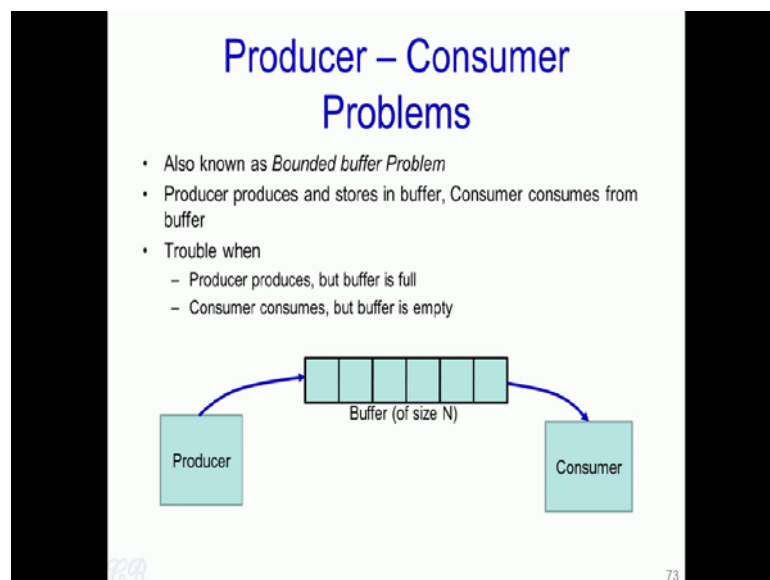


Introduction to Operating Systems
Prof. Chester Rebeiro
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Week – 06
Lecture – 29
Semaphores

Hello. In this video, we will look at another synchronization primitive known as Semaphores. As usual we will start with the motivating example and then we will show the Application of Semaphores.

(Refer Slide Time: 00:35)



Let us start with the very popular example known as the Producer-Consumer Problem. This is also known as the Bounded buffer Problem. Essentially, what we have here are two processors; one process is known as the Producer and the other process is known as the Consumer. Now the producer and the consumer share a bounded buffer. This is a normal buffer and it has size of N that is it has N data elements which can be stored in it.

In this particular case for example, there are 6 elements that can be stored in the buffer. Now the producer produces data. So for instance, it could be data acquisition module which collects data such as the temperature, pressure and so on, so this data is pushed

into the buffer. Now on the other side, the consumer takes from the buffer and then processes the data. For example, this consumer process could perhaps compute some analytics on the producer data.

So, everything would work quite well, that is the producer produces data, a puts it on the buffer and on the other side the consumer takes from the buffer and begins to consume the data. For instance computes something with the data. The trouble will occur when for instance the consumer is very slow compare to the producer. In such a case the producer will produce quite a bit of data at a faster rate compare to the consumer and therefore very quickly the buffer will be full.

So, what does the producer do in next? And the other problem could occur where the consumer is very fast compared to the producer. And therefore, it could very quickly consume all the data in the buffer, and resulting in a buffer which is empty. So, what should the consumer do next? This requires a synchronization mechanism between the producer and the consumer. Essentially, when the buffer is empty the consumer should wait until the producer fills in data into the buffer. Similarly, when the buffer is full the producer has to wait until the consumer takes out data from the buffer.

(Refer Slide Time: 03:25)

Producer-Consumer Code

Buffer of size N
int count=0;
Mutex mutex, empty, full;

```
1 void producer(){
2   while(TRUE){
3     item = produce_item();
4     if (count == N) sleep(empty);
5     lock(mutex);
6     insert_item(item); // into buffer
7     count++;
8     unlock(mutex);
9     if (count == 1) wakeup(full);
10  }
}
```

```
1 void consumer(){
2   while(TRUE){
3     if (count == 0) sleep(full);
4     lock(mutex);
5     item = remove_item(); // from buffer
6     count--;
7     unlock(mutex);
8     if (count == N-1) wakeup(empty);
9     consume_item(item);
10  }
}
```

74

So this is the general producer and consume code, and we are trying to solve the producer-consumer problem by using the Mutexes, Essentially, we are using 3 mutexes; empty, full, and a mutex just called a mutex.

So, the producer code essentially would produce an item insert the item into the buffer and increment the count, count plus plus. While, the consumer code would remove the item decrement a count and then consume the item. Now in order to take care of the troubled situation, that is when the buffer is full or the buffer is empty we are using these mutexes; empty and full. Essentially, before inserting the item the producer would check if count equal to N, that is it is going to check if the buffer is full. And if the buffer is full, it is going to sleep on this particular mutex called Empty.

On the other side, the consumer would check if count equal to N minus 1 which means that it has just removed one element from a full buffer. So if this is so then it is going to wakeup empty. This wakeup is a signal to the producer to wake up from it is sleep and then the producer can insert the item into the buffer and increment a count. On the other hand, if the consumer finds that the buffer is empty that is the count equal to 0 then it is going to sleep on this particular mutex called Full. So, it will block on this mutex until it gets a wakeup from the producer.

Essentially, if the producer finds that when he insert with the item and incremented count that there is exactly 1 item present in the buffer then he will send a wakeup signal to the consumer, so wakeup full. And therefore, this wakeup full will cause the consumer to unblock and put it back into the ready q and it would allow the consumer to execute, and remove that item and then consume that item. After he removes the item the counts goes back to 0 and search this case. Now in addition to this empty and full mutexes there is also the third mutex which is used. So, this mutex is essentially used to protect or synchronize access to the buffer. So, before inserting an item and incrementing the count, the producer needs to lock the mutex and unlocking is done after the item is pushed and count incremented.

On the other side, before the buffer is accessed to remove item and also count is decrementing. Essentially you notice that count and the buffer are shared among these

two a processes that is the producer and the consumer. And therefore, this mutex will help synchronize access to the buffer and to the count value, so this solution seems to work fine that is with the 3 mutexes. So, while this scheme seems fine we will show that under a certain condition the producer and the consumer will block infinitely without any progress.

So that condition is based on the fact that this particular line, if count equal to equal to 0 actually comprises of two steps which are non atomic. The first step is that the count value which is stored in memory will be loaded into a register in the processor, and the second step is when the register value is checked to be 0 or not. Let us look at the problem that could occur because this particular execution or this particular statement is non-atomic

(Refer Slide Time: 08:12)

Lost Wakeups

- Consider the following context of instructions
- Assume buffer is initially empty

```

3 read count value // count ← 0
3 item = produce_item();
5 lock(mutex);
6 insert_item(item); // into buffer
7 count++; // count = 1
8 unlock(mutex)
9 test (count == 1) // yes
9 signal(full);
----- context switch -----
3 test (count == 0) // yes
3 wait();
  
```

Note, the wakeup is lost.
 Consumer waits even though buffer is not empty.
 Eventually producer and consumer will wait infinitely

consumer
still uses the old value of count (ie 0)

75

So let us say, that the consumer starts executing first and it is starts executing with an empty buffer. It reaches the value of count from memory location into a register, since we are assuming that this is the initial state so the value of count that is loaded into the register would be 0. Let us then say that there is a context which that occurred and the producer has executed. Now the producer produces an item, increments the count to 1, and then inserts the item into the buffer. After it executes let us say that there is a context

switch again, as a result the consumer continues to execute from where it had stop that is from this point.

Now we know that it has already loaded the register previously with the value of 0. Now it is going to test whether count equal to 0, which is true in this case and therefore the consumer is going to wait. Essentially, it is going to wait till it receives signal from the producer. However, the actual value of count is 1, because the producer has pushed an item into the buffer, and thus we see there is lost wakeup that occurs. The consumer has missed wakeup signal which the producer has sent. Now there is nothing stopping the producer from pushing more items into the buffer.

So, eventually the entire buffer is full and the producer will then wait for the consumer to remove some item. However, this will not occur because the consumer itself is waiting. Thus, we have a producer waiting for the consumer to remove an item, while the consumer is also waiting because it has missed the wakeup. Thus, we eventually reach particular state where both producer and consumer will wait infinitely. We see that using three mutexes will not solve the producer-consumer problem.

(Refer Slide Time: 10:39)



Semaphores

- Proposed by Dijkstra in 1965
- Functions `down` and `up` must be atomic
- `down` also called `P` (Proberen Dutch for try)
- `up` also called `V` (Verhogen, Dutch form make higher)
- Can have different variants
 - Such as blocking, non-blocking
- If `S` is initially set to 1,
 - Blocking semaphore similar to a Mutex
 - Non-blocking semaphore similar to a spinlock

```
void down(int *S){
    while( *S <= 0);
    *S--;
}

void up(int *S){
    *S++;
}
```

76

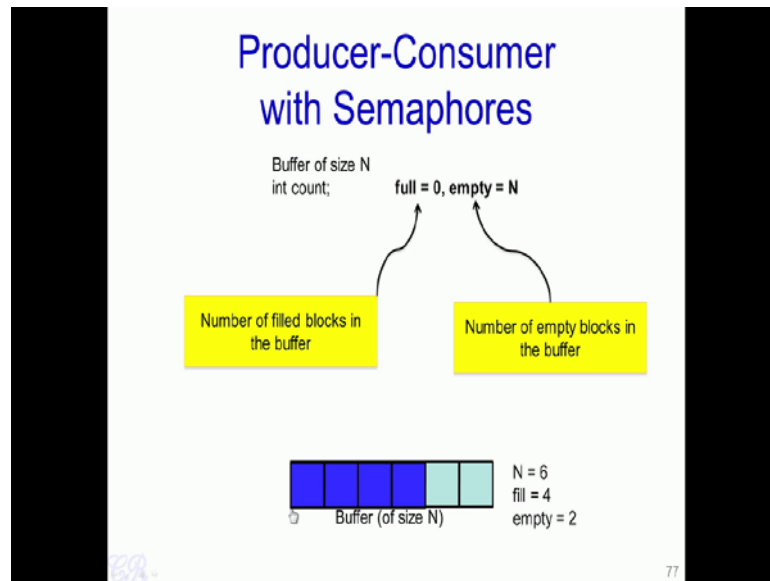
Let us look at another primitive known as Semaphores. This semaphore is

synchronization primitive it was proposed by Dijkstra in 1965. And semaphores are implemented with 2 functions called Down and Up, which we assume is atomic. These are the functions and thus requirement is that both these functions need to be atomic. There is shared memory location which is termed as S and in the down function the while loop will test whether S is less than equal to 0.

So, as long as S has a value which is less than equal to 0 this particular loop will execute. When S takes a value which is greater than 0, then the loop would break and the value of that memory location S would be decremented by 1. In the up function which is also atomic the value of the memory location S is incremented by 1. So, the down and up functions are sometimes called as the P and V functions respectively from there Dutch names. And we could also have two different variants of this semaphores, we could have a blocking semaphore and a non-blocking semaphore as well. So, a non-blocking semaphore is shown over here, essentially it is a while loop which is resulting in a busy waiting much like a spinlock.

On the other hand we can make a small modification and have a blocking semaphore, where this particular statement will result in the process going to block state, while signal from the up would wake up the process. If the values of s were initially set to 1 then a blocking semaphore is similar to a mutex, while a non-blocking semaphore is similar to a spinlock. So now, let us see how we produce the semaphore to solve the producer-consumer problem.

(Refer Slide Time: 12:55)



In order to solve the problem we require two semaphores; one is known as full. So when I say two semaphores it means two memory locations which we specify by this S over here. We have two memory locations or two semaphores full and empty. So, full is given the initial value 0, while empty is given the initial value N, where N here is the size of the buffer. The semaphore full indicates the number of filled blocks in the buffer, while the semaphore empty would indicate the number of empty blocks in the buffer.

In this particular case, where N equal to 6, fill will have a value of 4 because there are 4 filled blocks and empty will have a value of 2 because there are 2 empty blocks. So, the initial states just before the start of execution of the producer and consumer will have full equal to 0 and empty equal to N, because there is no data items in the buffer; essentially, because the buffer is empty.

(Refer Slide Time: 14:13)

Producer-Consumer with Semaphores

full = 0, empty = N

```
void producer(){
  while(TRUE){
    item = produce_item();
    down(empty);

    insert_item(item); // into buffer

    up(full);
  }
}
```

N = 6
fill = 4
empty = 2

Buffer (of size N)

78

So let see, how these semaphores are used. Let us look at the producer, so the producer produces an item and we will take this particular example where a fill is 4 and empty is 2, and then when the item is produced it invokes the down semaphore. The down semaphore, as we have seen will down the empty semaphore, so empty will go from 2 to 1. So, this is an atomic operation.

(Refer Slide Time: 14:52)

Producer-Consumer with Semaphores

full = 0, empty = N

```
void producer(){
  while(TRUE){
    item = produce_item();
    down(empty);

    insert_item(item); // into buffer

    up(full);
  }
}
```

N = 6
fill = 5
empty = 1

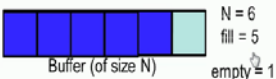
Buffer (of size N)

78

Then the next step in the producer used to insert an item, so the new item gets inserted into the buffer, and then there is an up full that is the semaphore full will get a value of 5.

(Refer Slide Time: 15:08)

Producer-Consumer with Semaphores



```
void consumer(){
while(TRUE){
→ down(full);

item = remove_item(); // from buffer

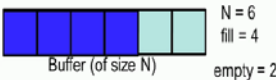
up(empty);
consume_item(item);
}
}
```

79

Similarly, in the consumer part is as follows. First there is the down full, so, the value of full will go from 5 to 4 as seen here.

(Refer Slide Time: 15:25)

Producer-Consumer with Semaphores



```
void consumer(){
while(TRUE){
down(full);

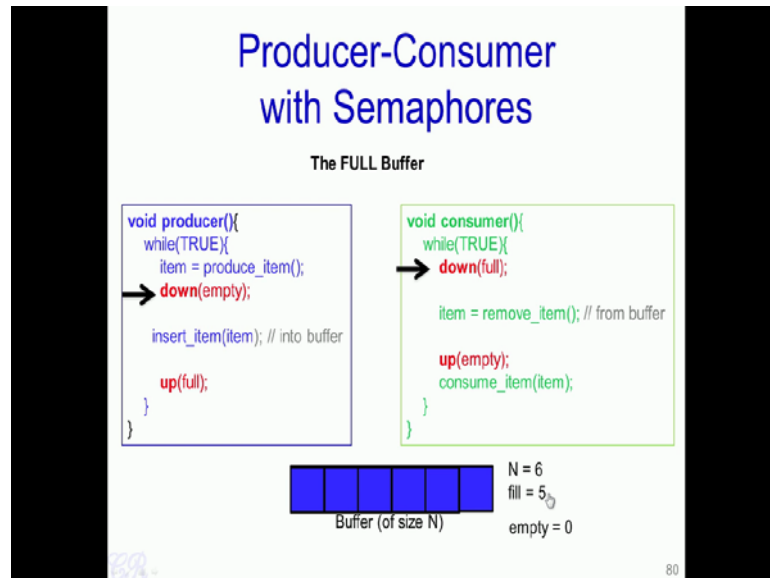
item = remove_item(); // from buffer

up(empty);
→ consume_item(item);
}
}
```

79

Then an item is removed from the buffer and the value of empty is set to up. So, up the value of empty will become 3, and then the consumer will consume the particular item.

(Refer Slide Time: 15:41)



Now let us see both the producer and consumer and the case of a full buffer. We have, let us assume that the buffer is full, so in such a case the full value has 6 which are equal to N, while the empty has a value of 0 indicating that there are no empty blocks in the buffer, and full of 6 indicates that there are 6 full blocks in the buffer. The producer as usual will produce an item and then it will down empty. Now you see that empty has a value of 0. If you go back to the down function of the semaphore it would cause the while statement to keep executing continuously. So, the producer would be blocked or waiting on this particular down semaphore.

Now, after a while when the consumer begins to execute, so it will execute the down full; as a result, it is going to consume one particular element, so it is going to decrement the value of full as will be seeing an over here, so full goes to 5. And then it is going to remove an item and it is going to up empty.

(Refer Slide Time: 17:00)

Producer-Consumer with Semaphores

The FULL Buffer

```
void producer(){  
  while(TRUE){  
    item = produce_item();  
    → down(empty);  
  
    insert_item(item); // into buffer  
  
    up(full);  
  }  
}
```

```
void consumer(){  
  while(TRUE){  
    down(full);  
  
    item = remove_item(); // from buffer  
  
    up(empty);  
    → consume_item(item);  
  }  
}
```

N = 6
fill = 5
empty = 1

80

Now empty is set to 1, and then of course it is going to consume the item. Now setting empty to 1 would result in the loop in the semaphore down to wakeup or for the loop in the semaphore to complete to break, and then the producer will set empty back to 0 and insert the item into the buffer. Then the full value is set to 6 yet again.

(Refer Slide Time: 17:31)

Producer-Consumer with Semaphores

The FULL Buffer

```
void producer(){  
  while(TRUE){  
    item = produce_item();  
    down(empty);  
  
    insert_item(item); // into buffer  
  
    up(full);  
  }  
}
```

```
void consumer(){  
  while(TRUE){  
    down(full);  
  
    item = remove_item(); // from buffer  
  
    up(empty);  
    consume_item(item);  
  }  
}
```

N = 6
fill = 6
empty = 0

81

In this week, the semaphores full and empty are used to solve the producer-consumer problem when the buffer is full.

(Refer Slide Time: 17:49)

Producer-Consumer with Semaphores

The Empty Buffer

```
void producer(){
  while(TRUE){
    item = produce_item();
    down(empty);

    insert_item(item); // into buffer

    up(full);
  }
}
```

```
void consumer(){
  while(TRUE){
    down(full);

    item = remove_item(); // from buffer

    up(empty);
    consume_item(item);
  }
}
```

Buffer (of size N)

N = 6
fill = 0
empty = 6

82

Similar analysis can be made when the buffer is empty. In such a case the values of full and empty are 0 and 6 respectively.

(Refer Slide Time: 18:03)

Producer-Consumer with Semaphores

Serializing Access to the Buffer

```
void producer(){
  while(TRUE){
    item = produce_item();
    down(empty);
    lock(mutex);
    insert_item(item); // into buffer
    unlock(mutex)
    up(full);
  }
}
```

```
void consumer(){
  while(TRUE){
    down(full);
    lock(mutex);
    item = remove_item(); // from buffer
    unlock(mutex)
    up(empty);
    consume_item(item);
  }
}
```

Buffer (of size N)

N = 6
fill = 3
empty = 3

83

So, one thing we have not taken care about so far in the producer and consumer code is that, we are not synchronizing access to this particular buffer. As a result it could be possible that the producer may be inserting item into the buffer and at exactly the same time the consumer may be removing an item from the buffer. In order to prevent such a thing to occur we use a mutex to synchronize access into a buffer.

Before accessing this particular buffer or the producer as well as the consumer would need to lock the mutex, and after accessing the buffer we unlock mutex needs to be invoked. As a result of this mutex we can be guaranteed that only one of these two processes are executing in this particular critical section, that is are accessing the buffer at any given instant of time.

So, now there are several other variants of the producer-consumer problem, and there are various schemes in which semaphores could be utilized to solve these various problems. But, for this particular course we will stop with this particular example.

Thank you.