| | |
|---|---|
| **Systems Programming and Operating Systems** | **Induprakas Kodukula** |
| **CS414, Summer 1996** | **Class#6, 7/16/96** |

# 1 Class #6

In this class, we'll study higher level objects that would help us write better (less complex) synchronization code.

## 1.1 Semaphores

A *semaphore* is a non negative integer variable with two operations defined on it:

- P - (wait) Proberen (dutch - to test or to probe)
- V - (signal) Verhogen (dutch - to signal or to increment)

We need semaphores because of two reasons:

Solutions are incredibly complex and mistifying otherwise

The other solutions require busy waiting

### 1.1.1 P & V

Informally, the P operation on a semaphore has the following effect: If the value of a semaphore is greater than 0, then the value of the semaphore is decremented and the P operation returns. Otherwise, the P operation blocks until the value of the semaphore exceeds zero. The V operation increments the value of the semaphore. The following code can be used to represent these operation.

```
P(S) : while (S ≤ 0) do skip;
       S := S - 1;
V(S) : S := S + 1
```

There are several restrictions on the atomicity requirements of the P and V operations:

Modification of S in $P$ and $V$ must be atomic.

The testing of S in $P$ (S $\leq$ 0) and the its possible modification must also happen without interruption.

Processes can use the $P$ and $V$ operations to solve the critical section problem as follows:

```
<enter> P (mutex)
CS
<exit>  V (mutex)
```

*lock* is a variable of semaphore type. The semaphore *mutex* is initialized to the value 1 in this case. As one can see, this code is much simpler than all the loops and local variables and other stuff that we had to deal with before.

We can use semaphores to get a safe and simple solution for the bounded buffer problem that we mentioned earlier. Recall that the bounded buffer problem has a buffer of a fixed number of elements. A producer process puts elements into the buffer and a consumer process takes elements out of the buffer. If the buffer is full, the producer blocks. Similarly, the consumer blocks if the buffer is empty. Here is how we can structure each of these processes. We'll use 3 semaphores. As we've already seen before, there is a critical section in the bounded buffer problem which needs to be accessed carefully. So, we'll use a semaphore *mutex* for that. We'll use two additional semaphores: *empty* will be used to indicate how many entries in the buffer are left unfilled. Similarly, *filled* will be used to indicate how many entries in the buffer have been filled. The reason is that if there are no slots empty, the producer must block. Similarly, if there is no data, the consumer must block. *mutex* is initialized to 1, *empty* to n and *filled* is initialized to 0. The producer and consumer processes are structured as below:

```
Producer Process                            Consumer Process

<create data element>                       <consume data element>
P(mutex) P(empty)                           P(mutex) P(filled)
CS                                          CS
V(mutex) V(filled)                          V(mutex) V(empty)
```

### 1.2 Implementation Notes

In the naive implementation of $P$ and $V$ operations as described above, we can immediately see that there is a concern regarding efficiency. The main disdvantage of all the solutions for mutual exclusion that we have seen so far is that the processes that are waiting to enter the critical section do "busy waiting". In other words, a process that is just waiting to enter the critical section keeps reading the value of some location until some condition of termination. This has two disadvantages.

- CPU time is wasted. In a multiprogramming environment, it is a lot of overhead to replace one running CPU process by another. If all that the new process does is test some condition till it evaluates to false, then it is a real waste of CPU resources.

- On a modern machine, if a particular location is the subject of a large number of reads and writes, the bandwidth/memory traffic is quite unacceptable.

The implication for our semaphore implementation is the following: while the $P$ and $V$ operations could be implemented as busy waiting, it is far more efficient to use a *sleep-wakeup* model. A semaphore maintains a list of *sleeping* processes that are waiting for $P$ operations to succeed. If a process can execute a $P$ operation successfully, it simply proceeds. If it needs to block, it is added to the list of sleeping processes. When a $V$ operations is performed on the semaphore, there are two possibilities. If there are no processes in the list of sleeping processes for this semaphore, then the $V$ operation simply increments the value of the semaphore. If there are any processes in the list of sleeping processes maintained by the semaphore, then a $V$ operation sends a *wakeup* call to one of these processes. It is possible to make the process scheduler aware of when a process is sleeping on a condition and when it has been woken up. For the sake of efficiency, a sleeping process should not be scheduled. A process is added back into the set of schedulable processes when it is woken up.