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, *counter* is used. The buffer is empty when *counter* == 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 *counter* 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.