Lecture 8: September 27

Lecturer: Prashant Shenoy *TA: Sean Barker & Demetre Lavigne*

8.1 Semaphores

A semaphore is a more generalized form of a lock that can be used to regulate traffic in a critical section or to order code execution. They were invented by the renown computer scientist Edsger Dijkstra in 1965 (read about him at `http://en.wikipedia.org/wiki/Edsger_W._Dijkstra`). Even though a semaphore is more generalized than a lock, they are not more powerful (just more flexible). A semaphore is implemented as an integer variable with atomic increment and decrement operations; so long as the value is not negative the thread will continue, it will block otherwise. The increment operation is called *P*, or *signal*; the decrement is called *V*, or *wait* :

- *Semaphore.wait()*: decreases the counter by one; if the counter is negative, then it puts the thread on a queue and blocks.
- *Semaphore.signal()*: increments the counter; wakes up one waiting process.

Binary Semaphore: A binary semaphore is initialized as free (1) and can vary from 0 to 1. This semaphore is essentially the same as a lock. It also guarantees that only one process will be in a critical section at a time.

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. These can be used to manage multiple resources and they are typically initialized to the number of resources.

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()
```

8.1.1 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);
      }
}
```

8.1.2 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.

Multiple Consumers and Producers Problem: The Multiple Consumers and Producers Problem is when we have multiple threads producing items and multiple threads consuming items. If this is implemented with the shared data being in some sort of bounded buffer, then it is necessary to ensure certain orderings. If all of the bounded buffer is full, then the producers need to be blocked until atleast one consumer runs. The reverse condition also needs to hold: if the buffer is empty, then none of the consumers should run until atleast one producer runs. These ordering can be ensured using two semaphores and the buffer should be protected by a mutex.

Semaphores can also 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.

8.2 Monitors

While semaphores can be a powerful synchronization mechanism, they have some drawbacks which make them difficult to use in practice. Semaphores are a low level mechanism, and improperly placed signal and wait calls can lead to incorrect behavior. Additionally, semaphores are a global data structure which are not explicitly tied to a critical section–they can be called from any thread at any point in the program. Semaphores also try to solve two problems (mutual exclusion and ordering) with the same device. These characteristics can cause semaphores to be difficult to use, and improper usage can easily lead to bugs.

A monitor is a higher level synchronization mechanism that tries to resolve some of these issues. A monitor can be viewed as a class that encapsulates a set of shared data (declared as private) as well as the operations on that data (e.g. the critical sections). The monitor is implemented in such a way so as to guarantee mutual exclusion–only one thread calling a method in the monitor class can run at a time. We consider a thread to be "in a monitor" if it has acquired control of the monitor; the monitor automatically ensures that only one thread can be in the monitor at a time.

A monitor contains a lock and a set of *condition variables*. The lock is used to enforce mutual exclusion. The condition variables are used as wait queues so that other threads can sleep while the lock is held. Thus condition variables make it so that if a thread can safely go to sleep and be guaranteed that when they wake up they will have control of the lock again. In Java, monitors can be used by adding the *synchronized* keyword to a method declaration. This will make it so that only one thread can execute the method at a time. This eliminates the need to manually acquire and release locks or call semaphore operations–the mutual exclusion is provided through the monitor system.

8.3 Condition Variables

A thread in a monitor may have to block itself so it can wait for an event, but this can cause a problem since the thread has already acquired the monitor lock–if it were to go to sleep, no other thread could enter the monitor. To allow a thread to wait for an event (e.g., I/O completion) without holding on to the lock, condition variables are used. If a thread must wait for an event to occur, that thread waits on the corresponding condition variable. If another thread causes an event to occur, that thread simply signals the corresponding condition variable. Thus, a condition variable has a queue for those threads that are waiting for the corresponding event to occur. Each monitor may have multiple condition variables.

8.3.1 Condition Variable Operations: Wait and Signal

There are two operations that can be applied to a condition variable: *wait* and *signal*, which are similar to the semaphore calls. It is important that a thread must hold the lock when using these operations. When a thread executes a wait call on a condition variable, it is immediately suspended and put into the waiting queue of that condition variable. Thus, this thread is suspended and is waiting for the event that is represented by the condition variable to occur. Because the calling thread is the only thread that is running in the monitor, it "owns" the monitor lock. When it is put into the waiting queue of a condition variable, the system will automatically take the monitor lock back. As a result, the monitor becomes empty and another

thread can enter. Eventually, a thread will cause the event to occur. To indicate a particular event occurs, a thread calls the signal method on the corresponding condition variable. At this point, we have two cases to consider. First, if there are threads waiting on the signalled condition variable, the monitor will allow one of the waiting threads to resume its execution and give this thread the monitor lock back. Second, if there is no waiting thread on the signalled condition variable, this signal is lost as if it never occurs.

There is a third operation *Broadcast()* that wakes up all waiting threads in the queue, instead of just one.