Amirkabir University of Technology
(Tehran Polytechnic)

Operating Systems

Processes-Part4

Seyyed Ahmad Javadi

sajavadi@aut.ac.ir

Spring 2023

Inter-Process Communication

- Processes within a system may be ***independent*** or ***cooperating***
- Cooperating process can affect or be affected by other processes, including ***sharing data***.
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience

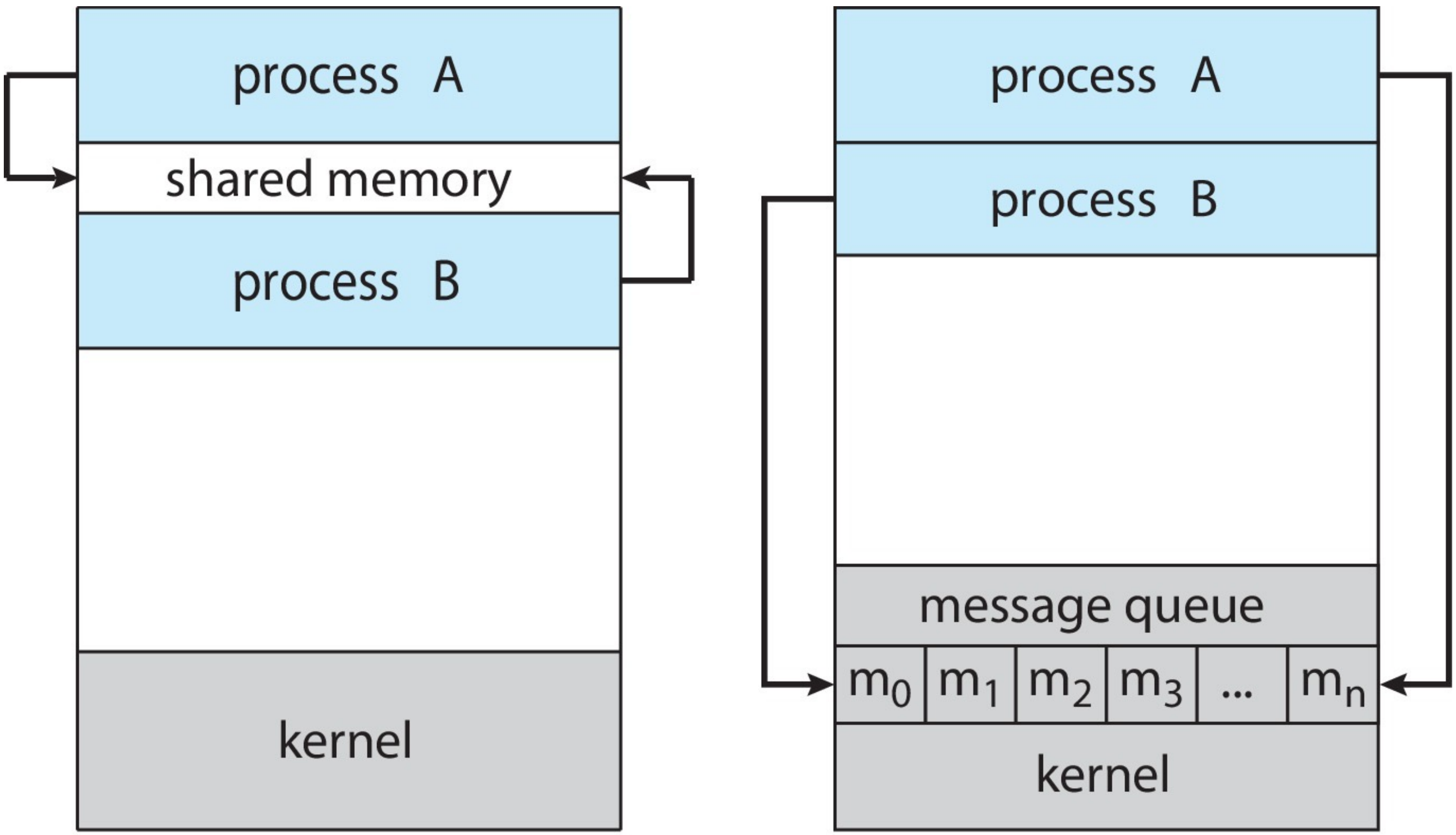
Inter-Process Communication (Cont.)

- Cooperating processes need interprocess communication (IPC)

- Two models of IPC
  - **Shared memory**
  - **Message passing**
    - **We do not cover this.**

Communications Models

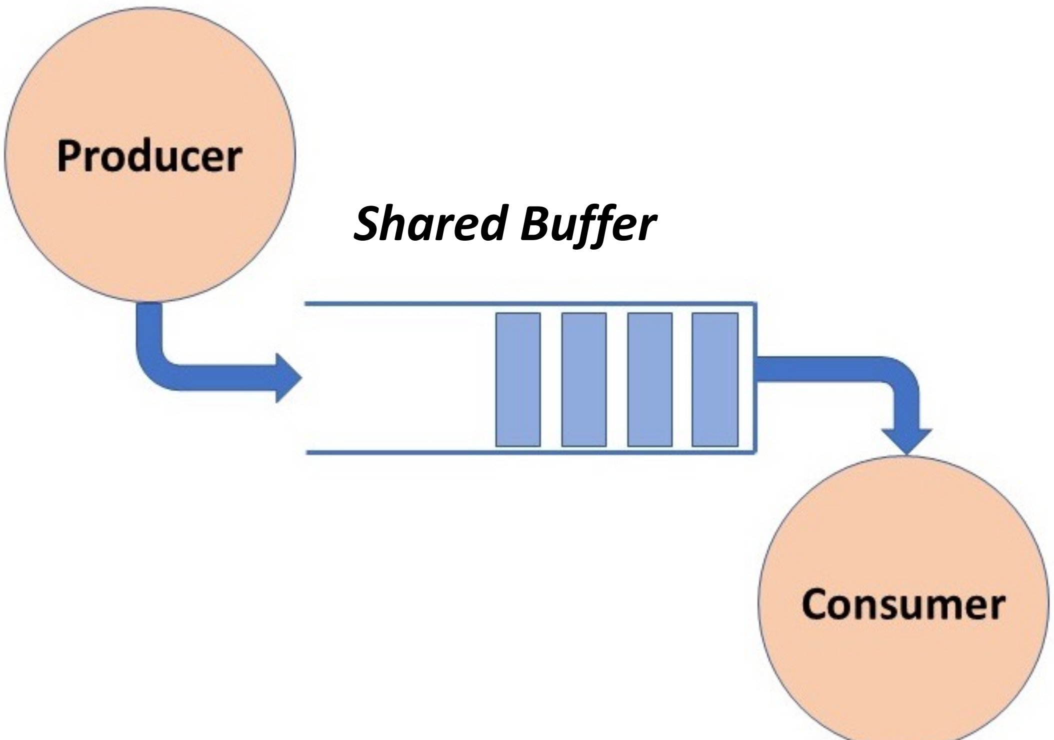
(a) Shared memory. (b) Message passing.



(a) (b)

Producer-Consumer Problem

- Paradigm for cooperating processes:
  - **Producer** process produces information that is consumed by a **consumer** process.



https://www.educative.io/edpresso/what-is-the-producer-consumer-problem

Producer-Consumer Problem-Variations

- **Unbounded-buffer** places no practical limit on the size of the buffer:
  - Producer never waits
  - Consumer waits if there is no buffer to consumer
- **Bounded-buffer** assumes that there is a fixed buffer size
  - Producer must wait if all buffers are full
  - Consumer waits if there is no buffer to consume

IPC – Shared Memory

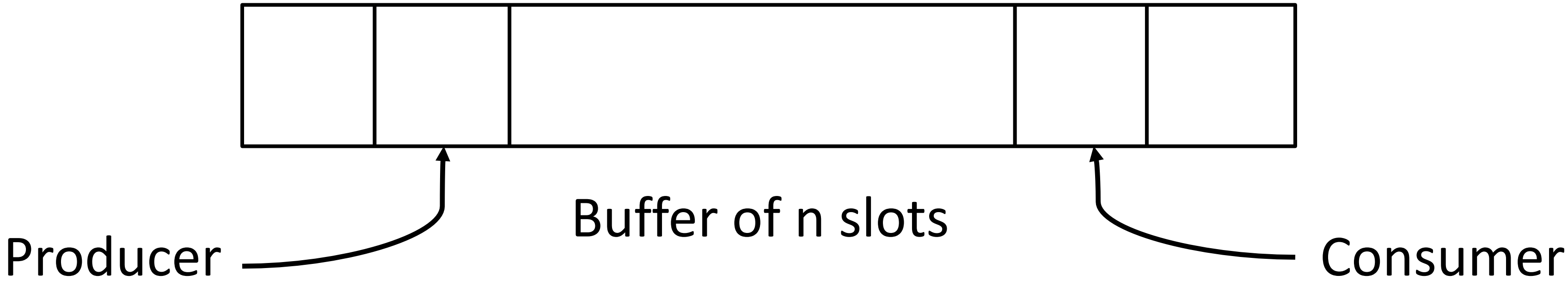
- An area of memory shared among the processes that wish to communicate.
- The communication is ***under the control of the users*** processes ***not the operating system***.
- Major issues is to provide mechanism that will allow the user processes ***to synchronize their actions*** when they access shared memory.
- Synchronization is discussed in great details in Chapters 6 & 7.

Bounded-Buffer – Shared-Memory Solution

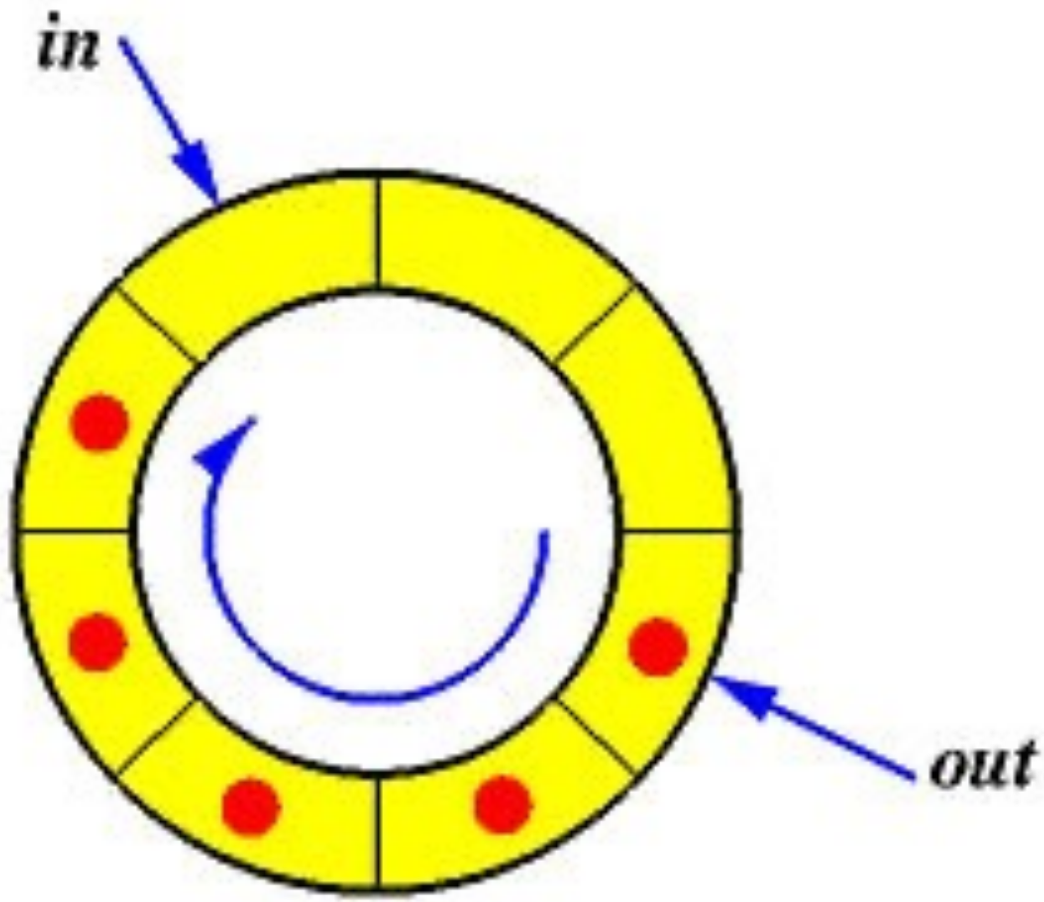
- Shared data

```
#define BUFFER_SIZE 10
typedef struct {
  . . .
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

- Solution is correct but can only use **BUFFER_SIZE - 1** elements.

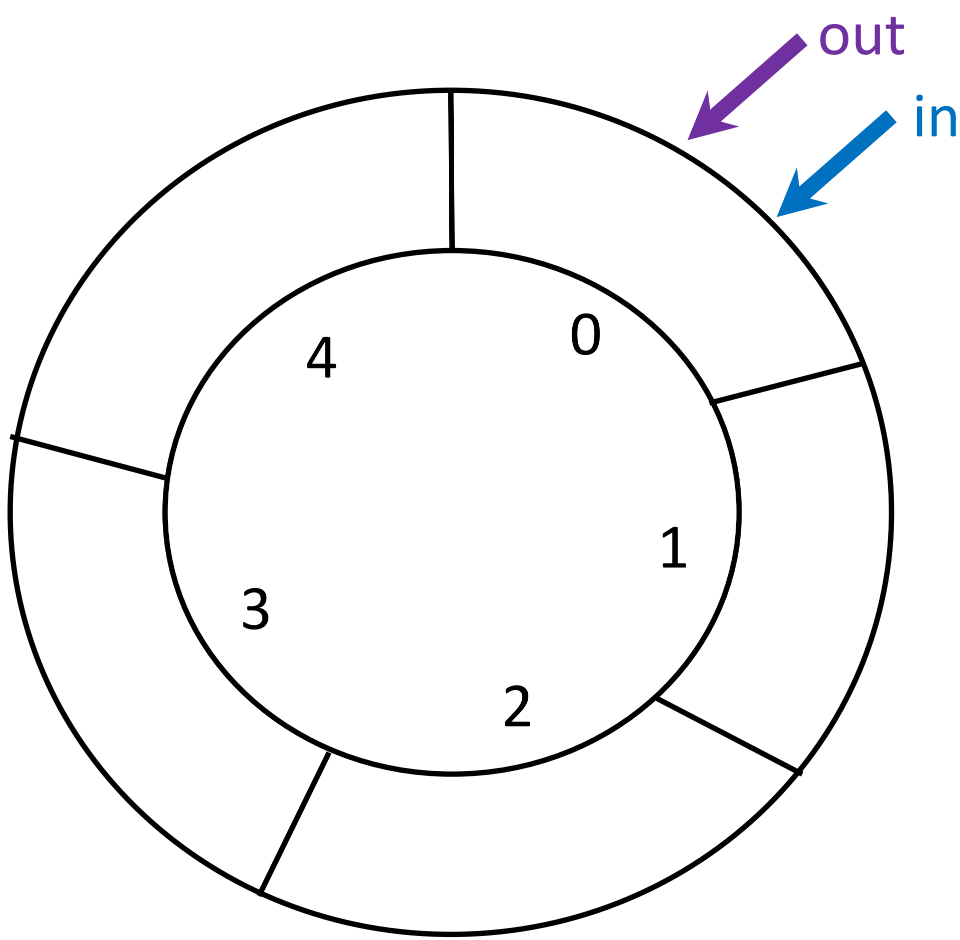


Circular Bounded-Buffer



Source: https://pages.mtu.edu/~shene/NSF-3/e-Book/SEMA/TM-example-buffer.html

Start Point

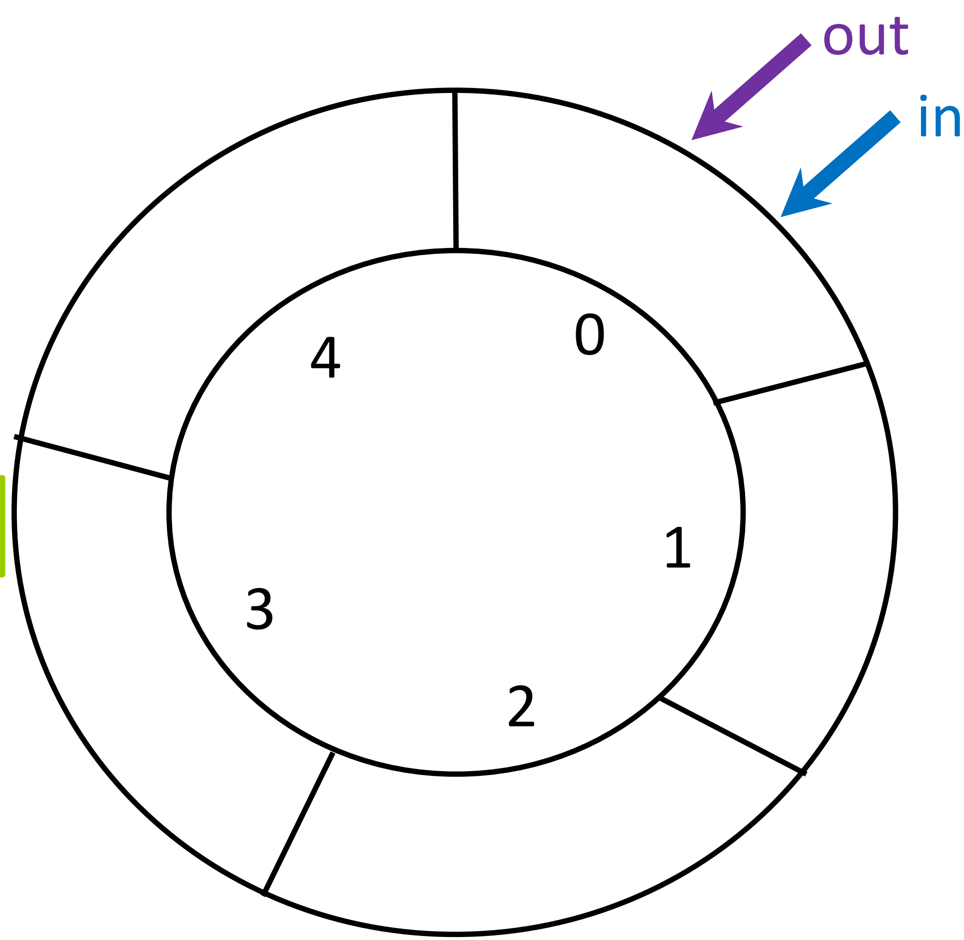


Producer

Produce an item

```
while (((in + 1) % BUFFER_SIZE) == out;
```

```
(0 + 1) % 5 = 1 != 0
```



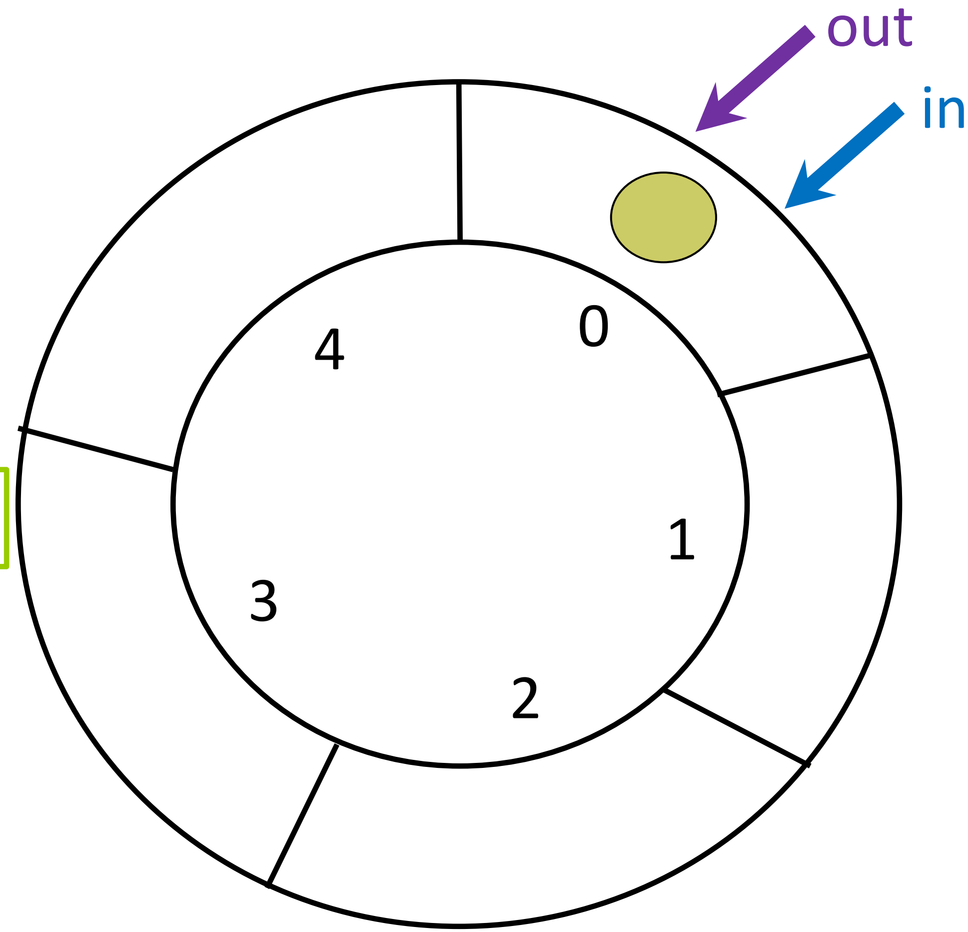
Producer

Produce an item

```
while (((in + 1) %
BUFFER_SIZE) == out;
```

```
(0 + 1) % 5 = 1 != 0
```

```
buffer[0] = item;
```



Producer

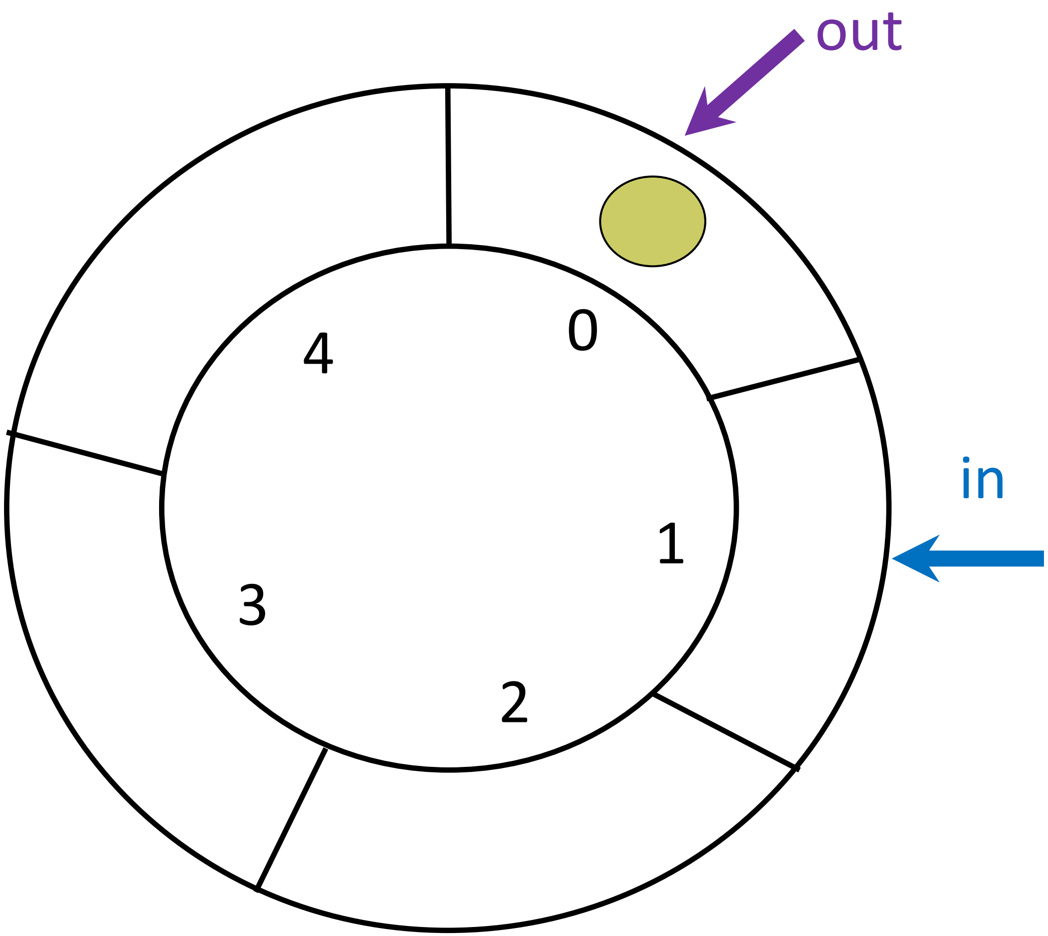
Produce an item

```
while (((in + 1) % BUFFER_SIZE) == out;
```

```
(0 + 1) % 5 = 1 != 0
```

```
buffer[0] = item;
```

```
in = (0 + 1) % 5;
```

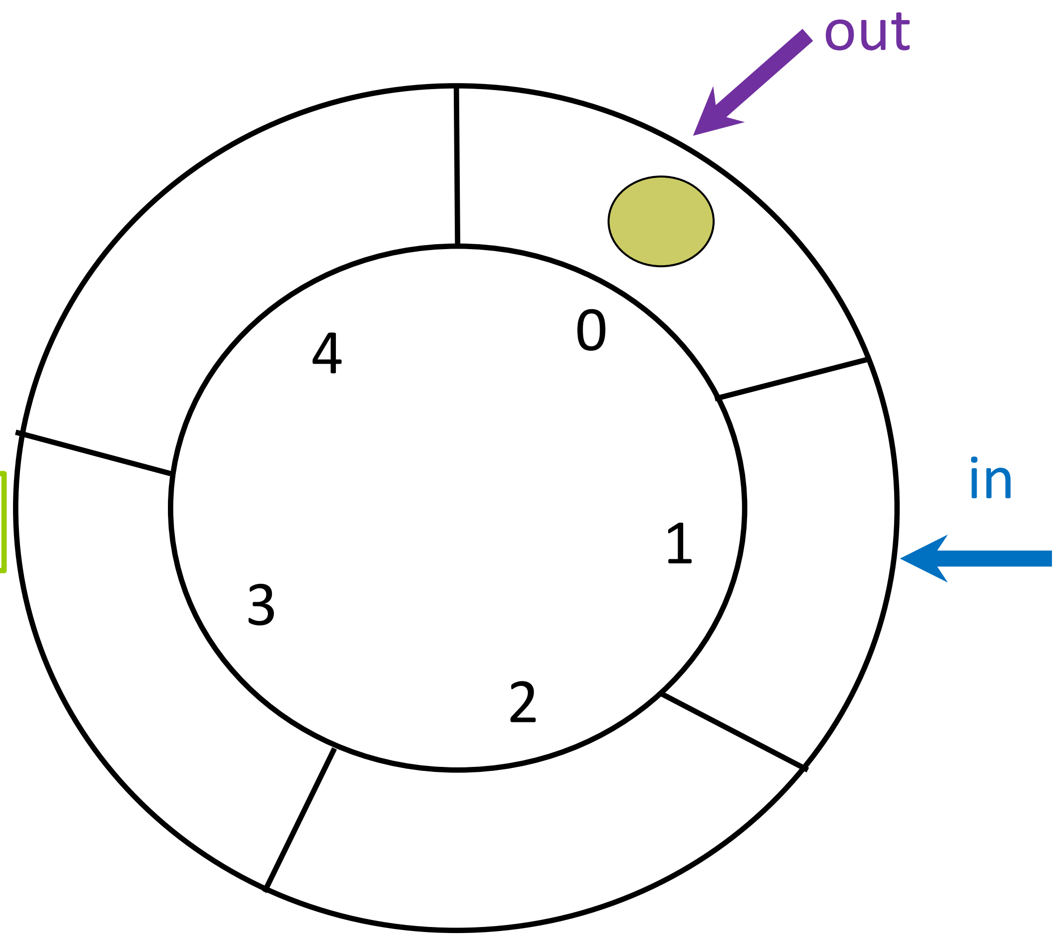


Producer

Produce an item

```
while (((in + 1) %
BUFFER_SIZE) == out;
```

```
(1 + 1) % 5 = 2 != 0
```



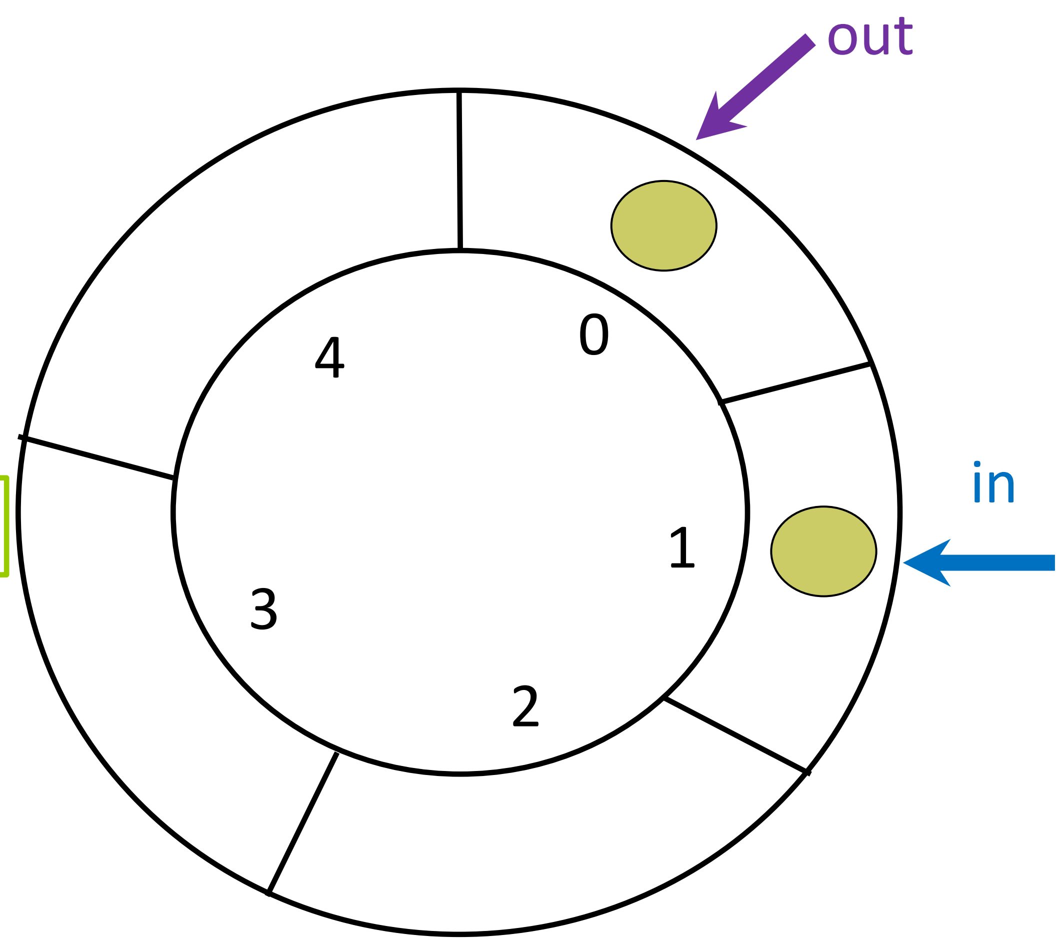
Producer

Produce an item

```
while (((in + 1) % BUFFER_SIZE) == out;
```

```
(1 + 1) % 5 = 2 != 0
```

```
buffer[1] = item;
```



Producer

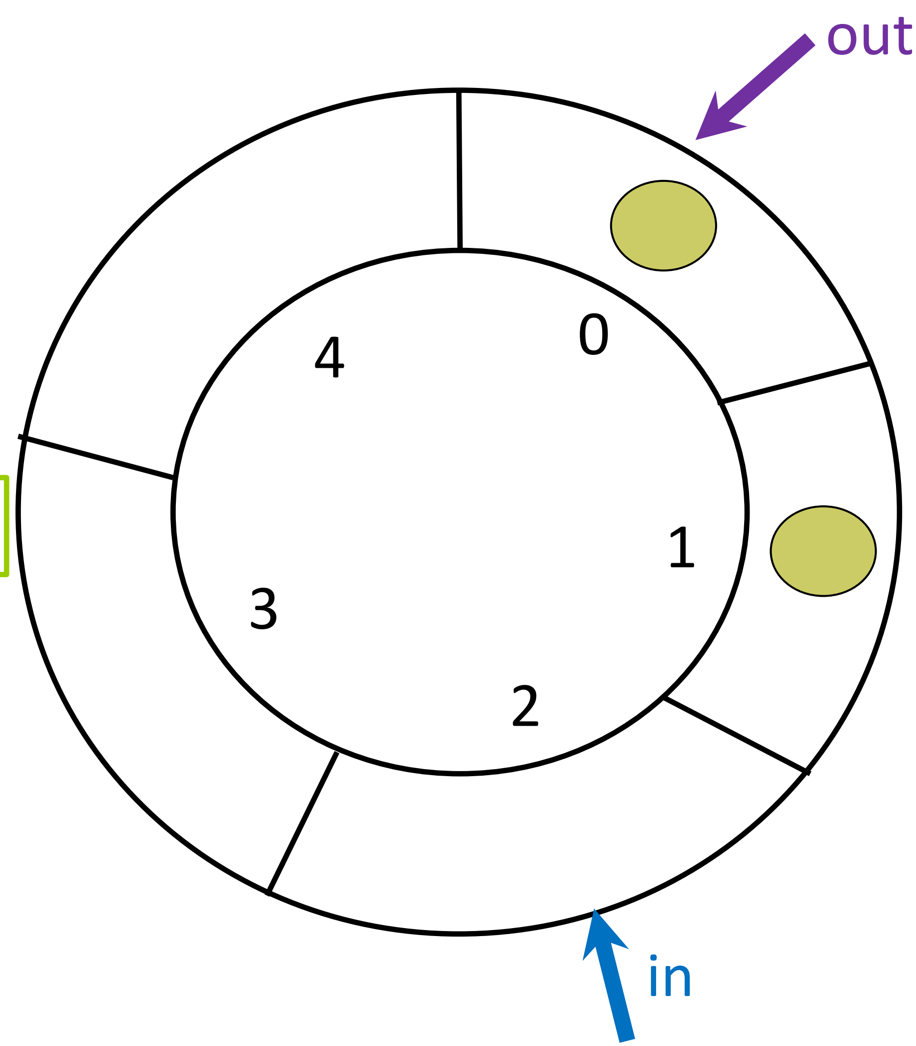
Produce an item

```
while (((in + 1) %
BUFFER_SIZE) == out;
```

```
(1 + 1) % 5 = 2 != 0
```

```
buffer[1] = item;
```

```
in = (1 + 1) % 5;
```



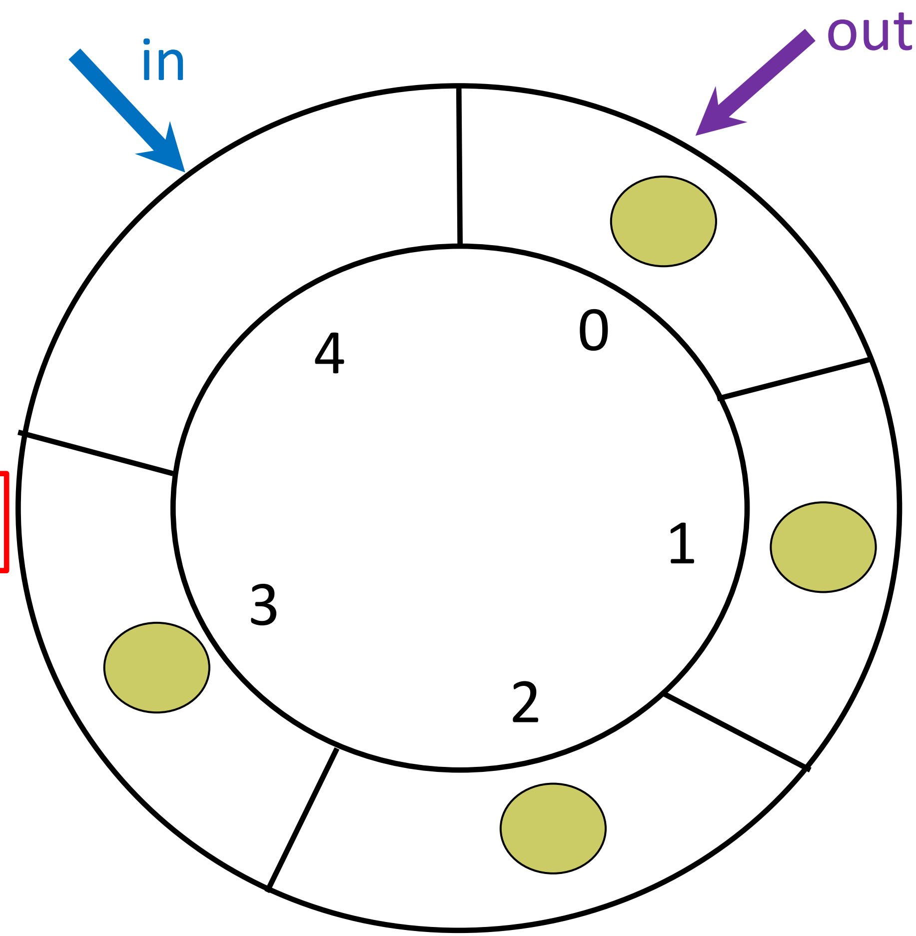
Producer

Produce an item

```
while (((in + 1) %
BUFFER_SIZE) == out;
```

```
(4 + 1) % 5 = 0 == 0
```

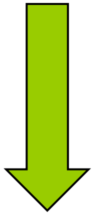
No more space



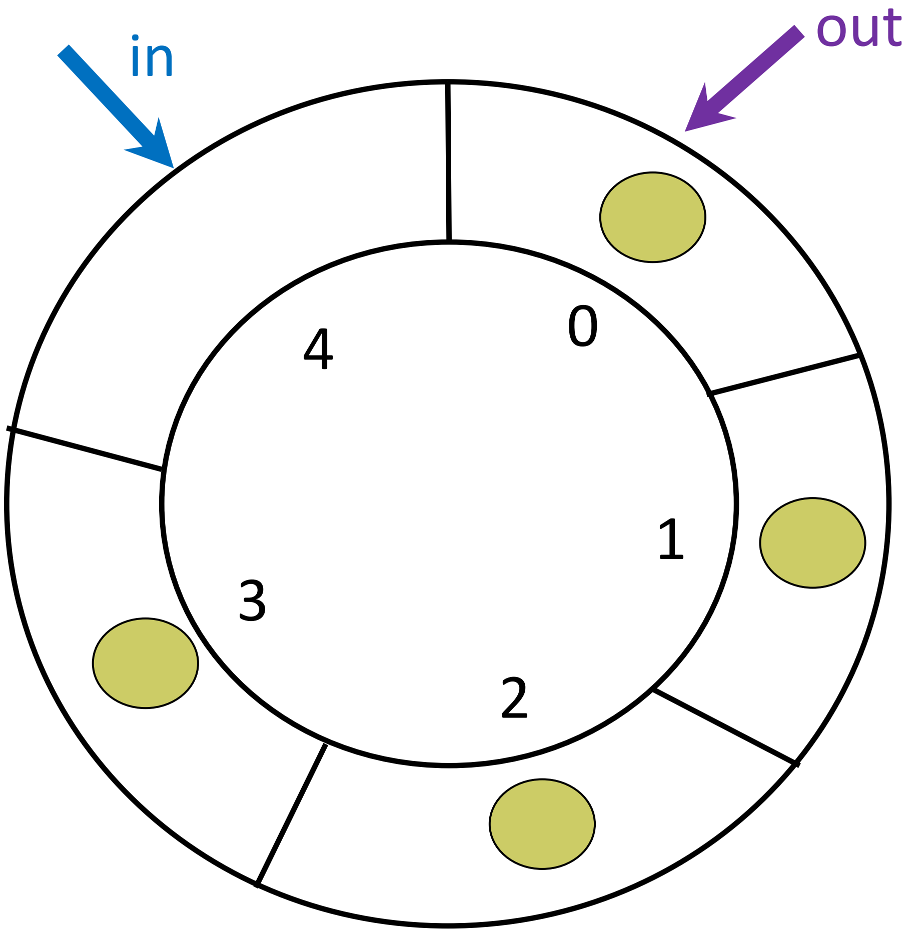
Consumer

Consume an item

```
while (in == out);
```



4	!=	0



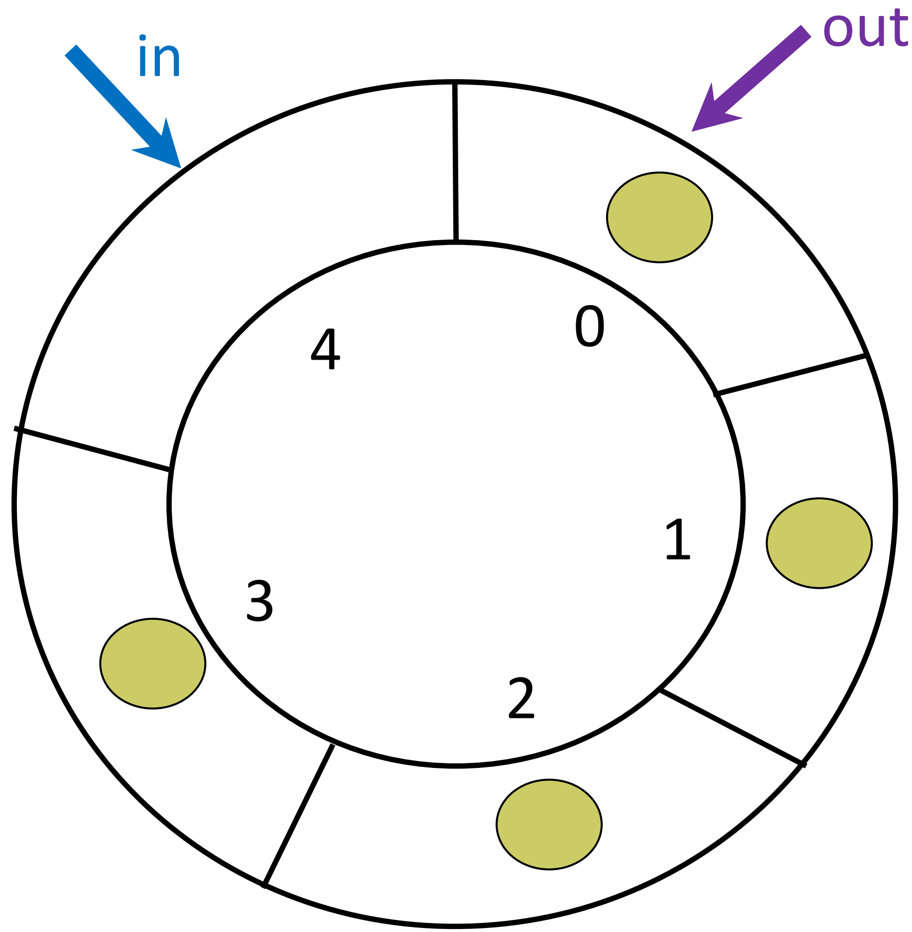
Consumer

Consume an item

```
while (in == out);
```

```
4 != 0
```

```
item = buffer[0];
```



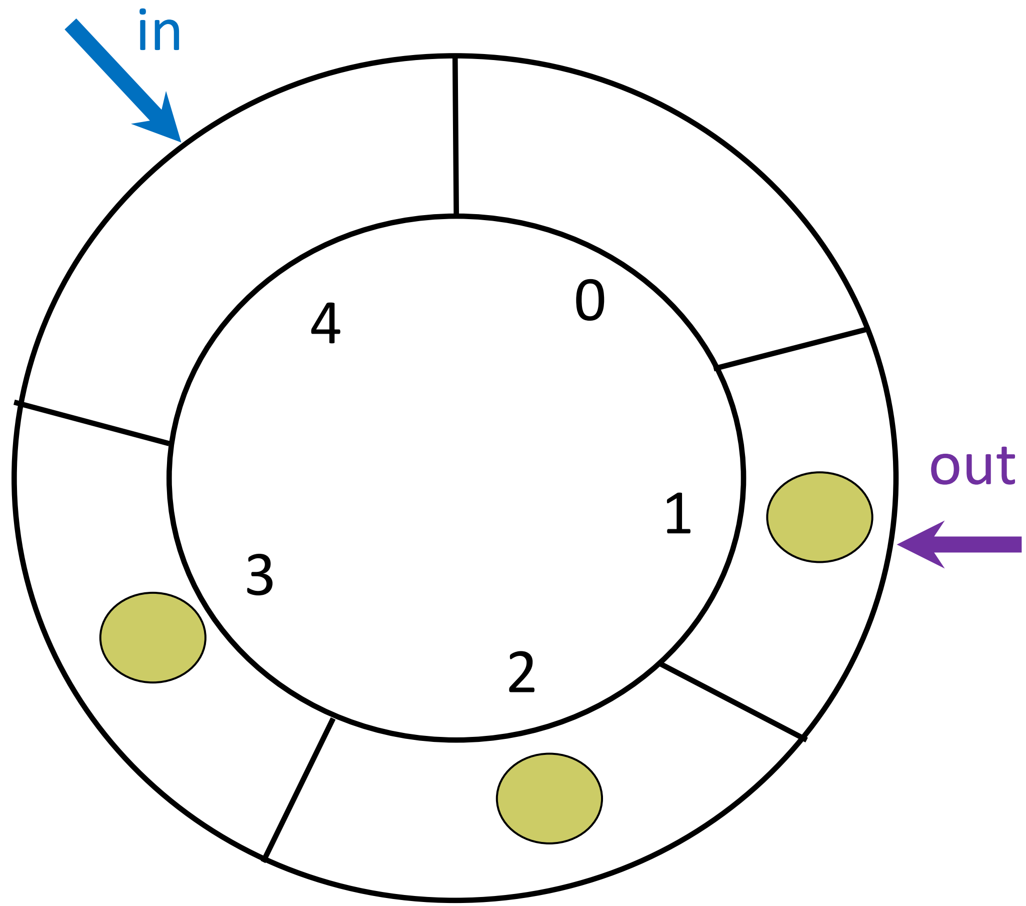
Consumer

Consume an item

```
while (in == out);
```

$\boxed{4 \; != \; 0}$

```
item = buffer[0];

out = (0 + 1) % 5;
```



Producer

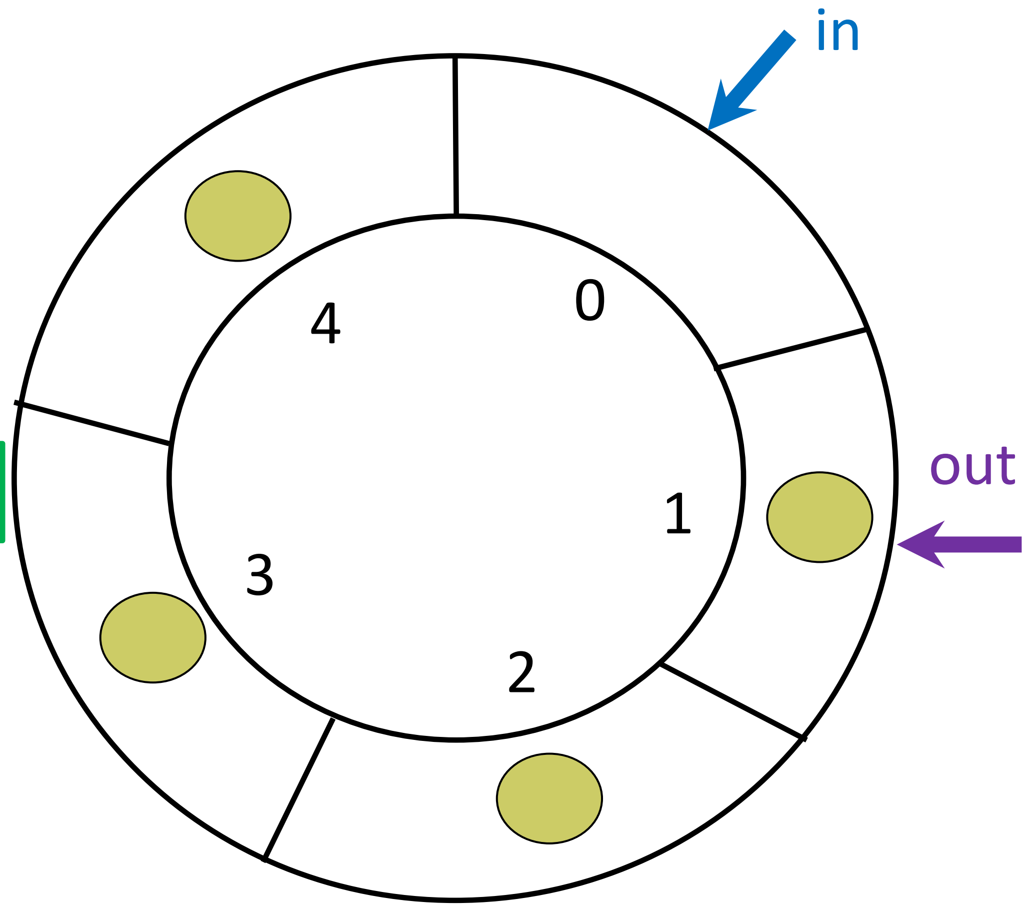
Produce an item

```
while (((in + 1) %
BUFFER_SIZE) == out;
```

```
(4 + 1) % 5 = 0 != 1
```

```
buffer[4] = item;
```

```
in = (4 + 1) % 5;
```



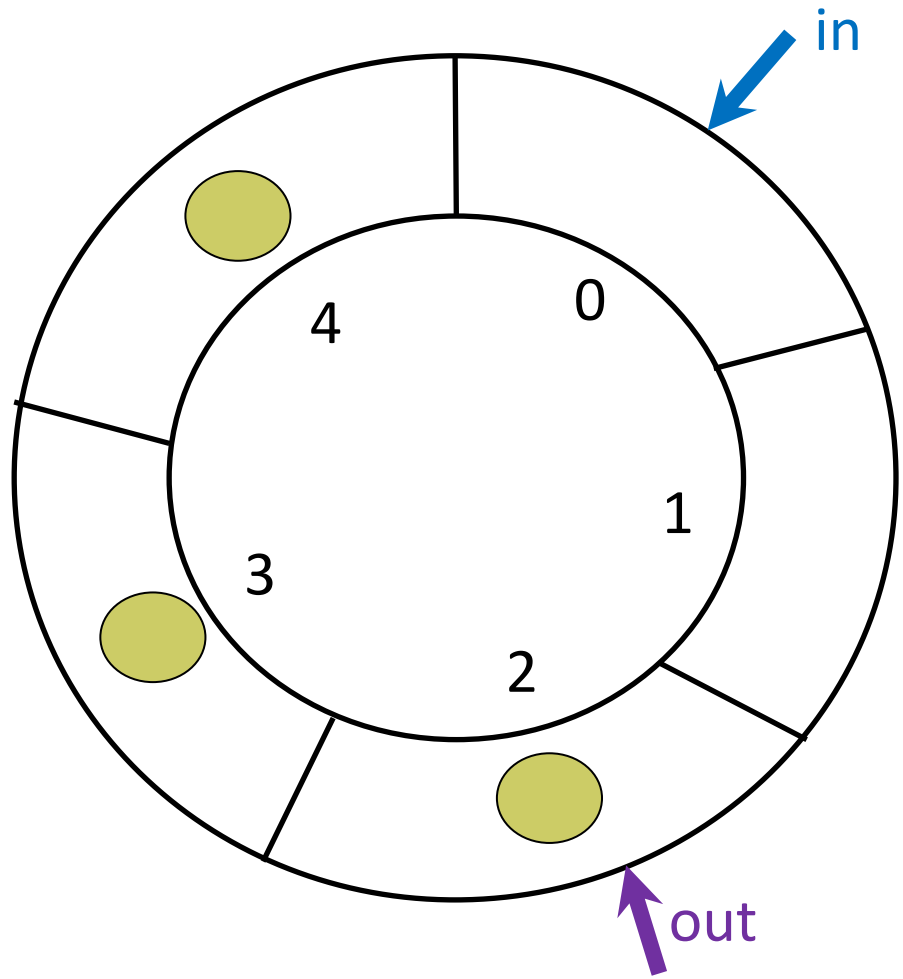
Consumer

Consume an item

```
while (in == out);
```

$\boxed{1 \; != \; 0}$

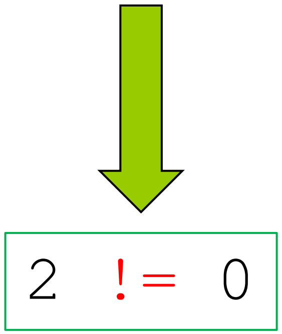
```
item = buffer[1];

out = (1 + 1) % 5;
```



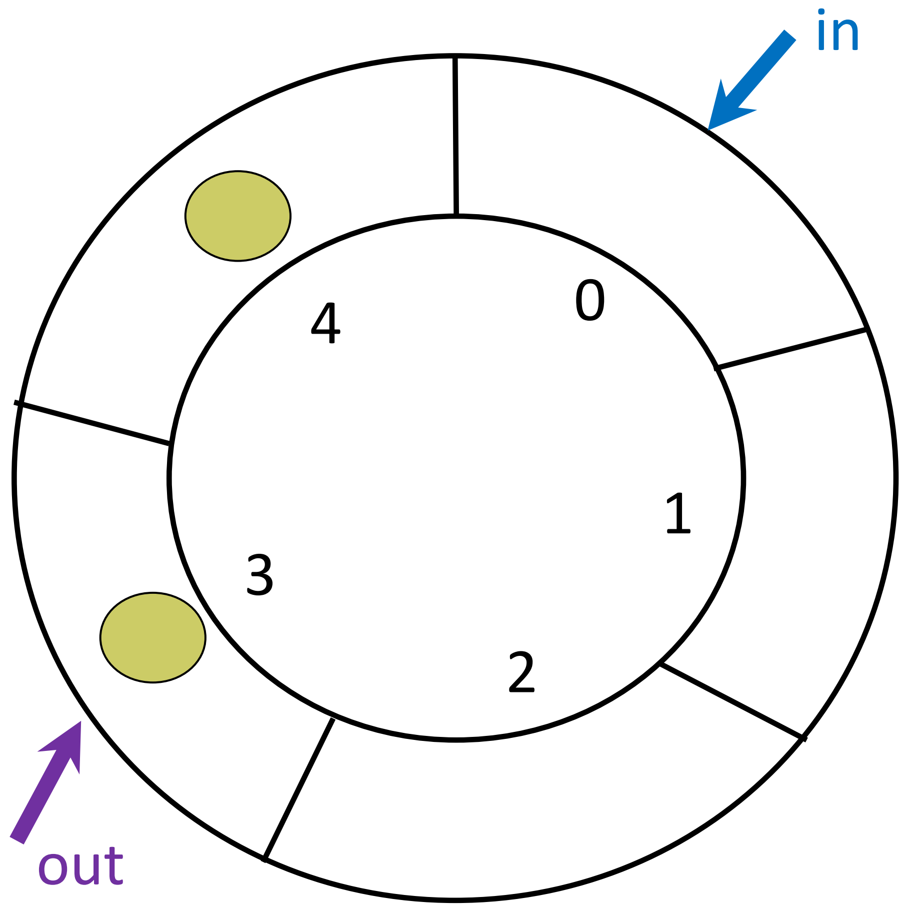
Consumer

Consume an item

```
while (in == out);
```




```
item = buffer[2];

out = (2 + 1) % 5;
```



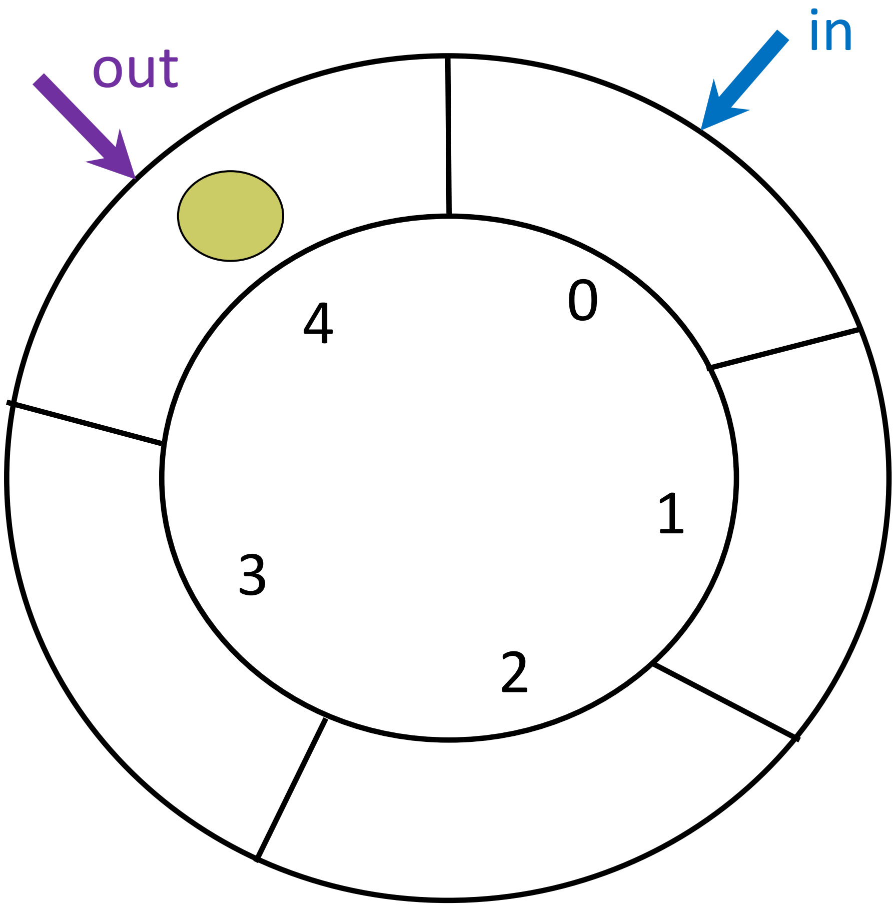
Consumer

Consume an item

```
while (in == out);
```

$\boxed{3\ != \ 0}$

```
item = buffer[3];

out = (3 + 1) % 5;
```



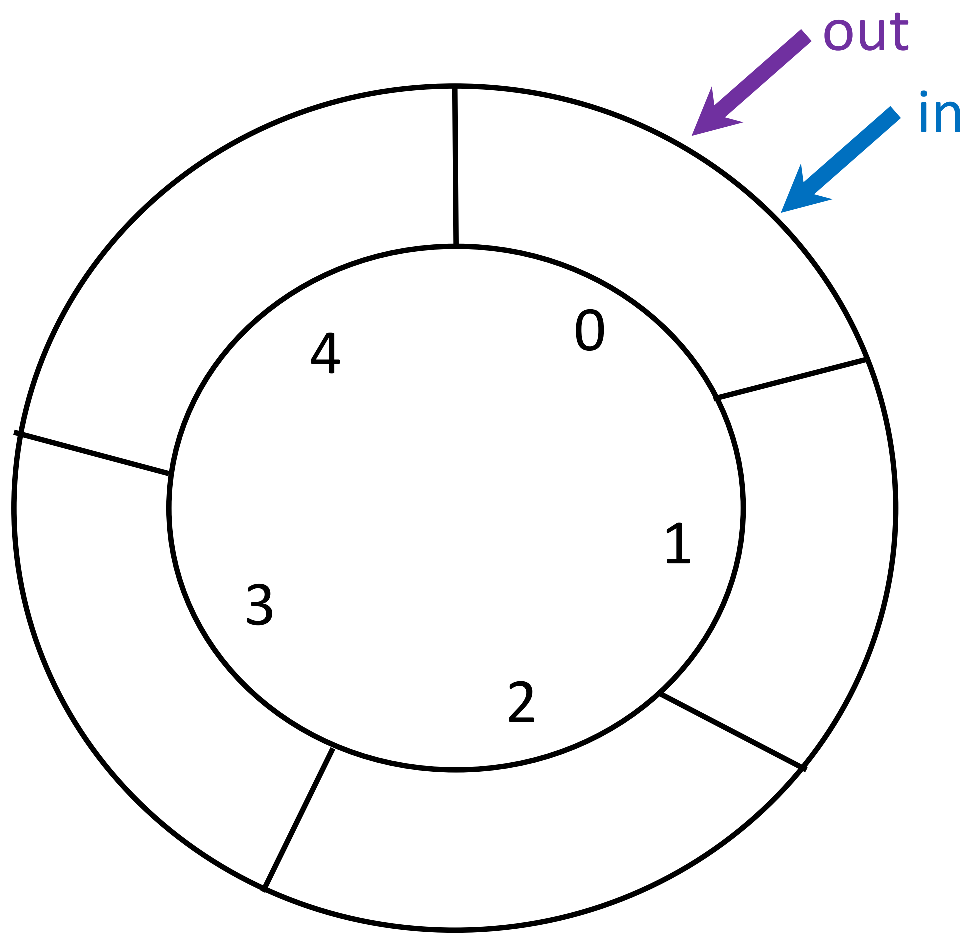
Consumer

Consume an item

```
while (in == out);
```

$\boxed{3\ \texttt{!=}\ 0}$

```
item = buffer[4];

out = (4 + 1) % 5;
```



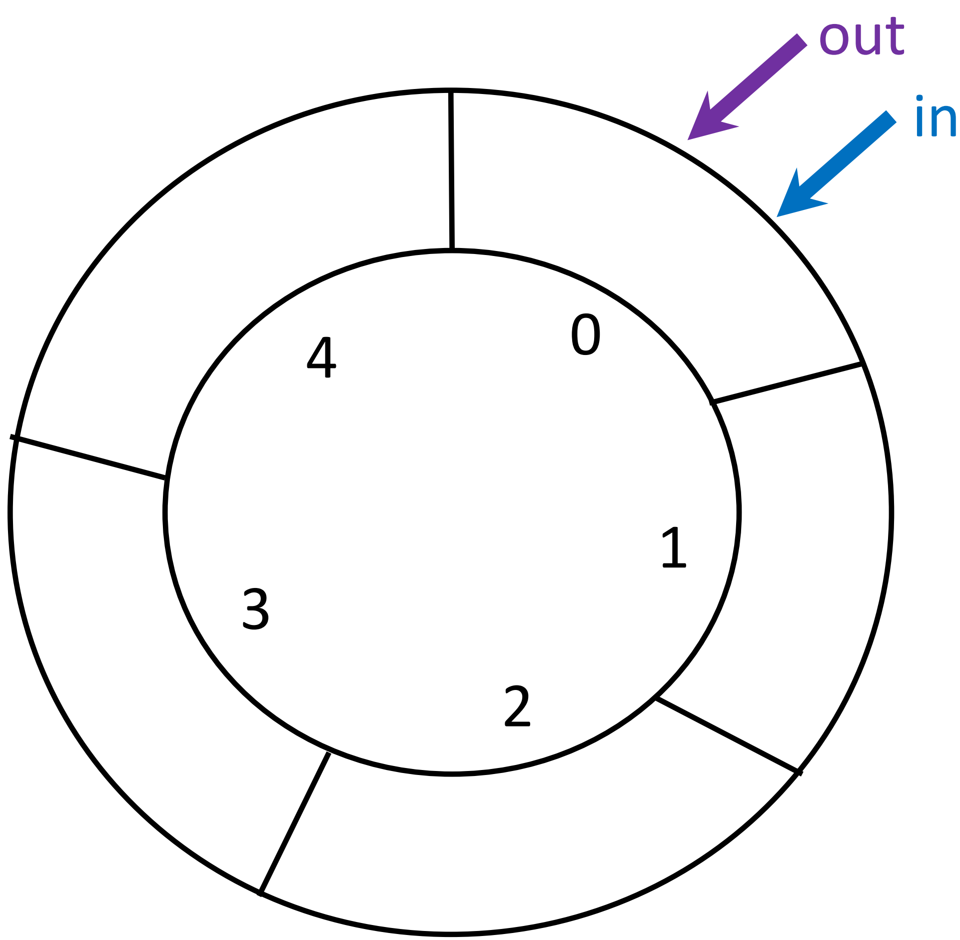
Consumer

Consume an item

```
while (in == out);
```

$\boxed{0 == 0}$

Nothing to consume.



Producer Process – Shared Memory

```
item next_produced;

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 – Shared Memory

```
item next_consumed;

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

      /* consume the item in next consumed */
}
```

What about Filling all the Buffers?

- Suppose that we wanted to provide a solution to the consumer-producer problem that **fills all the buffers.**

- How can we do it?

What about Filling all the Buffers? (ont.)

- We can do so by having ***an integer counter*** that keeps track of the number of full buffers.
- Initially, counter is set to 0.
- The integer counter is incremented by the producer after it produces a new buffer.
- The integer counter is decremented by the consumer after it consumes a buffer.

Producer

```
while (true) {
      /* produce an item in next produced */

      while (counter == BUFFER_SIZE)
            ; /* do nothing */
      buffer[in] = next_produced;
      in = (in + 1) % BUFFER_SIZE;
      counter++;
}
```

Consumer

```
while (true) {
      while (counter == 0)
            ; /* do nothing */
      next_consumed = buffer[out];
      out = (out + 1) % BUFFER_SIZE;
          counter--;
      /* consume the item in next
consumed */
}
```

Race Condition

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

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

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

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

Race Condition (cont.)

- 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}
```

Race Condition (cont.)

Question – why was there no race condition in the first solution (where at most N – 1) buffers can be filled?

More in Chapter 6.