

Class Notes Sept 1 2006.

Critical Sections

The basic problem we had with interfering computations on shared variables is that when I have two computations of the following kind, one can be injected inside the other by an interrupt.

```
Load R1,x
Add R1,x
Store R1,x
```

The problem happens when an interrupt occurs between the load and the store. This is known as the critical section of a program. This is a simple example, critical sections can be long and complex. The main idea is that while a critical section of one process is executing, the critical sections of other processes must be prevented from executing. The processes can interleave all they want in other respects, but the critical sections cannot be interleaved.

A critical section is always with respect to a particular variable or set of variables.

For simplicity we assume that a process is structured like this:

```
while (true)
{
    critical section
    remainder section
}
```

The remainder section is just the part of the program that is not the critical section. We assume that the critical section is executed over and over many times. (Not forever, but unpredictably long.) We could have some remainder section before the critical section too, but this really doesn't matter.

To control access to the critical section, we must place some code before the critical section to alert the other processes that a particular process is entering its critical section. We must also add code after the critical section to alert other processes that a particular process is no longer in its critical section, so the whole thing ends up looking like this:

```
while (true)
{
    entry section
    critical section
    exit section
    remainder section
}
```

Whatever we put in the entry and exit sections must have the following properties.

1. Mutual exclusion – no two critical sections interleaving their executions.
2. Progress – no stupid solutions, you can't get locked out except by another process.
3. Bounded waiting – you won't be forced to wait forever because of timing issues

Number 3 is normally not present in most solutions, because it is normally not an issue in real systems. In specialized cases, this can change.

Peterson's Solution

We can do this with raw code. We need the following assumption however. If two processes attempt to assign a value to a variable, one of them will succeed. The assigned value will not be some combination of the two values, but will be distinctly one value or the other.

```
int turn;
boolean flag[2];

//entry section, process 1:

flag[1] = true;
turn = 2;
while (flag[2] && turn == 2);

    critical section

// exit section process 1

flag[i] = false;
```

We need both of these things. If we use just the turn variable, we will need to swap between two processes even if only one process is running. If we use just the flag variable, both flags can get set simultaneously.

There are other more complex solutions that work for n processes, but the code uses n as one of its parameters. These solutions are bizarrely complex, and extremely difficult to code correctly. Nevertheless, it can be done. Peterson's solution can also be extended to multiple processes.

Synchronization hardware.

Doing mutual exclusion in code teaches us one thing. We don't want to do it in code. Therefore hardware mechanisms have been invented to do this job for us. The most common of these is test and set.

Test and set performs a hardware interlock on the memory module it addresses. It performs a read-test-and-write operation while locking out any other processor from the memory module. This permits the instruction to execute *atomically*.

The test and set addresses a single memory word. It sets this word to TRUE, and returns the previous value. This is done atomically, so no two processors can see the original value and set the new value.

The test and set instruction can be used as follows to enforce mutual exclusion. This is an n-process solution.

```
shared boolean Flag = false;
```

```
while (TestAndSet(Flag));  
    critical section  
Flag = false;
```