Lecture 9: September 30

Lecturer: Prashant Shenoy *Scribe: Shashi Singh*

9.1 Implementing Locks

No matter how we choose to implement them, we *must* have some hardware support. One possibility to implement locks is to disable interrupts, since these are the only way that a CPU has to change what it is doing; in other words, if we keep the CPU from switching processes, we can guarantee that only one process (the active one) will have access to the shared data. Another option would be to make use of atomic operations, such as test&set. This operation (which usually correspond to an assembly instruction), is such that test&set(x) returns 1, if x=1; otherwise, if x=0, it returns 0 and sets x to 1. All this is of course implemented atomically. Having this type of atomic operation, one could implement *thread_lock*(l) simply as

```
while test&set(l) do nothing;
```

and *thread_unlock*(l) simply as

```
l = 0;
```

This method of acquiring a Lock leads to *busy waiting*, wherein CPU cycles are wasted doing nothing useful. Not only is this inefficient, it could cause problems if threads can have different priorities. If the busy-waiting thread has higher priority than the thread holding the lock, the timer will go off, but (depending on the scheduling policy), the lower priority thread might never run. Though it is not possible to completely get rid of busy waiting, we can minimize it by atomically checking the lock value and giving up the CPU if the lock is busy.

```
Lock::Acquire()
      Disable interrupts;
      while (value != FREE) {
            put on queue of threads waiting for lock
            go to sleep
      }
      value = BUSY;
      Enable interrupts;

Lock::Release()
      Disable interrupts;
      if anyone on wait queue {
            take a waiting thread off
            put it on ready queue
      }
      value = FREE;
      Enable interrupts;
```

9.2 Semaphores

A semaphore is basically used to regulate traffic in a critical section. A semaphore is implemented as a non-negative integer counter with atomic increment and decrement operations; whenever the decrement operation would make the counter negative, the semaphore blocks instead. The increment operation is called P; the decrement is called V:

- *P(sem)*: tries to decrease the counter by one; if the counter is zero, blocks until greater than zero. "P" can be interpreted as *wait*;
- *V(sem)*: increments the counter; wakes up one waiting process. "V" can be interpreted as *signal*.

Binary Semaphore: A binary semaphore must be initialized with 1 or 0, and the implementation of P and V operations must alternate. If the semaphore is initialized with 1, then the first completed operation must be P. If the semaphore is initialized with 0, then the first completed operation must be V. Both P and V operations can be blocked, if they are attempted in a consecutive manner.

Counting Semaphore: A counting semaphore can be considered as a pool of permits. A thread used P operation to request a permit. If the pool is empty, the thread waits until a permit becomes available. A thread uses V operation to return a permit to the pool. A counting semaphore can take any initial value.

Notice that we can use semaphores to implement both mutual exclusion and ordering constraints. For example, by initializing a 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()
```

Let us now present how to implement an unbounded consumer-producer with semaphores.

```
semaphore sem_mutex = 1  // this is for mutual exclusion
semaphore slots_left = N
semaphore cokes_left = 0

producer()
    P(slots_left)
    P(sem_mutex)
       inserts coke into machine
    V(sem_mutex)
    V(cokes_left)

consumer()
    P(cokes_left)
    P(sem_mutex)
       buy coke
```

```
    V(sem_mutex)
    V(slots_left)
```

9.2.1 Implementing Signal and Wait

```
class Semaphore {
      public:
            void Wait(Process P);
            void Signal();
      private:
            int value;
            Queue Q; // queue of processes;
}
Semaphore::Semaphore(int val) {
      value = val;
      Q = empty;
}

Semaphore::Wait(Process P) {
      value = value - 1;
      if (value < 0) {
            add P to Q;
            P->block();
      }
}
Semaphore::Signal() {
      value = value + 1;
      if (value <= 0){
            remove P from Q;
            wakeup(P);
      }
}
```