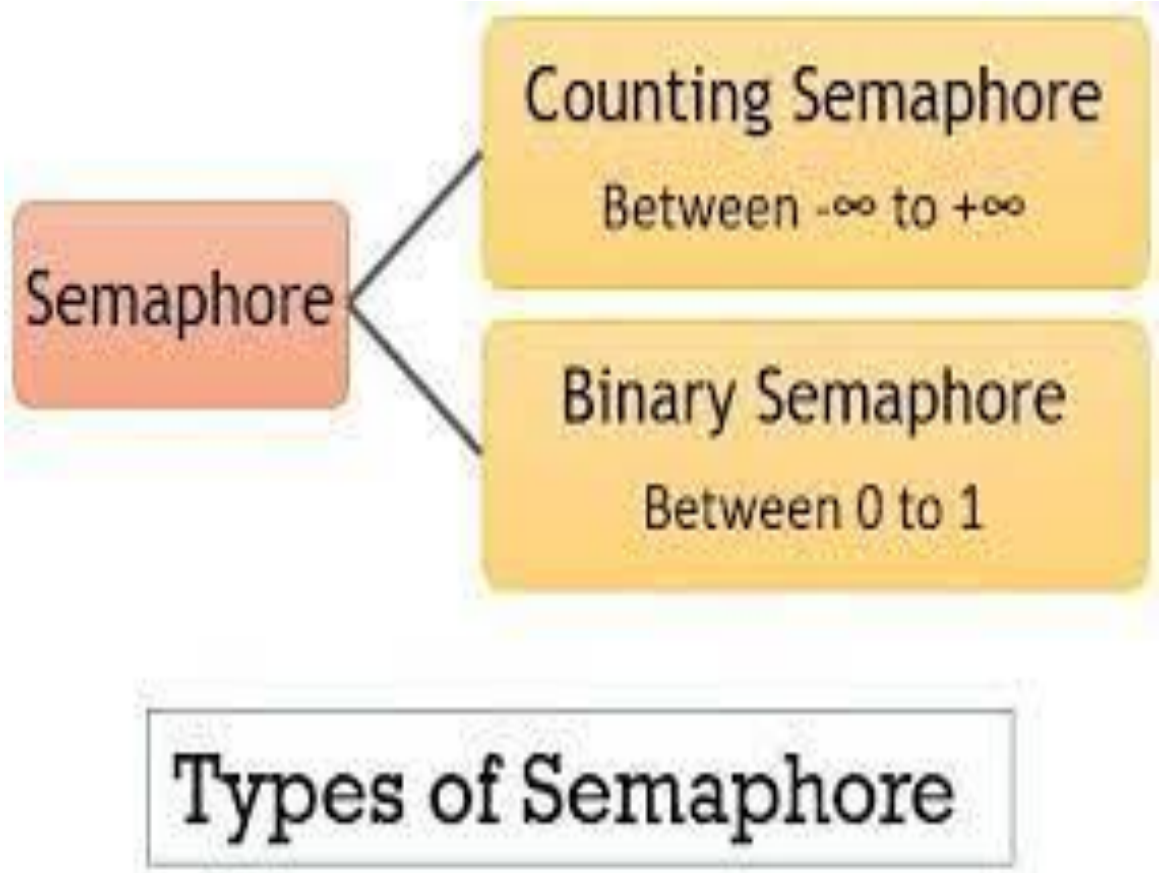
OPERATING SYSTEM: CSET209

OUTLINE

- **Semaphores**
- **Classical synchronization problem**
  - **Producer consumer problem**

SEMAPHORES

- A semaphore ***S*** – integer variable which is used by various processes in a mutual exclusive manner to achieve synchronization.
- The improper usage of semaphore will also give the improper results.



SEMAPHORES

- Can only be accessed via two indivisible (atomic) operations
  - **wait()** and **signal()**
    - Originally called **P()** and **V()**.
- Definition of the **wait() operation**

```
wait(S) {
    while (S <= 0)
      ; // busy wait
    S--;
}
```

- Definition of the **signal() operation**

```
signal(S) {
    S++;
}
```

Wait

The wait operation decrements the value of its argument S, if it is positive. If S is negative or zero, then no operation is performed.

Signal

The signal operation increments the value of its argument S.

SEMAPHORES

Semaphore Implementation

Mutual exclusion implementation with semaphores

Shared data:

```
semaphore mutex; //initially mutex = 1
```

Process Pi:

```
do {
    wait(mutex);
        critical section
    signal(mutex);
        remainder section
}while(1);
```

COUNTING SEMAPHORES

```
Struct Semaphore
{
int value;                  /* limit to enter at critical section */
QueueType  queue;   /* suspended list */
}
Wait (Semaphore S)    /* function for enter critical section */
{
  1. S.value = S.value – 1;
  2. if (S.value<0)
          { Add process in S.queue   /* insert process to suspended list */
            sleep( ) ;
          }
}

Signal (semaphore S)        /* function to leave critical section */
{      1.   S.value =S.value + 1;
       2. if (S.value  ≤ 0)
            { Select process from S.queue   /* select a process from suspended list */
                wakeup( );
            }
}
```

BINARY SEMAPHORES

```
B_Wait ( binary_semaphore   S)
{  if (S.value ==1)
   {
S.value=0;
   }
   Else
   {
            /* Place the process in
S.queuelist*/
Sleep ( );     /* Block the process*/
}
}
```

```
B_Signal ( binary_semaphoreS)
{  if (S.queuelist is empty())
   {
S.value=1;
   }
   Else
      {   /* Select the process p from
S.queuelist*/
wake up( );    /* Place the process P in
ready list*/
   }
}
```

Note: Wait and Signal is the same as the Down and Up operation.

Counting Semaphore vs Binary Semaphore

Counting	Binary
1. No mutual exclusion.	1. Mutual exclusion
2. Any integer value	2. Value is 0 and 1
3. More than one slot	3. Only one slot
4. Provide set of Processes	4. Have mutual exclusion mechanism.

PRACTICE PROBLEMS BASED ON COUNTING SEMAPHORES IN OS-

A counting semaphore S is initialized to 6. How many successful down operations can be performed?

Solution-

Six

5, 4, 3, 2, 1, 0

PRACTICE PROBLEMS BASED ON COUNTING SEMAPHORES IN OS-

A counting semaphore S is initialized to 10. Then, 6 P operations and 4 V operations are performed on S. What is the final value of S?

Solution-

We know-
•P operation also called as wait operation decrements the value of semaphore variable by 1.
•V operation also called as signal operation increments the value of semaphore variable by 1.

Thus,
Final value of semaphore variable S
= 10 – (6 x 1) + (4 x 1)
= 10 – 6 + 4
= 8

PRACTICE PROBLEMS BASED ON COUNTING SEMAPHORES IN OS-

A counting semaphore S is initialized to 7. Then, 20 P operations and 15 V operations are performed on S. What is the final value of S?

Solution-

We know-
•P operation also called as wait operation decrements the value of semaphore variable by 1.
•V operation also called as signal operation increments the value of semaphore variable by 1.

Thus,
Final value of semaphore variable S
= 7 – (20 x 1) + (15 x 1)
= 7 – 20 + 15
= 2

PRACTICE PROBLEMS BASED ON COUNTING SEMAPHORES IN OS-

A counting semaphore S is initialized to 10. Then, 12 P operations and 'X' V operations are performed on S. if the final value of S is 7, 'X' will be?

Solution-

We know-
•P operation also called as wait operation decrements the value of semaphore variable by 1.
•V operation also called as signal operation increments the value of semaphore variable by 1.

Thus,
$7 = 10 – 12 + X$
$7 = - 2 + X$
$X = 9$

PRACTICE PROBLEMS BASED ON BINARY SEMAPHORES IN OS-

Consider each process P_i { i = 1 to 9} execute code1 and process P_{10} execute code2. Each code shares and implements a binary semaphore S who's initialized to value 1.

Code1	Code2

while (true){
 	
wait(S);
//Critical section
signal(S);
}
	while (true){
	
	signal(S);
	//Critical section
	signal(S);
	}

What is the maximum number of processes that may be present in critical section at any point of time?

a. One
b. Two
c. Three
d. Ten

SOLUTION

Suppose We start with process P1, means P1 enters Critical section by performing a DOWN on semaphore mutex. At this point **MUTUX=0**, Now P10 executes and performs an UP on mutex and enters Critical section. At this point **MUTUX=1**, Now, Any one process out of **p2….p9** enters critical section by performing a DOWN on binary semaphore mutex.Finally, **mutux=0** and there is no way we could perform an UP on binary semaphore without allowing any process to leave Critical Section.

So, Maximum number of processes which are allowed to enter critical section =3 (two out of P1…P9 and P10).

WHAT IS THE OUTPUT??

Mutex a, b initially a = 1, b = 0;

P0	P0

While (True)
{
P(a); //down a
Print ("1");
V(b); // up b
}
	While (True)
	{
	P(b); //down p
	Print ("0");
	V(a); // up a
	}

PRACTICE PROBLEMS BASED ON BINARY SEMAPHORES IN OS-

Suppose we want to synchronize two concurrent processes P and Q using binary semaphores 'S' and 'T'. The code for the process P and Q is shown below:

Process P	Process Q

while(1)
{
W;
print("0");
print("0");
X;
}
	while(1)
	{
	Y;
	print("1");
	print("1");
	Z;
	}

Synchronization statements can be inserted only at points W, X, Y, and Z. Which of the following will always lead to output string "001100110011……."?

a. P(S) at W; V(S) at X; P(T) at Y; V(T) at Z where S=1, T=1 initially.

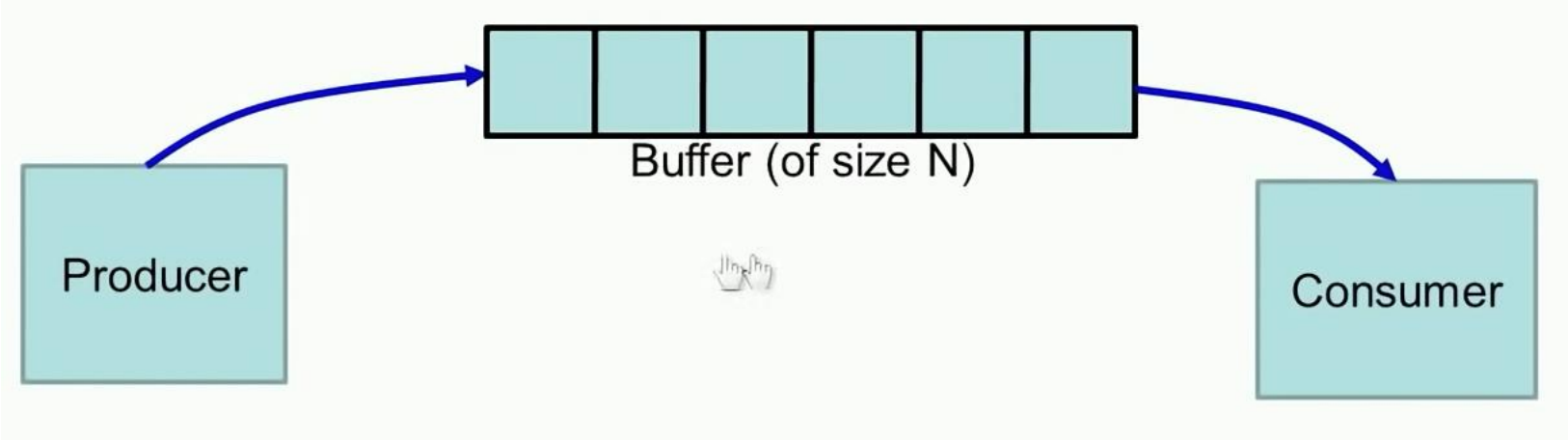
b. P(S) at W; V(T) at X; P(T) at Y; V(S) at Z where S=1, T=1 initially.

c. P(S) at W; V(S) at X; P(T) at Y; V(T) at Z where S=1, T=0 initially.

d. P(S) at W; V(T) at X; P(T) at Y; V(S) at Z where S=1, T=0 initially.

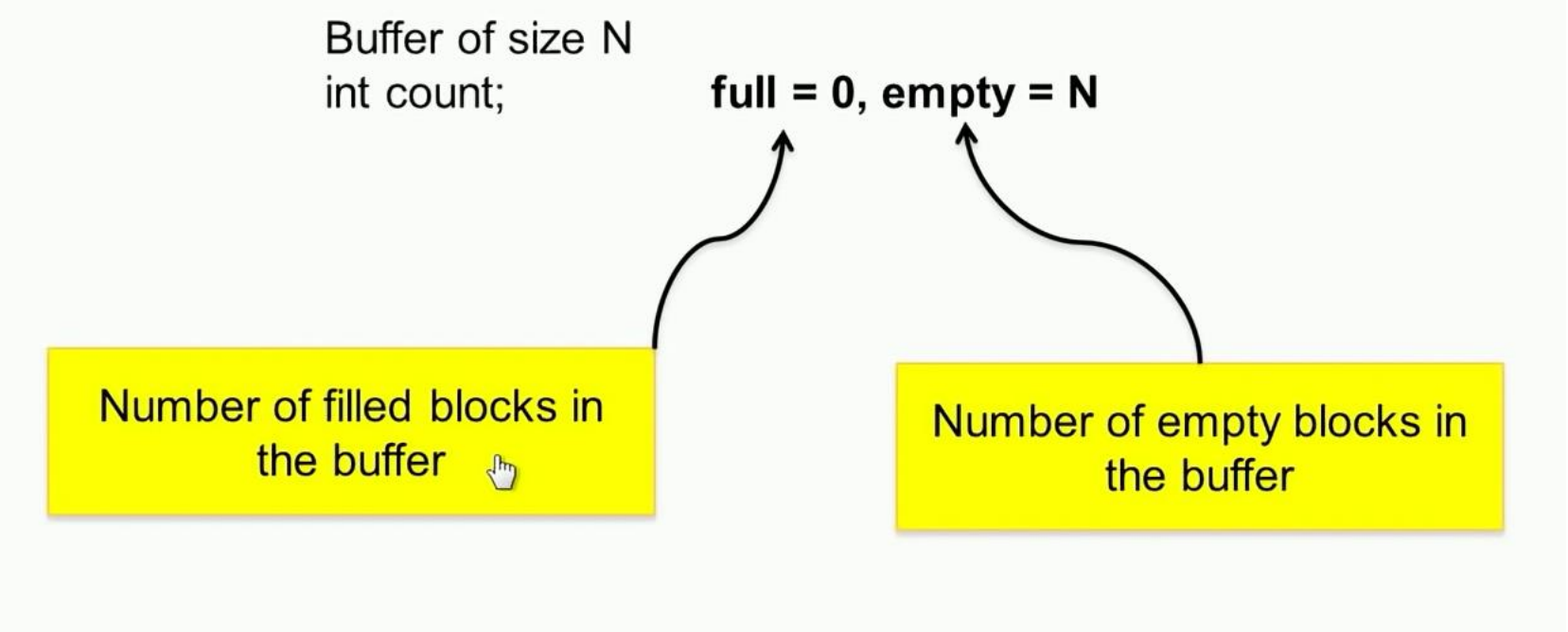
PRODUCER – CONSUMER PROBLEM

1. Producer produces and stores in buffer, Consumer consumes from buffer.
2. Required synchronization when:
    1. Producer produces, but buffer is full.
    2. Consumer consumes, but buffer is empty.
3. Also known as **bounded buffer problem**.



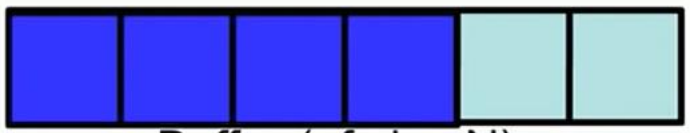
PRODUCER – CONSUMER WITH SEMAPHORES

1. Two Semaphores, **full** and **empty** are used.



PRODUCER – CONSUMER WITH SEMAPHORES

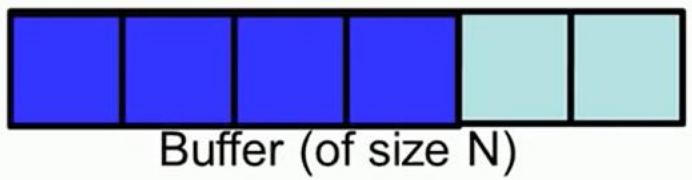
```
void producer(){
   while(TRUE){
      item = produce_item();
      down(empty);

      insert_item(item); // into buffer

      up(full);
   }
}
```



N=6
full = 4
empty = 2

PRODUCER – CONSUMER WITH SEMAPHORES

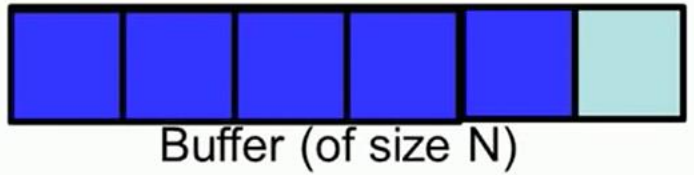
```
void producer(){
    while(TRUE){
        item = produce_item();
        down(empty);

        insert_item(item); // into buffer

        up(full);
    }
}
```



N=6
full = 4
empty = 1

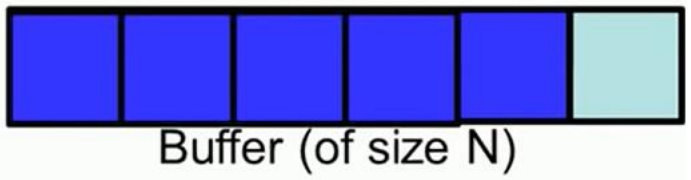
PRODUCER – CONSUMER WITH SEMAPHORES

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

        insert_item(item); // into buffer

        up(full);
    }
}
```



N=6
full = 5
empty = 1

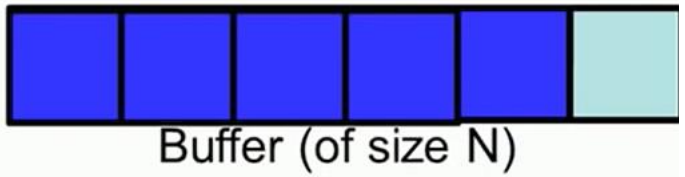
PRODUCER – CONSUMER WITH SEMAPHORES



N=6
full = 5
empty = 1

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

      item = remove_item(); // from buffer

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

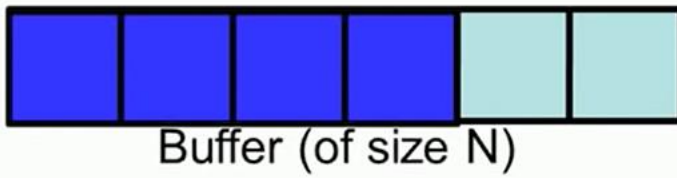
PRODUCER – CONSUMER WITH SEMAPHORES



N=6
full = 4
empty = 1

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

      item = remove_item(); // from buffer

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

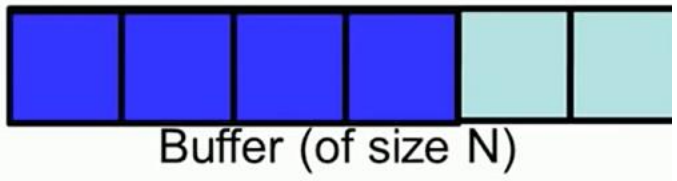
PRODUCER – CONSUMER WITH SEMAPHORES



N=6
full = 4
empty = 1

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

      item = remove_item(); // from buffer

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

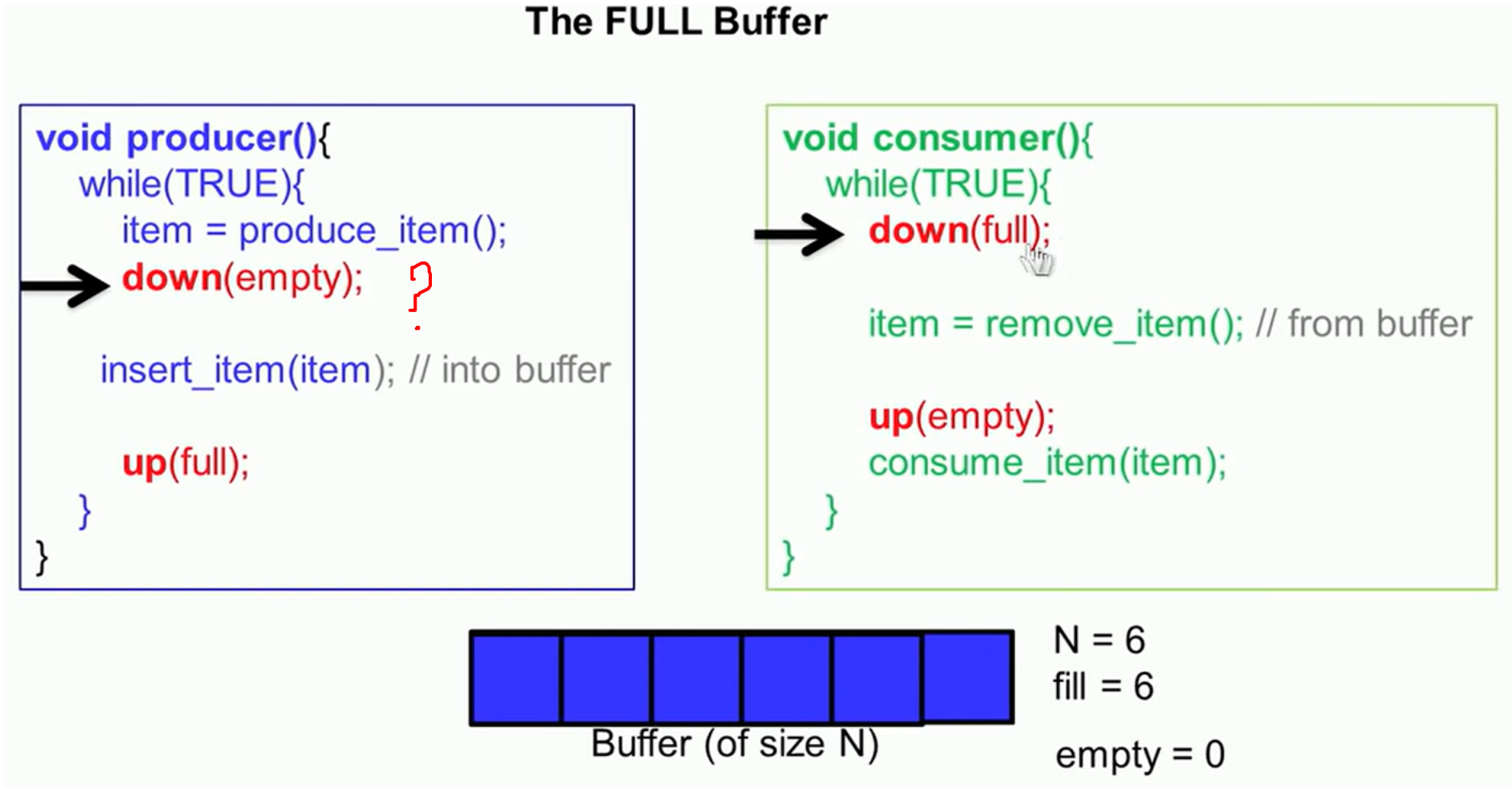
PRODUCER – CONSUMER WITH SEMAPHORES



N=6
full = 4
empty = 2

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

      item = remove_item(); // from buffer

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

PRODUCER – CONSUMER WITH SEMAPHORES

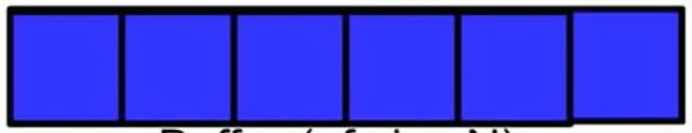


PRODUCER – CONSUMER WITH SEMAPHORES

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);
   }
}
```



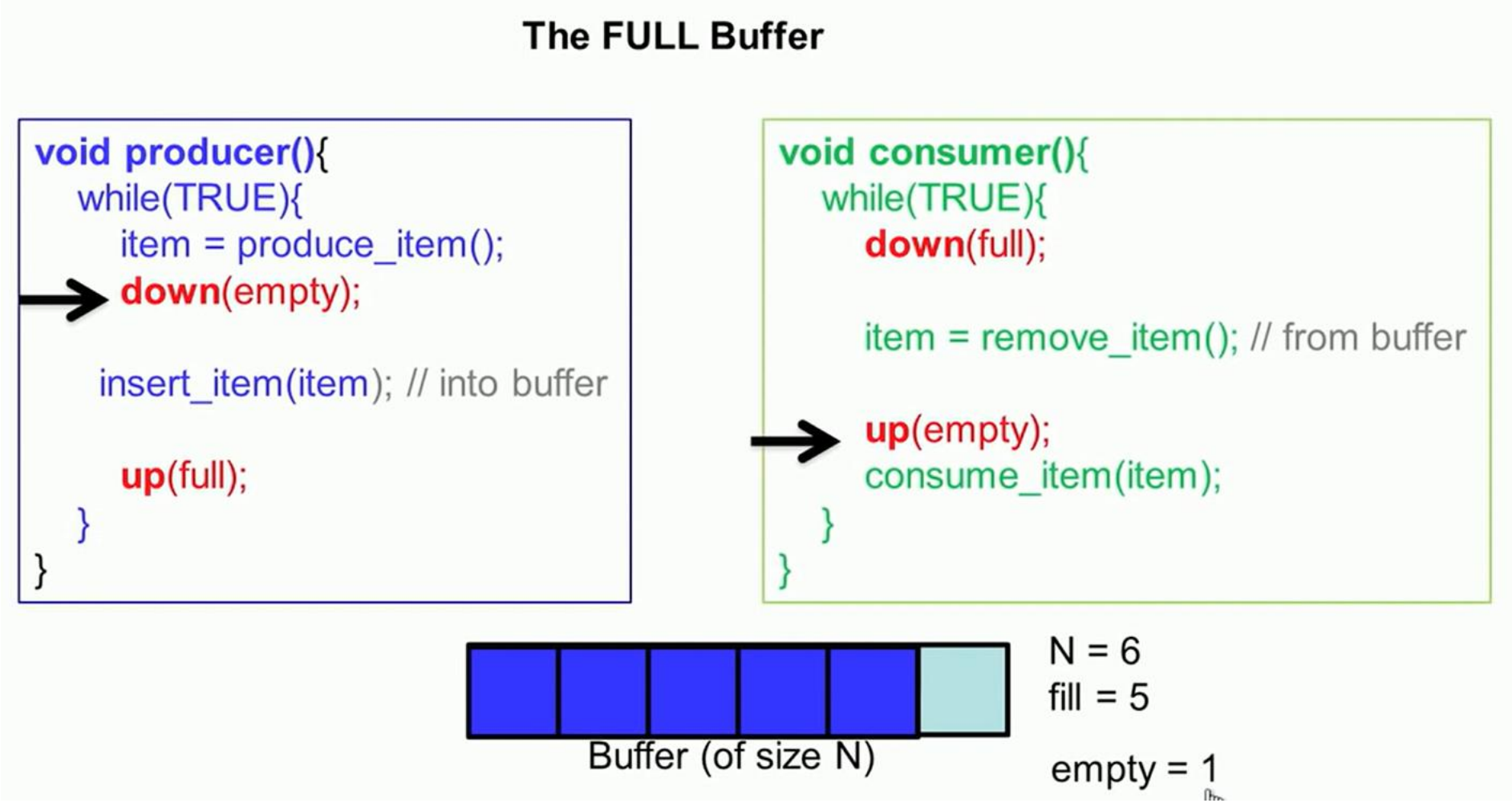
Buffer (of size N)

N = 6

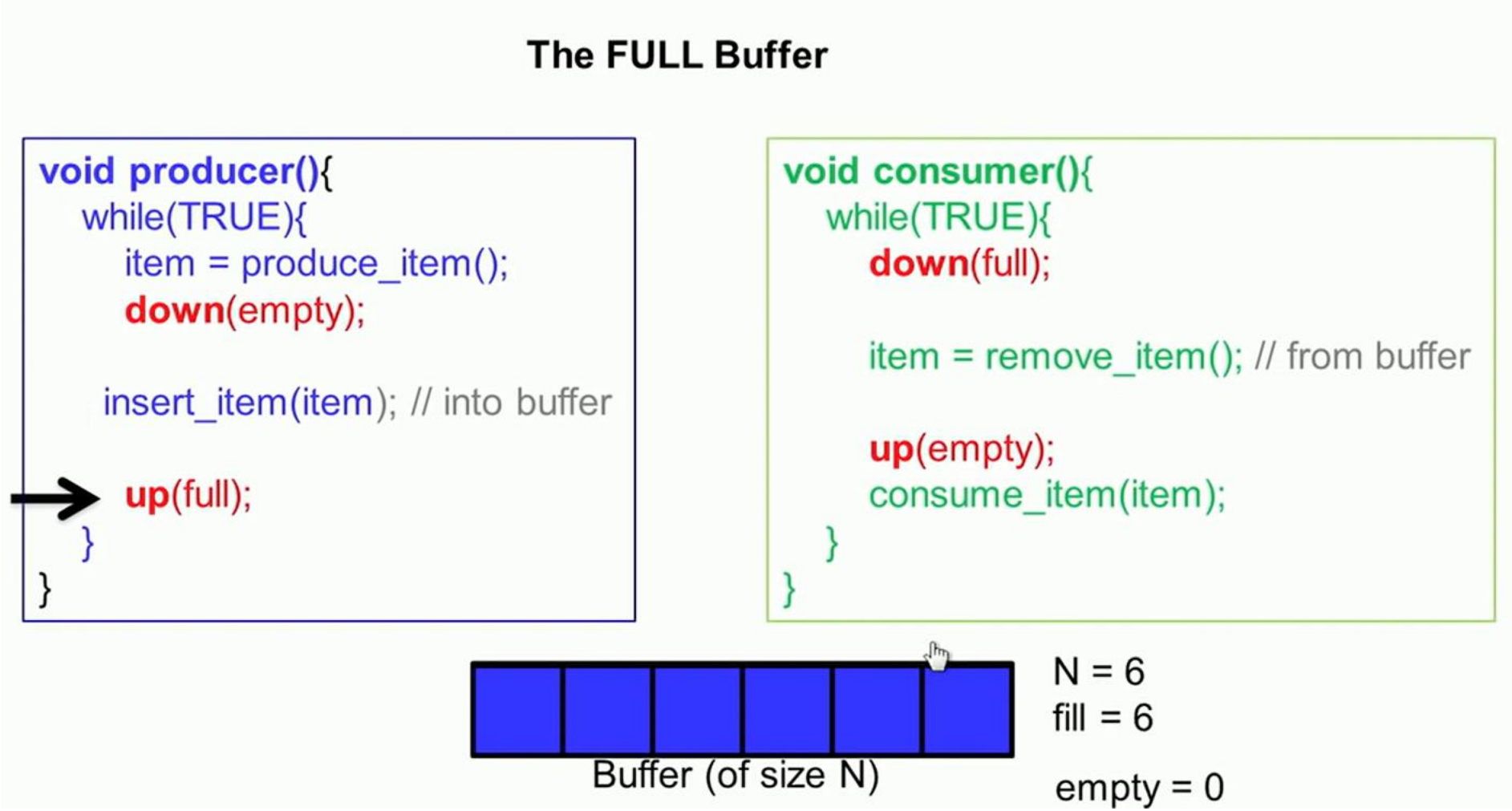
fill = 5

empty = 0

PRODUCER – CONSUMER WITH SEMAPHORES



PRODUCER – CONSUMER WITH SEMAPHORES

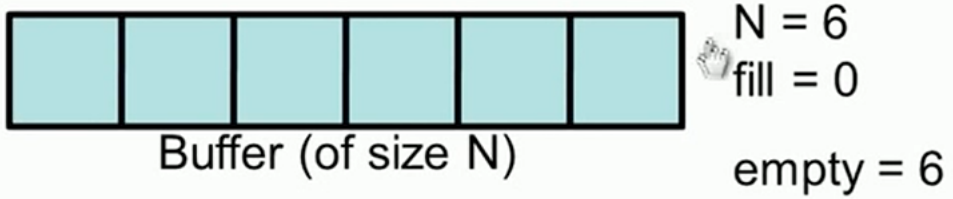


PRODUCER – CONSUMER WITH SEMAPHORES

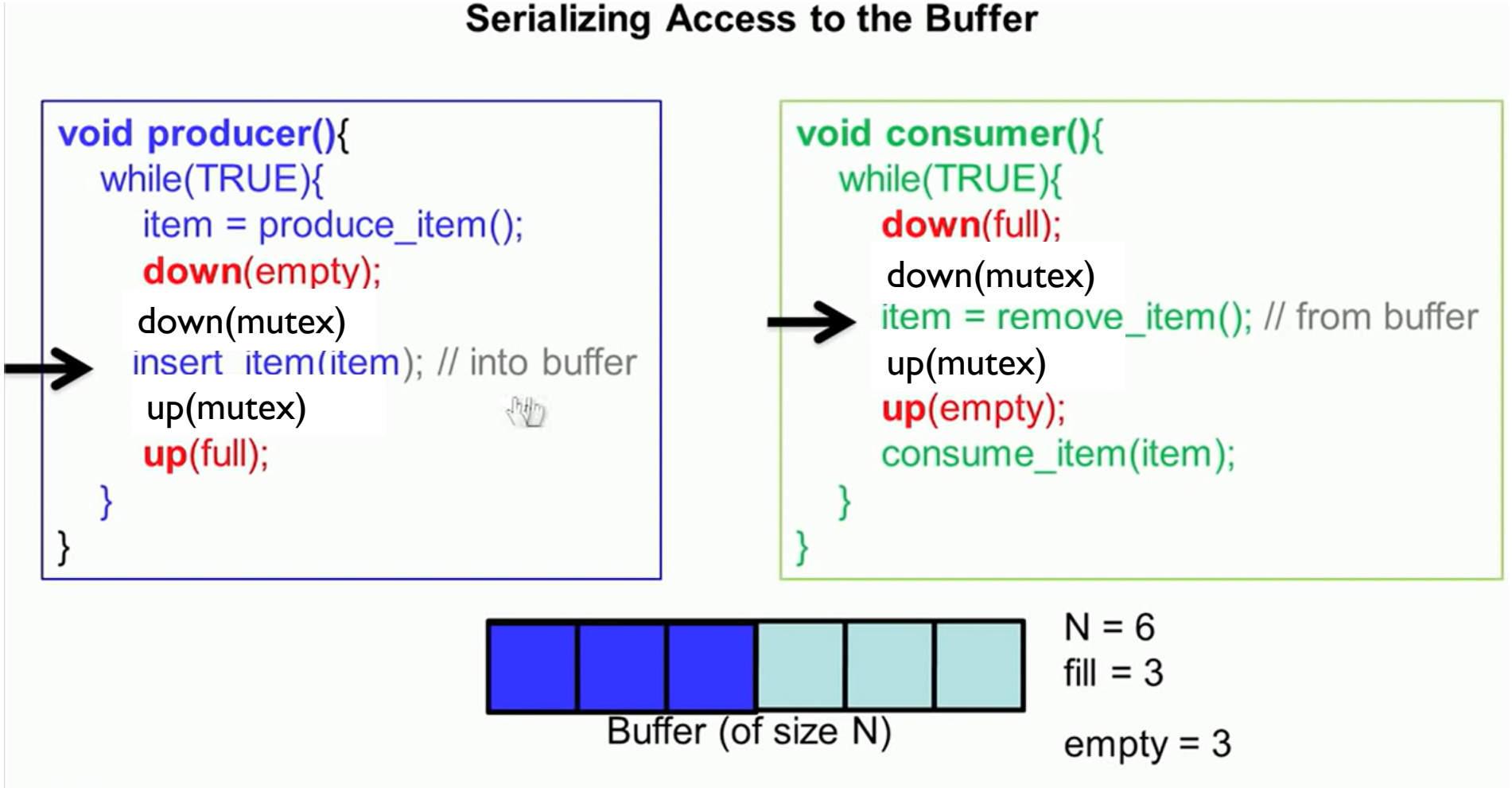
The Empty Buffer

```
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);
    }
}
```



PRODUCER – CONSUMER WITH SEMAPHORES



Question: What happens if we interchange wait(empty) and wait(mutex) in the producer code?

```
do{

//produce an item

wait(empty);        wait(mutex);
wait(mutex);        wait(empty);

//place in buffer

signal(mutex);
signal(full);

}while(true)
```

HINT: check when buffer is full.

Question: What happens if we interchange signal(full) and signal(mutex) in the producer code?

```
do{

//produce an item

wait(empty);
wait(mutex);

//place in buffer

signal(mutex);
signal(full);

}while(true)
```

```
Signal (full);
signal(mutex);
```

HINT: check when buffer is empty and full.

THANK YOU
?