

CS414: Operating Systems

Spring 2008

Nachos 2 Hints

Background

Semaphores, locks, and condition variables are different kinds of *synchronization mechanisms*. They are generally used to synchronize the processing of more than one process or thread. In particular, they are often used to control access to some shared resource (shared memory, physical devices, etc.). In doing so, they each function a little differently, but fundamentally they do the same thing: allow one process to access the shared resource at a time and delaying the other processes until it is their turn.

The reason for having different primitives (semaphores, locks, condition variables, monitors) is a) because sometimes one calling interface works better for a given problem than some other, and b) because different people implemented these in different ways in various important systems in the past and it is good to know about all of them.

Semaphores

Semaphores support two operations: P() and V() that work as follows:

- P() - Check the value of the semaphore. If it is greater than 0, decrease it by 1 and return. If it is 0, wait until it is greater than 0, then decrease it by 1 and return.
- V() - Increase the value of the semaphore by 1.

Note that the initial value of the semaphore, set when the semaphore is created, can be anything you want.

Semaphores can be used for many different synchronization purposes, but a common one, mentioned above, is to protect access to a shared resource. To do this, given a semaphore S with an initial value of 1, any number of processes can do the following:

```
// other code here

S.P();

// access the shared resource here

S.V();

// other code here
```

As long as access to the shared resource is surrounded by P() and V() calls, it is guaranteed that only one of them will be accessing the shared resource at a time.

However, semaphores can be used for other synchronization purposes as well. Specifically, a process can do a V() operation without having to do a P() operation first. This is useful when one process wants to wait for another process to do something. The "waiting" process can do a P() on a semaphore and the "doing" process can do a V() on that same semaphore when it has done whatever the "waiting" process was waiting for.

Locks

Locks are almost the same as semaphores, but are slightly more restricted in terms of how they work. Locks support two functions:

- Acquire() - If the lock is available, get it and return. If not, wait until it is available, then get it and return.
- Release() - If the calling process currently holds the lock, release it and return. If not, do nothing.

Locks are almost the same as semaphores, except for two key differences:

- Locks are always initially free. In other words, the count for a lock is always initially set to 1.
- Only the "owner" of a lock (the process that has Acquired the lock) can do a Release(). After releasing it, the lock is free (i.e. no longer owned by that process, so it can't do another Release() unless it first does another Acquire()).

Condition Variables

Condition variables are also similar to semaphores. They support three operations:

- Wait() - Wait until some process does a Signal() or a Broadcast()
- Signal() - Wake up one waiting process
- Broadcast() - Wake up all waiting processes

Condition variables are associated with locks as follows: to use the condition variable (i.e. to call the condition variable functions), the process must currently hold the lock associated with it.

Note that condition variables differ from semaphores and locks in a couple of ways:

- When a process does a Wait(), it always blocks until some other process does a Signal() or a Broadcast(). In other words, conditions don't store a value the same way that semaphores and locks do.
- Signal() wakes up one waiting process, while Broadcast() wakes up all waiting processes. Semaphores and locks have nothing equivalent to a Broadcast().

What to do for nachos2

There are three parts to this assignment. First you must add locks and condition variables to Nachos. Second, show that they work by implementing a bounded buffer. Third, implement the bridge problem using the locks and condition variables.

1. Hint #1: Use semaphores to implement locks. Include a semaphore as a member variable to the lock class, initialize it appropriately, keep track of the owner, and you are practically done.

Hint #2: Use semaphores to implement condition variables. This is a little trickier, but not much. Add a list of waiting threads to implement signal and broadcast.

2. Hint #1: read() and write() should only read or write one character. The Producer and Consumer have to call write() or read() once for each character they wish to write or read. Declare the bounded buffer as a global variable as all threads (producer and consumer) have to share the bounded buffer.
3. Hint #1: ArriveBridge() and ExitBridge() take only one car at a time. If a car is moving in direction 1 and a car arrives at the opposite end (direc 0), the second car must wait before ExitBridge() returns for the first car to avoid a collision. This must be true for all implementations. The question in part b is different, it says that in this situation, if a car (direc 1) arrives, which car crosses the bridge first? the second or the third car? This depends on your implementation.