (proberen te verlagen, "try to decrease")
- V(sem), a.k.a. signal = increment counter
  - Wake 1 waiting process
  - V = "verhogen" ("increase")

How do you implement a mutex using semaphores

- More elegant than locks
  - Mutual Exclusion **and** Ordering
  - By initializing semaphore to 0,
    threads can wait for an event to occur

thread A

```
// wait for thread B
sem.wait();
// do stuff …
```

thread B

```
// do stuff, then
// wake up A
sem.signal();
```

Counting Semaphores

- Controlling resources:
  - E.g., allow threads to use at most 5 files simultaneously
    - Initialize to 5

thread A

```
sem.wait();
// use a file
sem.signal();
```

thread B

```
sem.wait();
// use a file
sem.signal();
```

An in-class discussion
(producer consumer problem using sem)