Lecture 8: February 11

Lecturer: Prashant Shenoy *TA: Vimal Mathew & Tim Wood*

8.1 Implementing Locks

To implement a lock, the OS needs a way to ensure that a process will be able to run a *critical section* where it can access some shared data without another process modifying the data at the same time. No matter how an OS chooses to implement locks, it *must* have some hardware support. One way to implement locks is to *disable interrupts*, since interrupts are the only way that a CPU has to change what it is doing. Normally, a process in an operating system will continue running unless it either performs an I/O request or it is interrupted by the operating system through the use of an interrupt–for example because it's scheduling time quantum has run out or due to some kind of exception. By disabling interrupts, we can ensure that a process will maintain control of the CPU and guarantee that only one process (the active one) will have access to the shared data. Disabling interrupts will prevent any *external events* from causing the process to lose control of the CPU. In addition, the OS must prevent *internal events* as well; this is typically done by preventing a process from initiating an IO request while in a critical section.

Another option for implementing locks 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. Each of these operations must be implemented atomically by the hardware. Having this type of atomic operation, one could implement $acquire(L)$ simply as

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

and $release(L)$ simply as

```
l = 0;
```

A lock implementation can either use busy waiting or wait queues. In *busy waiting*, when a process calls acquire(), it will continuously check the lock to see when it becomes available again. While this is simple to implement, it can be inefficient because the process is using CPU time to constantly check a lock which another process has control of. An alternative is to use *wait queues*. In this case, when a process calls acquire(), the OS places it on a wait queue. This is a list of processes which are all waiting for the lock to become available. These processes will not be scheduled by the OS until the process currently using the lock calls release(). This is a more efficient approach, but the lock implementation needs to be very careful about how disabling and enabling interrupts is controlled as processes are put to sleep on the wait queue.

8.2 Semaphores

A semaphore is a more generalized form of a lock that can be 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, or *signal*; the decrement is called V, or *wait* :

- *Semaphore.wait()*: tries to decrease the counter by one; if the counter is zero, blocks until greater than zero.
- *Semaphore.signal()*: increments the counter; wakes up one waiting process.

Binary Semaphore: A binary semaphore must be initialized with 1 or 0, and the implementation of *wait* and *signal* operations must alternate. If the semaphore is initialized with 1, then the first completed operation must be *wait*. If the semaphore is initialized with 0, then the first completed operation must be *signal*. Both *wait* and *signal* 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 *wait* operation to request a permit. If the pool is empty, the thread waits until a permit becomes available. A thread uses *signal* 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 locks and ordering constraints. For example, by initializing a semaphore to 1, 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()
```

8.2.1 Implementing Signal and Wait

The signal and wait operations of a semaphore can be implemented as follows.

```
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);
      }
}
```

8.2.2 Using Semaphores

Semaphores can be used to implement critical sections or to enforce certain scheduling constraints. *Critical sections* are typically built by using semaphores that are initialized to 1. In this case, one process can call wait() in order to enter the critical section, preventing any other processes from passing the wait condition. Once the process finishes the critical section, it calls signal() which will allow the next process to enter the critical section. In some cases, it is useful to start the semaphore with a value greater than 1. This allows multiple processes to enter the critical section at once. While this can be a problem if the critical section is supposed to be protecting data, it can be useful in other cases such as where a limited number of resources are available for simultaneous use, such as in the Multiple Consumers and Producers Problem.

Semaphores can be used for *scheduling constraints* by initializing the semaphore to 0 and only incrementing it once a certain condition is met. This will cause all processes to initially block during their wait calls. Only once the condition is met will signal be called, allowing one of the waiting processes to continue execution. An example use case of this is for implementing the *waitpid()* function which causes a process to wait until another process exits. By setting the semaphore to 0 initially, any process that calls waitpid would enter a waiting state. Once the process being waited on completes, the signal() call could be made, causing the waiting processes to wake up.