

# CS162 – Section # 4, 02/18/2003

Rodrigo Fonseca

---

## Announcements

Design document due tomorrow  
See examples on class web site

## Priority inversion problem

Scheduling is discussed in the book, but will be discussed later during class. For now, just consider that threads may be given priority. This affects the dispatcher's decision at `runNectThread()`. From the Nachos code, "*Essentially, a priority scheduler gives access in a round robin fashion to all the highest priority threads, and ignores all other threads*".

## Example

Thread – Priority  
A – 2  
B – 1  
C – 4  
D – 3

Scenario:

I. A holds lock L  
C tries to acquire L (and goes to sleep)

What is the problem?  
How do we solve it?

II. Now, suppose A calls join on B

How do we solve this?

## Mars Pathfinder

Excerpts from

[http://research.microsoft.com/~mbj/Mars\\_Pathfinder/Mars\\_Pathfinder.html](http://research.microsoft.com/~mbj/Mars_Pathfinder/Mars_Pathfinder.html)

- VxWorks realtime embedded systems kernel
- Preemptive priority scheduling (just like Nachos...)

The problem

Pathfinder contained an "information bus", which you can think of as a shared memory area used for passing information between different components of the spacecraft. A bus management task ran frequently with high priority to move certain kinds of data in and out of the information bus. Access to the bus was synchronized with mutual exclusion locks (mutexes).

The meteorological data gathering task ran as an infrequent, low priority thread, and used the information bus to publish its data. When publishing its data, it would acquire a mutex, do writes to the bus, and release the mutex. If an interrupt caused the information bus thread to be scheduled while this mutex was held, and if the information bus thread then attempted to acquire this same mutex in order to retrieve published data, this would cause it to block on the mutex,

waiting until the meteorological thread released the mutex before it could continue. The spacecraft also contained a communications task that ran with medium priority.

Most of the time this combination worked fine. However, very infrequently it was possible for an interrupt to occur that caused the (medium priority) communications task to be scheduled during the short interval while the (high priority) information bus thread was blocked waiting for the (low priority) meteorological data thread. In this case, the long-running communications task, having higher priority than the meteorological task, would prevent it from running, consequently preventing the blocked information bus task from running. After some time had passed, a watchdog timer would go off, notice that the data bus task had not been executed for some time, conclude that something had gone drastically wrong, and initiate a total system reset.

This scenario is a classic case of priority inversion.

### Solution

When created, a VxWorks mutex object accepts a boolean parameter that indicates whether priority inheritance should be performed by the mutex. The mutex in question had been initialized with the parameter off; had it been on, the low-priority meteorological thread would have inherited the priority of the high-priority data bus thread blocked on it while it held the mutex, causing it be scheduled with higher priority than the medium-priority communications task, thus preventing the priority inversion. Once diagnosed, it was clear to the JPL engineers that using priority inheritance would prevent the resets they were seeing.

VxWorks contains a C language interpreter intended to allow developers to type in C expressions and functions to be executed on the fly during system debugging. The JPL engineers fortuitously decided to launch the spacecraft with this feature still enabled. By coding convention, the initialization parameter for the mutex in question (and those for two others which could have caused the same problem) were stored in global variables, whose addresses were in symbol tables also included in the launch software, and available to the C interpreter. A short C program was uploaded to the spacecraft, which when interpreted, changed the values of these variables from FALSE to TRUE. No more system resets occurred.

### Key Issues

- determine in which events the state of the system changes
  - o e.g., owners of locks, waiters in queues
- determine where it makes sense to keep the necessary state (*cache*)
- take care of cascading priorities

Also look at Emil's document:

<http://inst.eecs.berkeley.edu/~cs162-tc/pri>

## More Synchronization

Coke Machine, producers and consumers

### Correctness Constraints

- The machine only holds MAXCOKES cokes
- Consumer must wait for producer to put cokes, if machine empty
- Producer must wait for consumer to take cokes, if machine full
- Only one person can manipulate the machine at once (mutex)

### With Monitors

```
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 before
    wantToTake.wake();
    lock.release();
}

Consumer() {
    Lock.acquire();
    while (numCokes == 0) { //Like a semaphore P()
        wantToTake.sleep(); //lock released!
    }
    numCokes--; //lock reacquired before
    wantToAdd.wake();
    lock.release();
}
```

### With Semaphores

```
Semaphore availableCokes = 0; //consumer constraint
Semaphore emptySlots = MAXCOKES; //producer constraint
Semaphore mutex = 1; //mutual exclusion to the machine

Producer() {
    emptySlots.P(); //check if can add more cokes. If not, wait
    mutex.P(); //wait for exclusive access
    numCokes++;
    mutex.V(); //release lock;
    availableCokes.V(); //tell there is one more coke in the machine
}

Consumer() {
    availableCokes.P(); //check if can get a coke. If not, wait
    mutex.P(); //wait for exclusive access
    numCokes--;
    mutex.V(); //release lock
    emptySlots.V(); //tell one coke was consumed
}
```