Systems Programming and Operating Systems	Indupprakas Kodukula
CS414, Summer 1996	Class#5, 7/15/96

1 Class #5

1.1 Baker's algorithm for N processes

Next, we'll study a particular algorithm that achieves mutual exclusion in the presence of $N > 2$ processes. The problem with the algorithm in the previous section is that *self* and *other* have well defined meanings for two processes. But, it is hard for them to be easily generalized to $N > 2$ processes. For the following discussion, we assume that each of the N processes under consideration is assigned a unique id.

The idea behind the baker's algorithm is following: Mutual exclusion can be thought of as an ordering process. Two processes cannot be in the critical section at the same time. In other words, all the processors are **ordered** according to how they execute the critical section. So, a simple approach is the following: The processes can be handed out a series of tokens of non decreasing face value when they desire to enter the critical section. Processes are allowed to enter the critical section in the same order as the order of the tokens they receive. However, since the handing out of tokens is done without mutual exclusion, it is quite possible that several different processes land up with the same token value. In this case, the unique process id is used for arbitration purposes.

```
<enter> choosing[self] = true;
        number[self] = max (number[0],...,number[n-1]) + 1;
        choosing[self] = false;
        i = self;
        for j = 0 to n-1 do
          while (choosing [j]) do skip;
          while ((number[j]≠0) & (number[j],j)<(number[i],i))
            do skip;
CS
<exit> number[self] = 0;
```

1.2 Hardware support

The complexity of the previous solution is due to the fact that while memory read and write operations are themselves atomic, a sequence of read and write operations is not. So, it is reasonable to expect that hardware support might make a software solution easier. In this section, we'll examine two hardware features that make synchronization easier.

1.3 Test-and-set-lock

The test and set instruction can be thought of as a function that atomically sets its argument to true and returns the old value.

```
TS(l): <TS,l := l,true>
```

The key feature of this instruction is that both these sequence of operations are executed **indivisibly**. After the test-and-set instruction is started, it is **never** interrupted until its completion. Now, let's implement a critical section with this hardware instruction

```
<enter> while TS (l) do skip;
CS
<exit> lock = false;
```

l is initialized to false. The critical section property is maintained because of the following reason: If l is set to false, when a process gets out of a critical section, let's assume that there are several processes waiting on the value of the lock l. By definition, precisely one of them will be able to set l to true and will see l to be false. After one process has passed into the critical section, all the other processes will again keep looping until this process has exited from the critical section.

1.4 Atomic swap

In this section, we consider the atomic swap instruction which swaps its argument atomically:

```
SWAP(a,b): <a,b := b,a>
```

The values of a and b are swapped simultaneously. Again, as is the case of the Test-and-set lock instruction, the entire instruction is completed without any interruption. We can use the atomic swap instruction to provide a solution to the critical section problem as follows.

```
<enter> temp = true;
        repeat
          SWAP (lock, local)
        until local = false;
CS
<exit>  lock = false;
```

1.5 Bounded Wait

One of the problems with the algorithms we've examined in this section is that they do not guarentee any kind of bounded time for execution. In other words, they do not guarentee that a given process does not starve. To prevent startvation with the Test-and-set-lock algorithm, we need to use the following algorithm:

```
<enter> waiting[i] = true;
        local = false;
        while (waiting[i] && local) do local = test-and-set (lock);
        waiting[i] = false;
CS
<exit> j = (i⊕+1) mod n
       while (j≠i) and (!waiting[j]) j = (j⊕+1)mod n
       if (j == i)
         lock = false;
       else
         waiting[j] = false;
```

Why does this prevent starvation? Basically, when the current process comes out of the critical section, *lock* is still set to true. This is because whenever *lock* is false, there will not be any process in the critical section. So, when a process comes out of a critical section, if there are any processes waiting to enter the critical section, then the "first" such process is let into the critical section by the process that just exited. If a process exits the critical section and finds that no one wants to enter it, then it resets lock to false. The first process that acquires lock then proceeds into the critical section.