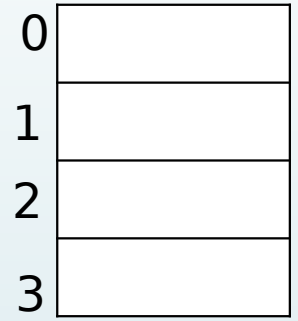
OPERATING SYSTEMS

Module3_Part1

Textbook : Operating Systems Concepts by Silberschatz

Producer consumer problem using shared memory

```
//shared data
Int item,in,out;
in=0;out=0;
Int
BUFFER_SIZE=4
```



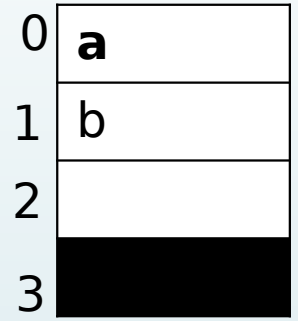

```
//producer process
while (true) {
        /* produce an item in next produced
*/
        while (((in + 1) % BUFFER_SIZE ) ==
out)
                ; /* do nothing */
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
}
```

```
//consumer process
while (true) {
        while (in == out)
                ; /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) %
BUFFER_SIZE;
}
```

Producer consumer problem using shared memory

```
//shared data
Int
n=4,item,in,out;
in=0;out=0;
Int
BUFFER_SIZE=4
```

in=2



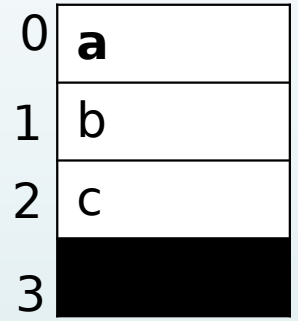

```
//producer process
while (true) {
        /* produce an item in next produced
*/
        while (((in + 1) % BUFFER_SIZE ) ==
out)
                ; /* do nothing */
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
}
```

```
//consumer process
while (true) {
        while (in == out)
                ; /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) %
BUFFER_SIZE;
}
```

Producer consumer problem using shared memory

```
//shared data
Int
n=4,item,in,out;
in=0;out=0;
Int
BUFFER_SIZE=4
```

in=3 producer waits




```
//producer process
while (true) {
        /* produce an item in next produced
*/
        while (((in + 1) % BUFFER_SIZE ) ==
out)
                ; /* do nothing */
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
}
```

```
//consumer process
while (true) {
        while (in == out)
                ; /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) %
BUFFER_SIZE;
}
```

Producer consumer problem using shared memory

```
//shared data
Int
n=4,item,in,out;
in=0;out=0;
Int
BUFFER_SIZE=4
```

in=3 producer waits

0	**a**
1	b
2	c
3	

Now consumer runs

```
//producer process
while (true) {
      /* produce an item in next produced
*/
      while (((in + 1) % BUFFER_SIZE ) ==
out)
            ; /* do nothing */
      buffer[in] = next_produced;
      in = (in + 1) % BUFFER_SIZE;
}
```

```
//consumer process
while (true) {
      while (in == out)
            ; /* do nothing */
      next_consumed = buffer[out];
      out = (out + 1) %
BUFFER_SIZE;
}
```

The solution allowed at most BUFFER_SIZE - 1 items in the buffer at the same time.

we can modify the algorithm to remedy this deficiency.

One possibility is to add an integer variable counter, initialized to 0.

counter is incremented every time we add a new item to the buffer and is decremented

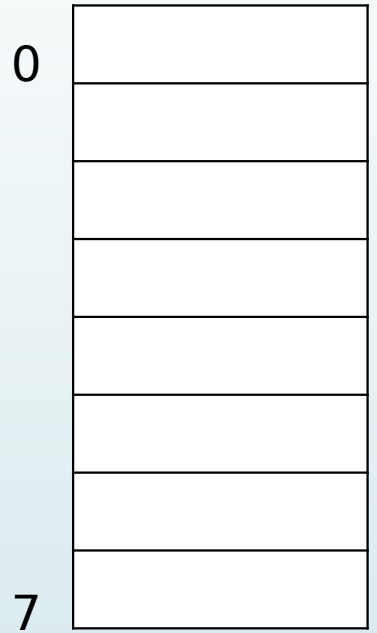
every time we remove one item from the buffer.

Producer consumer problem using shared memory

```
//shared data
int count=0,
n=8;
```

```
//producer process
while (true) {

    while (counter == BUFFER_SIZE)
            ; /* do nothing */

    buffer[in] = next_produced;

    in = (in + 1) % BUFFER_SIZE;
    counter++;
}
```

```
//consumer process
while (true) {
        while (counter == 0)
                ; /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;

    counter--;
}
```



Producer consumer problem using shared memory

```
//shared data
int count=0;
```

Current status
count=5

0	**x1**
1	X2
2	X3
3	X4
4	x5
5	
6	
7	

```
//producer process
while (true) {

    while (counter == BUFFER_SIZE)
        ; /* do nothing */

    buffer[in] = next_produced;

    in = (in + 1) % BUFFER_SIZE;
    counter++;
}
```

```
register1 = counter
register1 = register1 + 1
counter = register1
```

```
//consumer process
while (true) {
    while (counter == 0)
        ; /* do nothing */

    next_consumed = buffer[out];

    out = (out + 1) % BUFFER_SIZE;

    counter--;
}
```

```
register2 = counter
register2 = register2 - 1
counter = register2
```

Producer consumer problem using shared memory

```
//shared data
int count=0;
```

Producer runs: S0: producer execute
register1 = counter {register1 = 5}
S1: producer execute
register1 = register1 + 1 {register1 = 6}
Process switch occurs

```
//producer process
while (true) {

    while (counter == BUFFER_SIZE)
            ; /* do nothing */

    buffer[in] = next_produced;

    in = (in + 1) % BUFFER_SIZE;
    counter++;
}
```

```
//consumer process
while (true) {
        while (counter == 0)
                ; /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;

    counter--;
}
```

0	**x1**
1	X2
2	X3
3	X4
4	X5
5	x6
6	
7	

Producer consumer problem using shared memory

```
//shared data
int count=0;
```

S2: consumer execute
register2 = counter {register2 = 5}
S3: consumer execute
register2 = register2 – 1 {register2 = 4}
Process switch occurs

```
//producer process
while (true) {

        while (counter == BUFFER_SIZE)
                ; /* do nothing */

        buffer[in] = next_produced;

        in = (in + 1) % BUFFER_SIZE;
        counter++;
}
```

```
//consumer process
while (true) {
        while (counter == 0)
                ; /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;

        counter--;
}
```

0	
1	X2
2	X3
3	X4
4	X5
5	x6
6	
7	

Producer consumer problem using shared memory

```
//shared data
int count=0;
```

S4: producer execute
counter = register1 {counter = 6 }
Process switch occurs

0	
1	X2
2	X3
3	X4
4	X5
5	x6
6	
7	

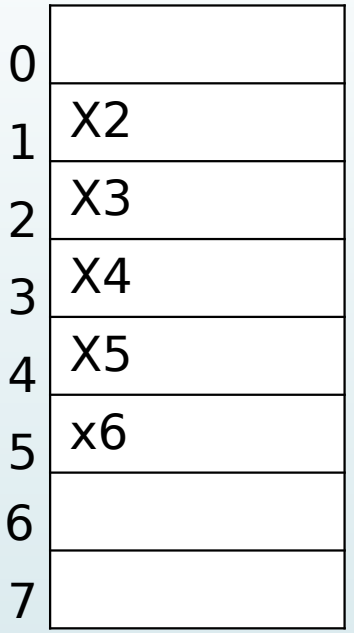
```
//producer process
while (true) {

        while (counter == BUFFER_SIZE)
                ; /* do nothing */

        buffer[in] = next_produced;

        in = (in + 1) % BUFFER_SIZE;
        counter++;
}
```

```
//consumer process
while (true) {
        while (counter == 0)
                ; /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;

        counter--;
}
```

Producer consumer problem using shared memory

```
//shared data
int count=0;
```

S5: consumer execute **`counter = register2`**
{counter = 4}




```
//producer process
while (true) {

        while (counter == BUFFER_SIZE)
                ; /* do nothing */

        buffer[in] = next_produced;

        in = (in + 1) % BUFFER_SIZE;
        counter++;
}
```

```
//consumer process
while (true) {
        while (counter == 0)
                ; /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;

      counter--;
}
```

- **counter++** could be implemented as

```
register1 = counter
register1 = register1 + 1
counter = register1
```

- **counter--** could be implemented as

```
register2 = counter
register2 = register2 - 1
counter = register2
```

- Consider this execution interleaving with “count = 5” initially:

S0: producer execute **`register1 = counter`** {register1 = 5}
S1: producer execute **`register1 = register1 + 1`** {register1 = 6}
S2: consumer execute **`register2 = counter`** {register2 = 5}
S3: consumer execute **`register2 = register2 – 1`** {register2 = 4}
S4: producer execute **`counter = register1`** {counter = 6 }
S5: consumer execute **`counter = register2`** {counter = 4}

- Notice that we have arrived at the incorrect state "counter == 4", indicating that four buffers are full, when, in fact, five buffers are full.
- We would arrive at this incorrect state because we allowed both processes to manipulate the variable counter concurrently.
- A situation like this, where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a race condition.

To guard against the race condition above, we need to ensure that only one process at a time can be manipulating the variable counter. To make such a guarantee, we require that the processes be synchronized in some way.