CS162, Spring 2004 **Discussion #4** **Amir Kamil**
UC Berkeley **2/12/04**

Topics: Monitors

1 Announcements

- Project design reviews will be held the week after you turn in your design docs. All members of your group must attend, and you are graded on both the doc and the review. Make sure your entire group knows the design for all parts. We will quiz each member on aspects of your design.

2 Monitors

In lecture, we saw the simplest higher-level synchronization construct, the *semaphore*. Recall that a semaphore contains a counter and two operations, `P()` and `V()`. The former decrements the counter and blocks until the counter is non-negative, and the latter increments the counter. We can use these operations to both implement mutual exclusion and for scheduling purposes. However, this can get confusing, as you saw in the producers and consumers problem, so it is useful to separate these two functions into different constructs, the lock and the condition variable. A lock and its associated condition variables are known as a *monitor*.

It is important to note that all these synchronization constructs are essentially equivalent; they can be built out of each other. However, each is most useful for a specific purpose, and you should use the construct that makes the most sense to use for a particular problem.

2.1 Locks

While semaphores can be used for mutual exclusion, they have no concept of an owner, so any thread can call `V()` and allow another to enter a critical section while that section is still occupied. A *lock*, on the other hand, keeps track of who owns it. The lock provides two functions, `acquire()` which obtains ownership of a lock if it is free and waits until it is if not, and `release()`, which releases ownership of a lock when called by its owner. Thus a lock can be used for mutual exclusion; a thread calls `acquire()` when entering a critical section and `release()` when exiting. Since only the owner can release a lock, this ensures that only a single thread executes a critical section at a time. And less confusion arises from using locks, since they are only used for mutual exclusion.

```
class Lock {
  Semaphore s = 1;
  Thread owner = null;
  acquire() {
    s.P();
    owner = current thread;
  }
  release() {
    if (owner == current thread) {
      owner = null;
      s.V();
    }
  }
}
```

Figure 1: A lock implemented in terms of a binary semaphore.

```
int numCokes, MAXCOKES;
Lock lock;
Condition wantToAdd = new Condition(lock);
Condition wantToTake = new Condition(lock);

Producer() {
  lock.acquire();
  while (numCokes == MAXCOKES) {
    wantToAdd.sleep(); // Lock released
  }
  numCokes++;   // Lock reacquired
  wantToTake.wakeAll();
  lock.release();
}

Consumer() {
  lock.acquire();
  while (numCokes == 0) {
    wantToTake.sleep(); // Lock released
  }
  numCokes--;  // Lock reacquired
  wantToAdd.wakeAll();
  lock.release();
}
```

Figure 2: Producers and consumers using condition variables.

2.2 Condition Variables

The corresponding higher-level construct used for scheduling and notification is the *condition variable (CV)*. A condition variable has an associated lock, and all operations on a CV require that the lock be held. Three operations are provided: `sleep()` atomically puts the calling thread to sleep and while releasing the associated lock and reacquires it when the thread wakes up, `wake()` wakes a single sleeping thread, if any, and `wakeAll()` wakes all sleeping threads. Note that `sleep()` at least must be atomic. What would happen if the act of releasing the lock putting a thread to bed was not atomic in `sleep()`?

There are actually two types of condition variables. In *Mesa-style* condition variables, a thread must manually reacquire the lock when it is woken up (this is done at the end of the `sleep()` function). In *Hoare-style* condition variables, when a thread calls `wake()`, it gives the lock to the thread woken up and puts that thread on the CPU. The disadvantage of Mesa is that a third thread can take the lock before the woken thread gets a chance to, while the advantage is that it is much easier to implement. Nachos uses Mesa-style condition variables.

Condition variables are intended to be used inside a critical section to notify threads of an occurrence of some event. For example, in the producers/consumers problem, the producer and consumer code compose a critical section and must be guarded by a lock. Consumers wait for there to be a coke to take, while producers wait for there to be space to add a coke. Condition variables can be used to signify these two events as shown in figure 2. This implementation is much cleaner than the semaphore implementation in lecture.