Operating System Based Solution of the Critical Section Problem

Operating system provides special functionality for process coordination. The idea is based upon the fundamental principle "Two or more processes can cooperate by means of simple signals, such that a process can be forced to stop at a specified place until it has received a specific signal. Any complex coordination requirement can be satisfied by the appropriate structure of signals".

Semaphores

For signaling, special variables called semaphores are defined by the operating system. To transmit a signal via semaphore ***s***, a process executes the primitive semSignal(***s***). To receive a signal via semaphore ***s***, a process executes the primitive semWait(***s***); if the corresponding signal has not yet been transmitted, the process is put in wait state until the transmission takes place.

To put semaphores in practice in computing environments and have the desired effect, semaphore can be viewed as a variable that has an integer variable, initialized to a non-negative value upon which only two operations are defined:

1. The semWait operation decrements the semaphore value. If the value becomes negative, then the process executing the semWait is blocked. Otherwise, the process continues execution.
2. The semSignal operation increments the semaphore value. If the resulting value is less than or equal to zero, then a process blocked by a semWait operation, if any, is unblocked.

Other than these two operations, there is no way to inspect or manipulate semaphores. semSignal and semWait are indivisible (atomic) units, which cannot be interrupted and always run to completion.

Semaphores can be used to control access to a given resource consisting of a finite number of instances and in that case the semaphore is initialized to number of resources available. Each process that wishes to use a resource performs a semWait operation on the semaphore (decrementing the value). When a process releases a resource, it performs a semSignal operation (incrementing value). When the value of the semaphore goes to '0', all resources are being used. After that, processes that wish to use a resource will block until the count becomes greater than '0'.

For using semaphores in computing environments, we define semaphores as listed below:

```
typedef struct {
        int count;
        queuetype queue;
} semaphore;

semWait(semaphore S)
{
        S.count--;
        if (S.count < 0) {
                add this process to S.queue ;
                 block();    /*  block this process  */;
        }
}
```

```
semSignal(semaphore S)
{
        S.count++;
        if (S.count<= 0) {
                remove a process P from S.queue;
                wakeup (P);  /* blocked process is unblocked */
        }
}
```

Each semaphore has an integer count and a queue of processes. When a process must wait on a semaphore, it is added to the list of processes waiting on that semaphore.
A semSignal() operation removes one process from the list of waiting processes.

Semaphores are used for synchronization of cooperating processes and for critical section problem. We will describe usage of semaphores through examples, so would like to simply define semWait as wait, and semSignal as signal with simplified code as follows:

```
wait(S):  S = S – 1;
          if  S < 0 {
                put calling process in queue;
                block();
          }
signal(S):  S = S + 1;
          if S ≤  0 {
                remove a process P from queue;
                wakeup(P);
          }
```

Semaphores to Solve Synchronization Problem
Semaphores can be used to solve various synchronization problems. Requirement that one process stop to wait to pass a point until another process sends a signal. Waiting point represents a condition that must be true for subsequent execution to be valid. Signal represents the event of the condition becoming true. We can clarify by considering two concurrently running processes: P1 with a statement S1 and P2 with a statement S2. Suppose we require that S2 be executed only after S1 has completed. We can implement this scheme by letting P1 and P2 share a common semaphore *Synch*, initialized to ‘0’. In process P1, we insert the statement signal(*Synch*); after S1 and in process P2, we insert the statement wait(Synch); before S2. Because *Synch* is initialized to ‘0’, P2 will execute S2 only after P1 has invoked signal (*Synch*), which is after statement S1 has been executed.
An example of synchronization condition being satisfied by using semaphore:

Two process P1() and P2() executing statements S1-S7 and S11-S17 respectively. P1 and P2 cooperating processes, where P1 generates some information while executing S3. This information is required by P2 to execute statement S12. P1 and P2 are executed concurrently, so their execution must be synchronized to ensure that S12 is executed after execution of S3. We define syn semaphore and initialized it to ‘0’. Wait operation on syn semaphore is inserted before S12 and signal operation on syn semaphore is inserted after statement S3.

S12 of P2 should be executed after S3 of P1

```
Semaphore syn = 0;
Process-P1( )              Process-P2( )
S1                             S11
S2                             wait(syn);
S3                             S12
signal(syn);                   S13
S4                             S14
S5                             S15
S6                             S16
S7                             S17
```

If P2 is executed first, then wait (syn) will set values of syn to '-1' and is blocked. Now P1 is executed and signal(syn) will set value of sys to '0', wake the process P1 and P1 is placed in ready list.
If P1 is executed first, signal(syn) will set value of syn to '1' means signaling that event has occurred. Now P2 is executed, when wait (syn) is executed, it will set value of sys to '0' and continue its execution. Simply using a semaphore, synchronization of P1 and P2 is accomplished in whatever sequence P1 and P2 are executed.

Critical Section Problem (Mutual Exclusion)

Usage of semaphore provides a simple solution to the mutual exclusion of a critical section problem. For n processes which need access to the same resource, i.e., each process has a critical section. We use mutex semaphore, initialized to '1'. In each process, a wait(mutex) is executed just before its critical section as listed:

```
share (mutex = 1) semaphore
P (i)
while(condition) {
            wait(mutex);
              <Critical Section>
            signal (mutex);
  remaining Section
}
```

First process that executes wait(mutex) will be able to enter the critical section immediately, setting the value of mutex to '0'. Any other process attempting to enter the critical section will find it busy and will be blocked, setting the value of mutex to '-1'. Any number of processes may attempt entry; each such unsuccessful attempt results in further decrement of the value of mutex. When the process that initially entered its critical section departs, value of mutex is incremented and one of the blocked processes (if any) is removed from the queue of blocked processes associated with the semaphore and is now in ready state. When it is next scheduled by the OS, it may enter the critical section.

Classic Problems of Synchronization

So far, we have discussed three approaches to solve synchronization problems. To verify and test a proposed synchronization scheme, several classical synchronization problems are used. We will describe a few of these problems (producer-consumer, reader-writer, and dining philosophers) and how semaphores are used to resolve issues of mutual exclusion and synchronization in concurrent processing environments.

Producer Consumer Problem

Producer-Consumer problem is one of the most common problems faced in concurrent processing. There are one or more producers generating some type of data (characters records, files) and placing these in a buffer. A consumer that is taking items out of the buffer one at a time. The system is to be disciplined to prevent the overlap of buffer operations. That is, only one producer or consumer may access the buffer at any one time. The problem is to make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.
Producer-Consumer problem is defined algorithmically as follows:

```
#define BUFFER SIZE
typedef struct {
. . .
} item;
item buffer[BUFFER SIZE];
int  in   = 0;
int out = 0;
count  = 0;
```

The shared buffer is implemented as a circular array with two logical pointers: *in* and *out*. The variable *in* points to the next free position in the buffer; *out* points to the first full position in the buffer. To keep count of buffer locations filled, *count* is used. The buffer is empty when *count* = 0; the buffer is full when count = BUFFER SIZE). The producer process has a local variable next-produced in which the new item to be produced is stored. The consumer process has a local variable next-consumed in which the item to be consumed is stored.
The code for the producer process is listed below:

```
while (true) {
/* produce an item in next-produced */
while (count == BUFFER SIZE)            /* buffer is full */
        ; /* do nothing */
buffer[in] = next-produced;
in = (in + 1) % BUFFER SIZE;
}
```

The code for the consumer process is listed below:

```
while (true) {
while (count == 0)                  /* buffer is empty  */
        ; /* do nothing */
next-consumed = buffer[out];
out = (out + 1) % BUFFER SIZE;
/* consume the item in next-consumed */
}
```

As, *buffer*, *in*, *out*, and *count* are shared, by producers and consumers, access to these resources must be exclusive to maintain buffer data integrity. To make sure consumer process is not accessing empty buffer, and producer process is not overwriting full buffer, producer and consumer processes must be synchronized. All synchronizations involve waiting.

- Consumer must wait for producer to fill buffers, if none filled.
- Producer must wait for consumer to empty buffers, if all full.

We can use following semaphores for exclusive access and to synchronize producer and consumer processes.

```
semaphore mutex = 1;
semaphore Empty = N;
semaphore Full = 0
```

We assume that the pool consists of 'N' buffer slots, each capable of holding one item. The mutex semaphore ensure exclusive access to the buffer pool and is initialized to '1". The Empty and Full semaphores count the number of empty and full buffer slots respectively. Initially all buffer slots are free, so the semaphore Empty is initialized to the value 'N'; the semaphore Full is initialized to value '0'. Producer and Consumer process has local variable item-p and item-c respectively of type item.
These semaphores are embedded within code of producer and consumer process as listed below:

```
Producer-Process ()
        item item-p;
        while (true) {
          produce an item-p;  /* produce an item in item-p  */
          wait(Empty);
             wait(mutex);
                buffer[in] = item-p;
                in = (in + 1) % N;
             signal(mutex);
           signal(Full);
        }

Consumer-Process ()
        item item-c;
        while (true) {
          wait(Full);
              wait(mutex);
                item-c = buffer[out];
                out = (out + 1) % N;
             signal(mutex);
          signal(Empty);
        consume item-c; /* consume the item in item-c  */
        }
```

You can view the code of producer and consumer processes and notice that mutex semaphore ensures that at any given time either a producer or a consumer is accessing shared buffer (producer filling buffer slot and consumer taking from buffer slot). Value of Full semaphore is initialized to '0'since buffer has empty slots. If a consumer process arrives, it will execute wait() operation on Full semaphore, and after decrementing the value of Full semaphore will be blocked. Subsequent consumer processes will be blocked on Full semaphore. A blocked consumer process will be unblocked by first producer, you can view producer code, and notice that when a producer arrives, it executes wait() operation on

Empty semaphore, since value of Empty semaphore is not zero, it will decrement its value and proceed further, it executes wait() operation on mutex semaphore (which has values ‘1’) and will set its values to ‘0’, before filling buffer slot, to make sure no other process is accessing buffer slots. After filling buffer slot, it executes, signal operation on mutex semaphore (setting its values to ‘1’). Now producer process has filled the buffer slot, it can signal about it (a buffer slot is just filled) and for that it executes signal() operation on Full semaphore. Value of Full semaphore has been set to ‘-1’ by the consumer process, so value will be incremented (set to ‘0’) and as a result the consumer process which was waiting is unblocked. If there was no consumer process waiting on Full semaphore (i.e. value of Full semaphore is ‘0’) then value of Full semaphore is set to ‘1’, indicating one buffer is filled. You can trace different sequences of producer and consumer process to see for yourself that solution of producer-consumer problem with bounded buffer using semaphores works.

Producer-Consumer problem with bounded buffer discussed above can easily be modified for an unbounded buffer. First we do not keep a circular buffer, second *in*, and *out* buffer position pointer for producer and consumer will be simply incremented. In case of an unbounded buffer, producer process never waits for buffer space availability. To block producer process, wait(Empty) operation is used, so we can remove this operation from producer process code. To signal, a free buffer space, consumer process after getting buffer value executes signal(Empty) operation. In case of unbounded buffer, no need to keep track of empty buffer space, so we can remove this operation from consumer process. Operations on Empty semaphore are removed from producer and consumer processes, so no need to have Empty semaphore. With these modification, solution of producer-consumer problem with bounded buffer using semaphores works as solution of producer-consumer problem with unbounded buffer.