BMCS3003 Distributed Systems and Parallel Computing

L04 - Concurrency Control (Part 1)

Presented by

Assoc Prof Ts Dr Tew Yiqi

May 2023

Table of contents

01

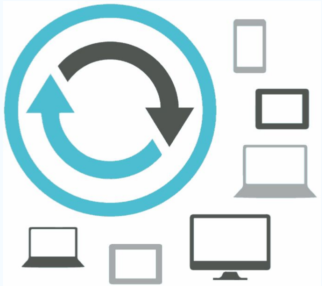
Mutual Exclusion & Critical Region

A program object that blocks multiple users from accessing the same shared variable data at the same time.

Synchronization

Parallel processing (multi-processing) – 2 or more processors operate in unison.

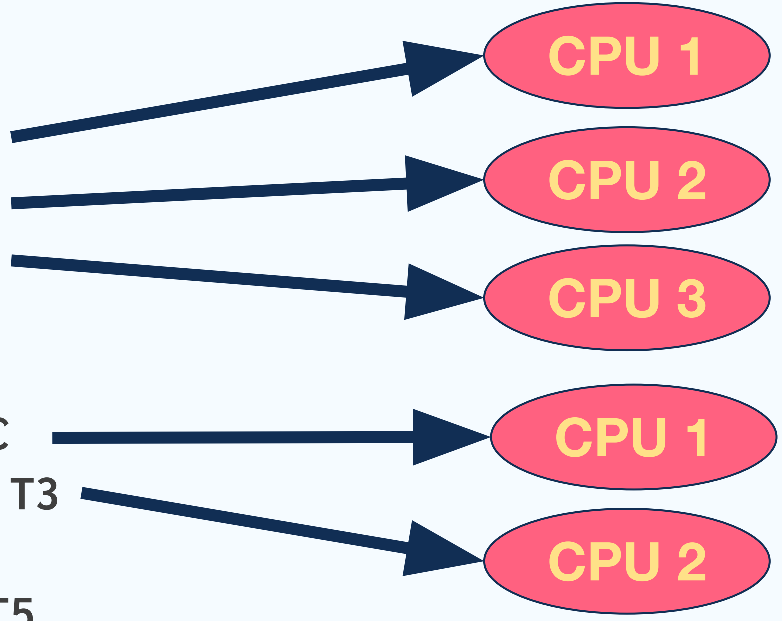
- ➢ 2 or more CPUs are executing instructions simultaneously.
- ➢ Each CPU can have a process in RUNNING state at same time.
- ➢ Processor Manager has to coordinate activity of each processor and synchronize interaction among CPUs.



Synchronization is critical to a system's success because many things can go wrong in a multiprocessing system.

Example - Application of concurrent processing

```
Compute A = 3 * B * C + 4  /  (D+E) ** (F-G)

COBEGIN
T1 = 3 * B
T2 = D+E
T3 = F-G
COEND
COBEGIN
T4 = T1 * C
T5 = T2 ** T3
COEND
A =  T4+4/T5
```



Example - Application of concurrent processing

```
DO i = 1,3
   A(i) = B(i) + C(i)
ENDDO
```

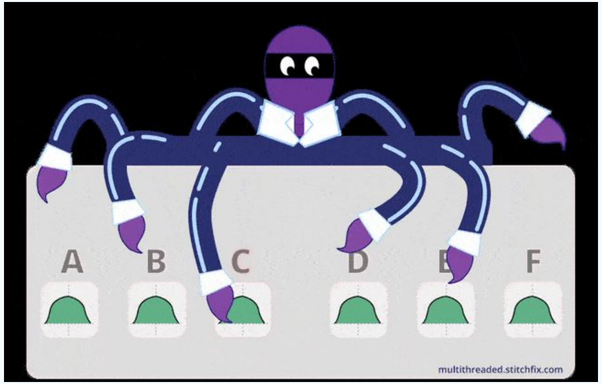
Processor 1 perform : A(1) = B(1) + C(1)

Processor 2 perform : A(2) = B(2) + C(2)

Processor 3 perform : A(3) = B(3) + C(3)

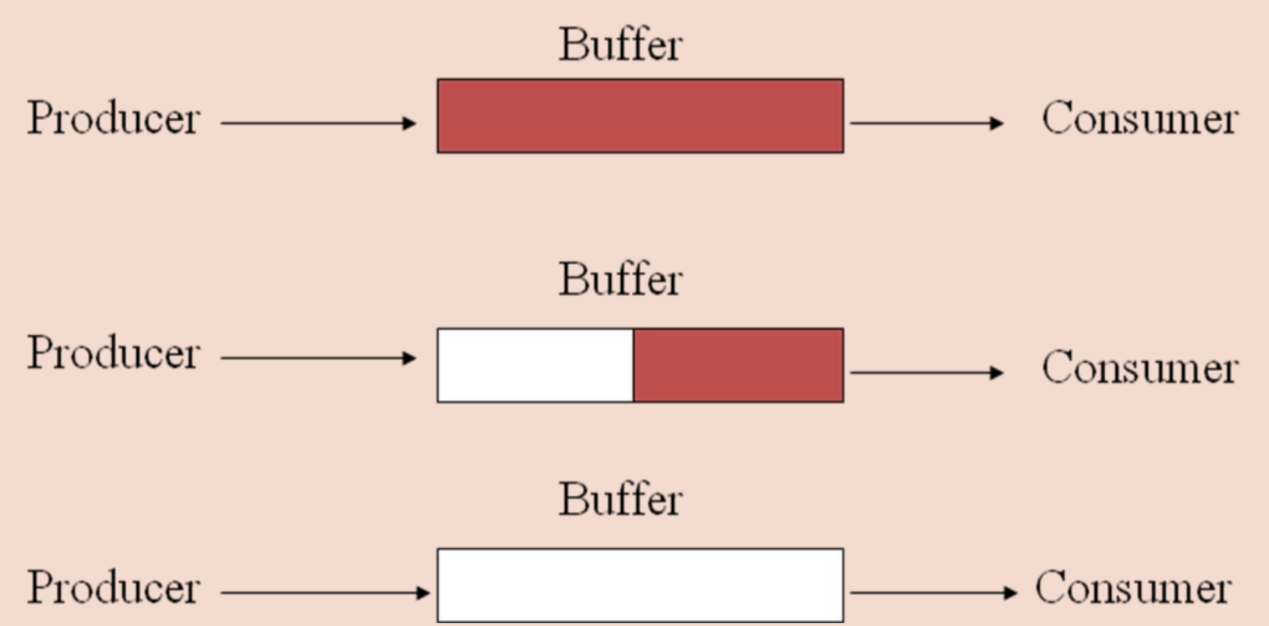
Classical Synchronisation Problems

- Common element in all synchronization schemes is to allow a process to finish work on a **critical region** of program before other processes have access to it.
  - Applicable to both multiprocessors and to 2 or more processes in a single-processor (time-shared) processing system.
  - Called a critical region because its execution must be handled as **ONE UNIT.**



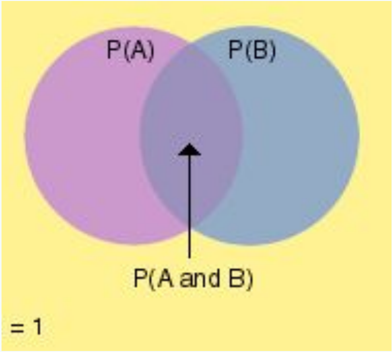
Producers and Customers

One Process Produces Some Data That Another Process Consumes Later



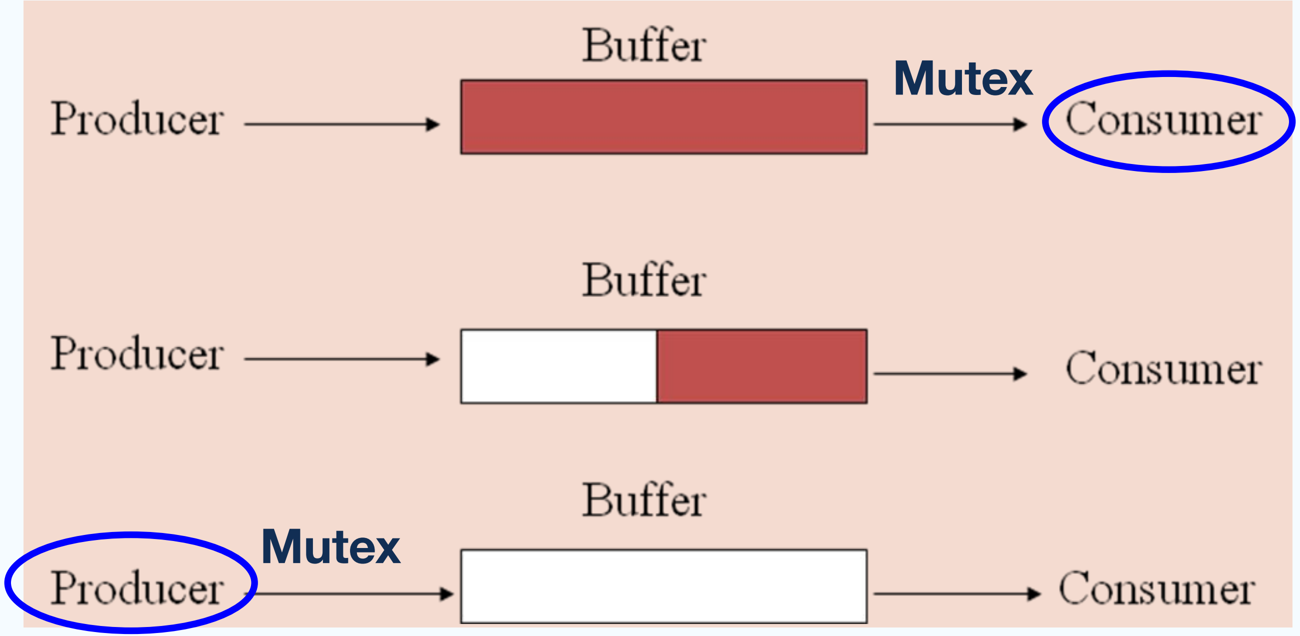
Mutual Exclusion (Mutex)

- Because buffer holds finite amount of data, synchronization process must delay producer from generating more data when buffer is full.
- Delay consumer from retrieving data when buffer is empty.
- This task can be implemented by 3 semaphores (signals):
  - Indicate number of full positions in buffer.
  - Indicate number of empty positions in buffer.
  - Mutex, will ensure mutual exclusion between processes (A **mutex** is a program object that is created so that multiple program thread can take turns sharing the same resource)



Producers and Customers

One Process Produces Some Data That Another Process Consumes Later

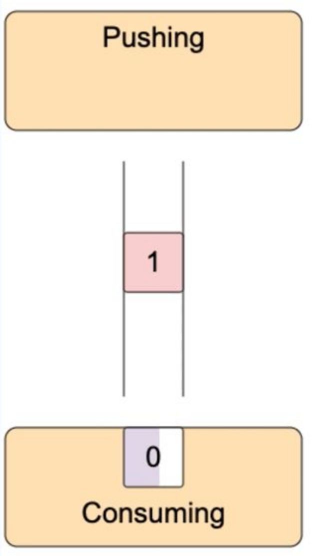


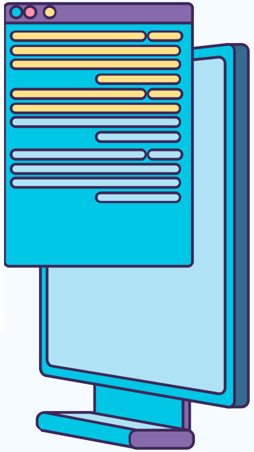
Producers and Customers

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Suppose that we wanted to provide a solution to the consumer-producer problem that fills all the buffers. We can do so by having an integer count that keeps track of the number of full buffers. Initially, count is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

Producers

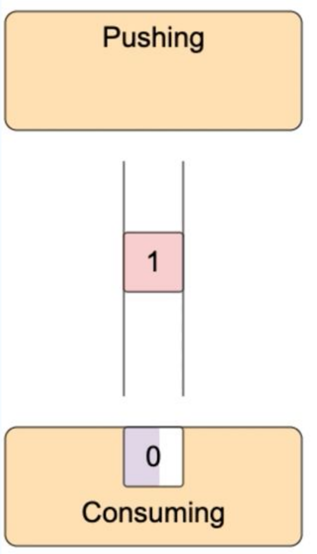
```
while (true)
{
    -- Produce an item and put in nextProduced
    while (count == BUFFER_SIZE)
        ; // do nothing
    buffer [in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    count++;
}
```

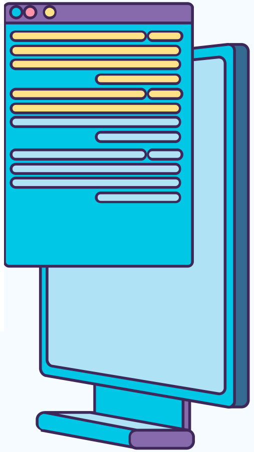


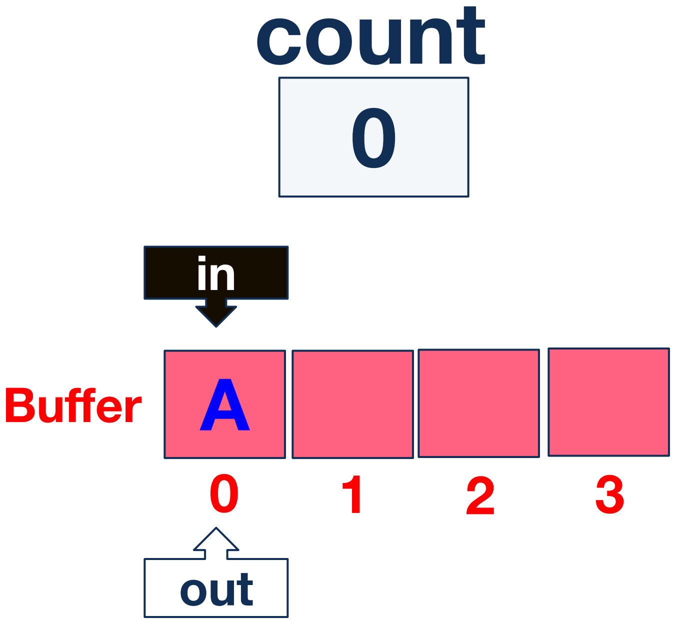


Customers

```
while (true)
{
    while (count == 0)
        ; // do nothing
    nextConsumed =  buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    count--;
    -- Consume the item in nextConsumed
}
```

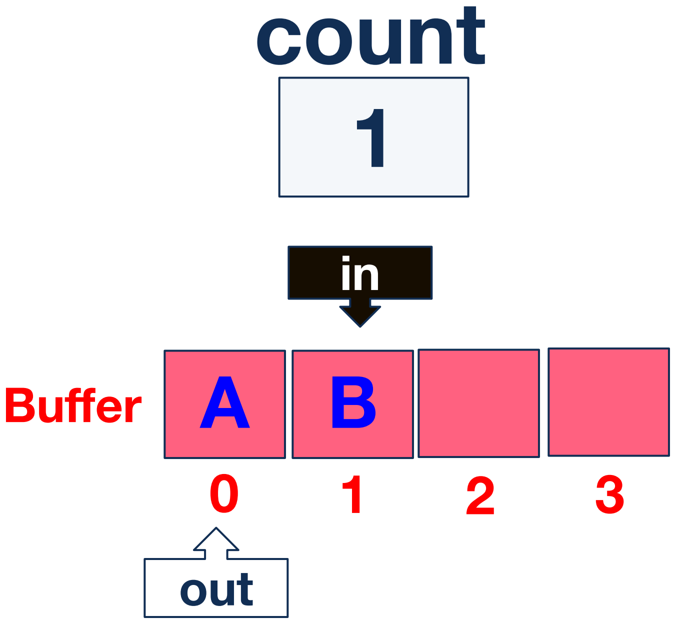






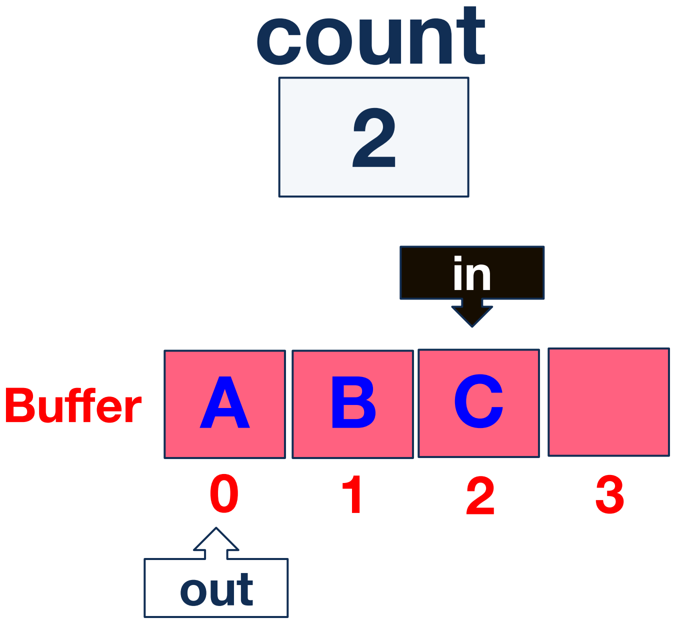
Producer

```
while (true)     {
    /* produce an item and put in nextProduced
    while (count == BUFFER_SIZE)
    ; // do nothing
    buffer [in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    count++;
  }
```



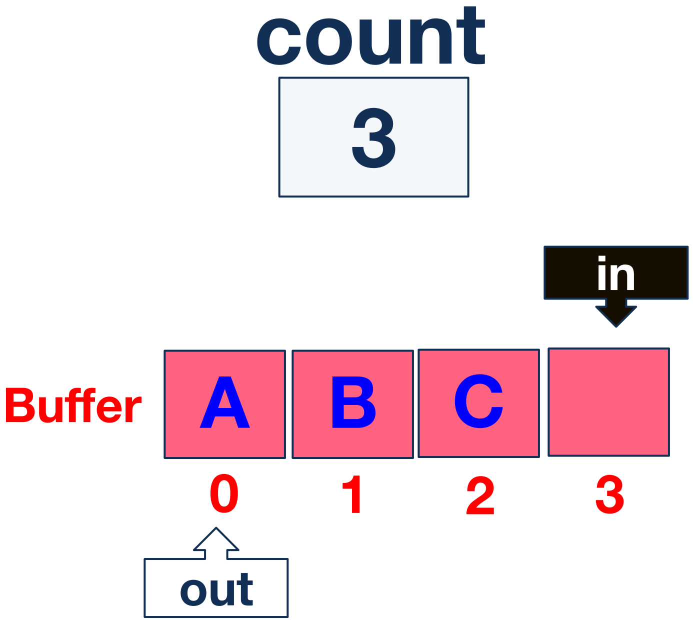
Producer

```
while (true)     {
    /* produce an item and put in nextProduced
    while (count == BUFFER_SIZE)
    ; // do nothing
    buffer [in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    count++;
   }
```



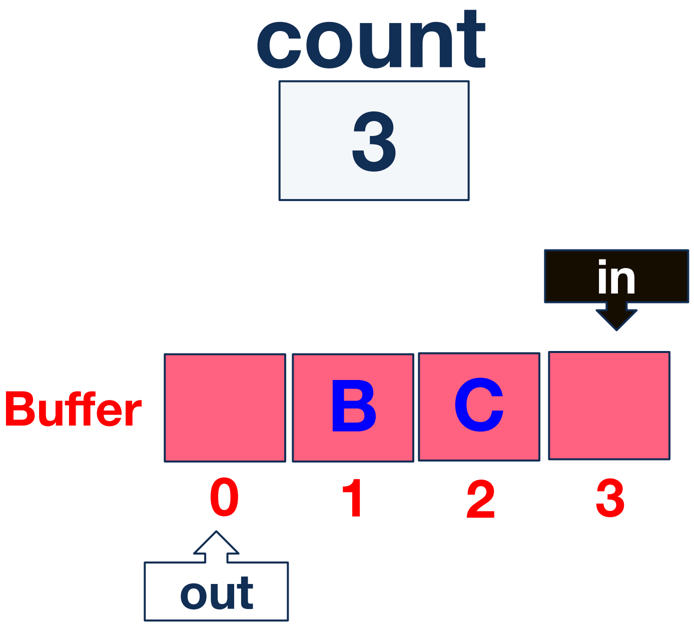
Producer

```
while (true)     {
    /* produce an item and put in nextProduced
    while (count == BUFFER_SIZE)
    ; // do nothing
    buffer [in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    count++;
   }
```



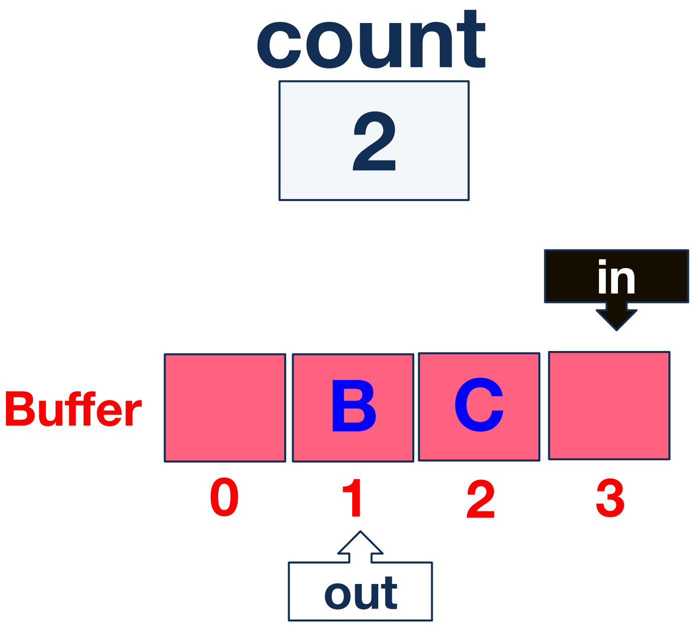
Producer

```
while (true)     {
    /* produce an item and put in nextProduced
    while (count == BUFFER_SIZE)
    ; // do nothing
    buffer [in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    count++;
  }
```



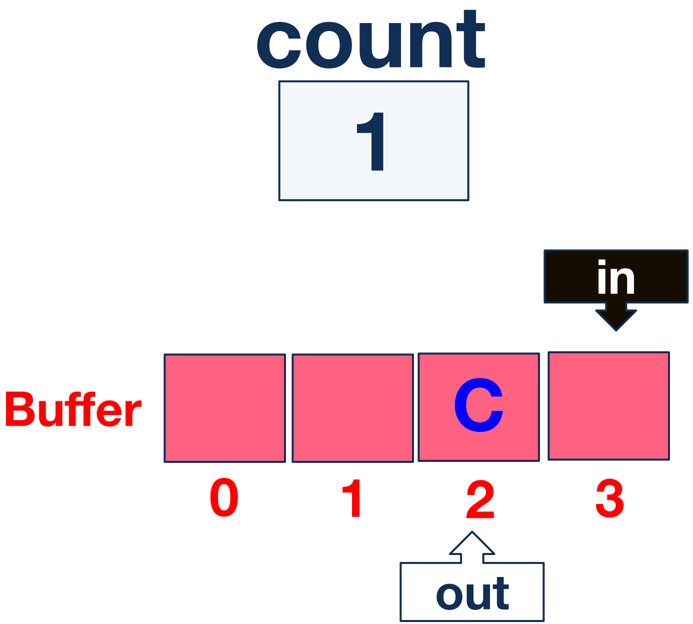
Consumer

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



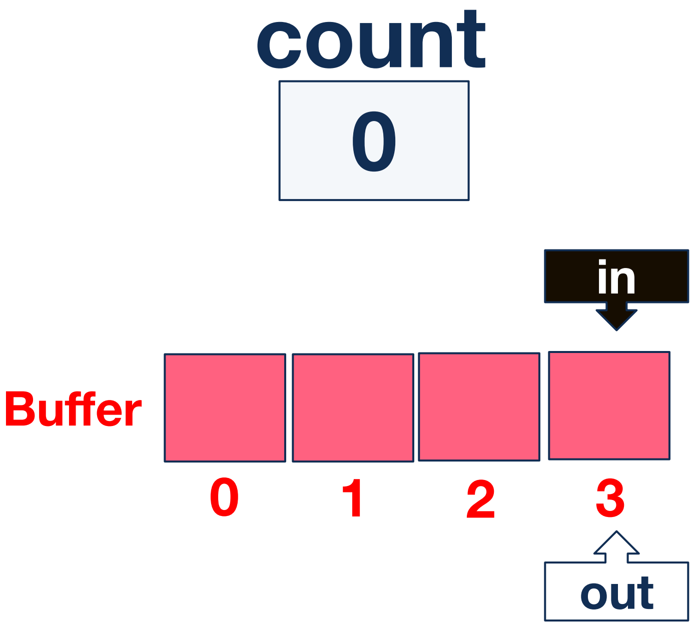
Consumer

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



Consumer

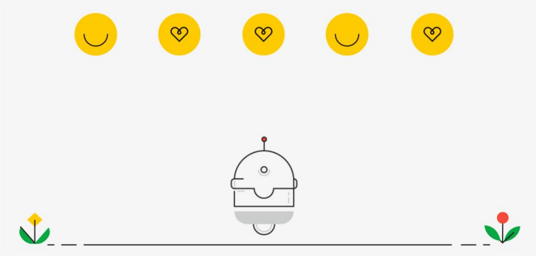
```
while (true)     {
   while (count == 0)
    ; // do nothing
   nextConsumed =  buffer[out];
   out = (out + 1) % BUFFER_SIZE;
   count--;
   /*  consume the item in nextConsumed
   }
```



Consumer

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

a. Producer-Consumer Race Condition



➢ `count++` is actually implemented as

```
register1 = count
register1 = register1 + 1
count = register1
```

➢ `count--` is actually implemented as

```
register2 = count
register2 = register2 - 1
count = register2
```

Consider this execution interleaving with "count = 5" initially:

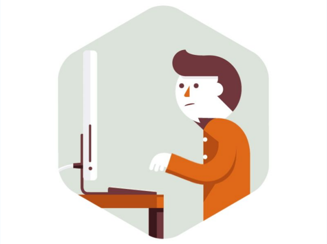
- t0: producer execute register1 = count {register1 = 5}
- t1: producer execute register1 = register1 + 1 {register1 = 6}
- t2: consumer execute register2 = count {register2 = 5}
- t3: consumer execute register2 = register2 - 1 {register2 = 4}
- t4: producer execute count = register1 {count = 6 }
- t5: consumer execute count = register2 {count = 4}

This is a problem! Count should be always 5!

Illustration

Count

5



Producer

```
register1  = count
register1  = register1 + 1
count      = register1
```

Consumer

t0

```
register2  = count
register2  = register2 - 1
count      = register2
```

Illustration

Count

5

Producer

```
register1  = count
register1  = register1 + 1  ⬅ t1
count      = register1
```

Consumer

```
register2  = count
register2  = register2 - 1
count      = register2
```

Illustration

Count

5

Producer

```
register1  = count
register1  = register1 + 1
count      = register1
```

Consumer

t2 ➡

```
register2  = count
register2  = register2 - 1
count      = register2
```

Illustration

Count

5

Producer

```
register1  = count
register1  = register1 + 1
count      = register1
```

t3 ➡

Consumer

```
register2  = count
register2  = register2 - 1
count      = register2
```

Illustration

Count

6

Producer

```
register1  = count
register1  = register1 + 1
count      = register1
```

⬅ **t4**

Consumer

```
register2  = count
register2  = register2 - 1
count      = register2
```

Illustration

Count

4

Producer

```
register1  = count
register1  = register1 + 1
count      = register1
```

Consumer

```
register2  = count
register2  = register2 - 1
count      = register2
```

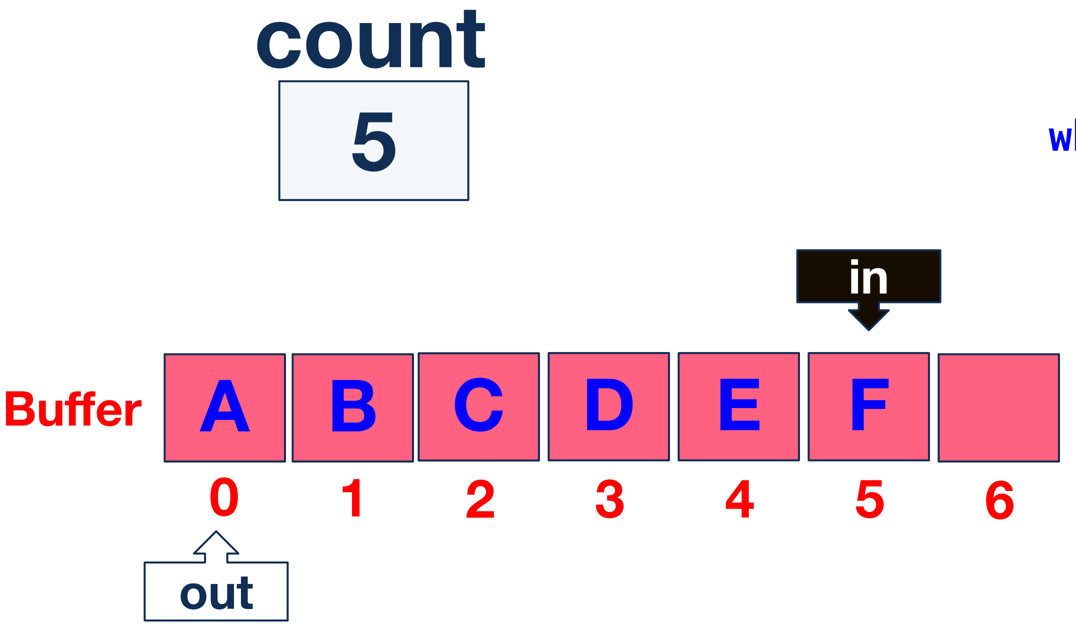
t5 ➡ count = register2

Elaboration

- If the producer and consumer run concurrently the value of the variable counter may be 4 or 6 (the value of counter is 6 when we reversed the order of the statements at t4 and t5)
- Incorrect state occurs because we allowed both processes to manipulate the variable counter concurrently.
- Several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which access take place is called a Race Condition

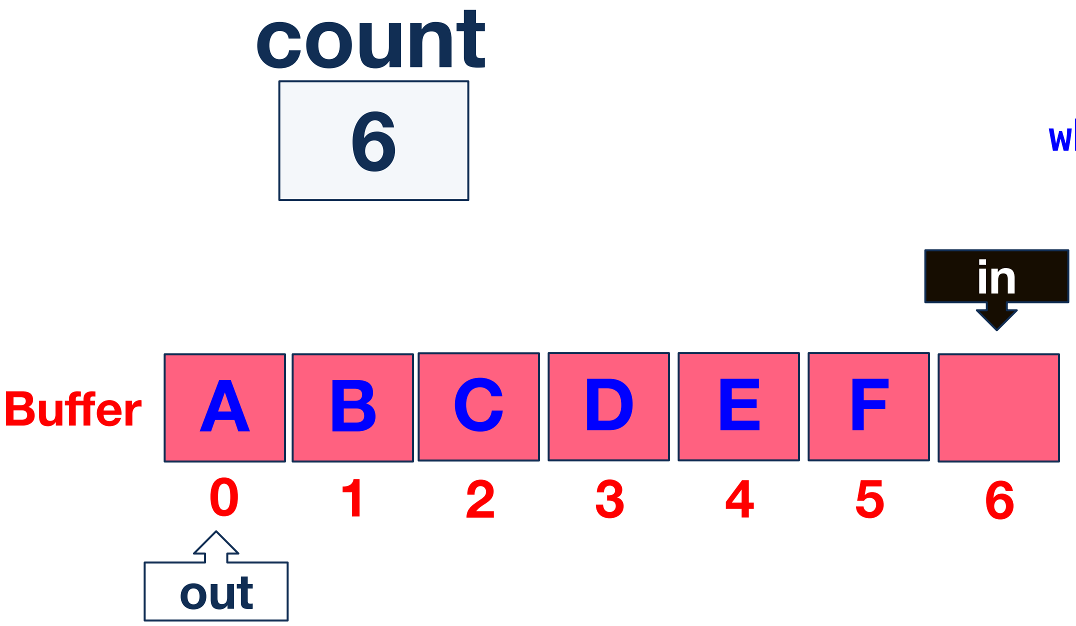
Elaboration

- In fact, the producer program has to finish it execution then the consumer program to print or display the result, so the counter value suppose is 5. Which is generated correctly if producer and consumer execute separately.



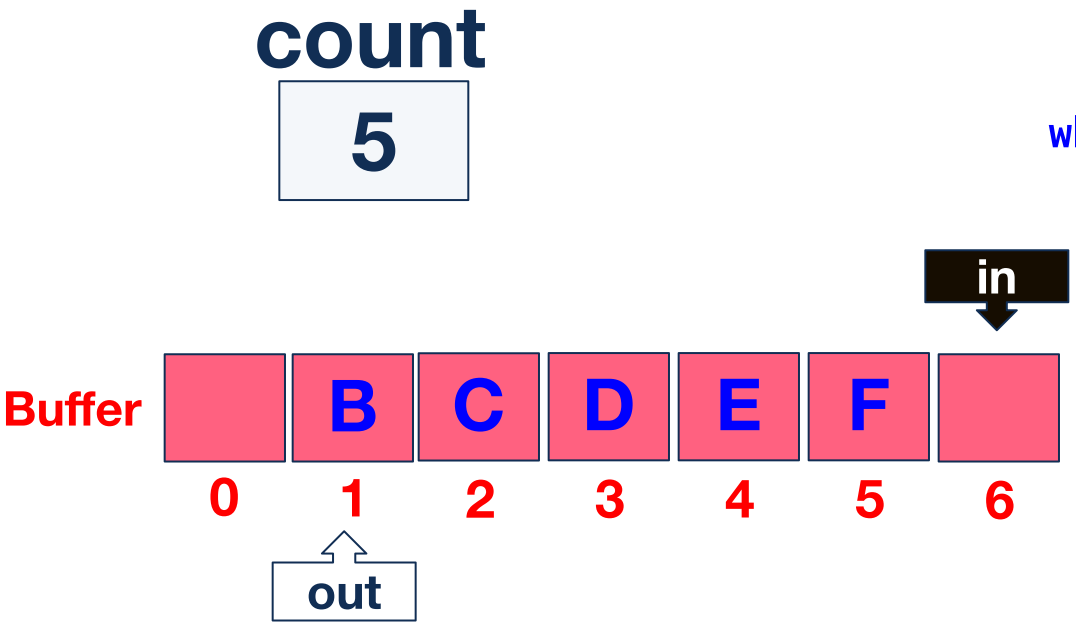
Producer

```
while (true)     {
  /* produce an item and put in
 nextProduced
  while (count == BUFFER_SIZE)
  ; // do nothing
  buffer [in] = nextProduced;
  in = (in + 1) % BUFFER_SIZE;
  count++;
 }
```



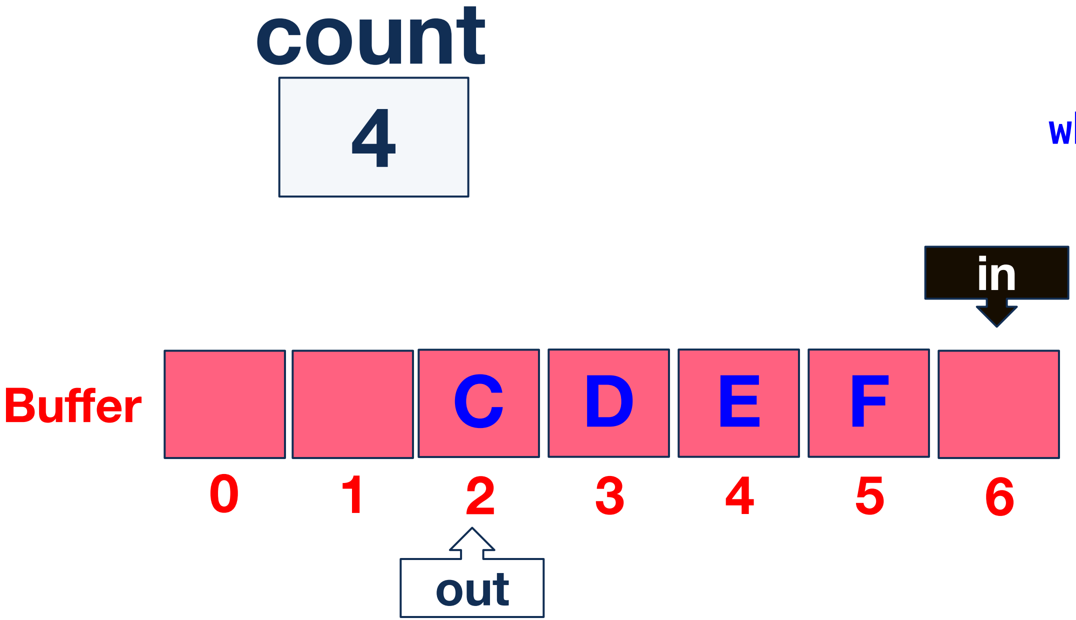
Producer

```
while (true)     {
  /* produce an item and put in
 nextProduced
  while (count == BUFFER_SIZE)
  ; // do nothing
  buffer [in] = nextProduced;
  in = (in + 1) % BUFFER_SIZE;
  count++;
 }
```



Consumer

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



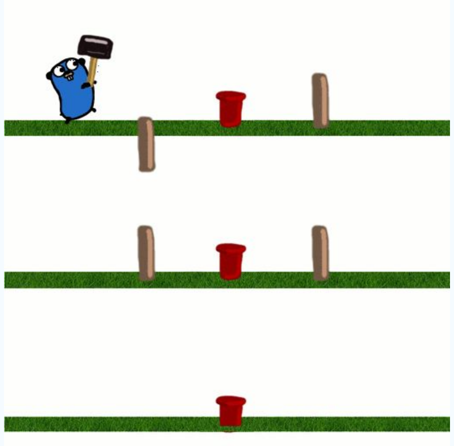
Consumer

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

b. Print Spooler Race Condition

* A **daemon** is a program that waits for another program to ask it to do something.

- ➢ A process wants to print a file, it enter the file name in spooler directory.
- ➢ Another process, the printer daemon*, periodically checks to see if there are any files to be printed, then prints them and removes their names from the directory.
- ➢ Imagine that our spooler directory has very large number of slots, numbered 0,1,2,3,..., each one capable of holding a file name.
- ➢ There are two shared variables `out` and `in`.
  - `out` which points to the next file to be printed.
  - `in` is point to next free slot in directory.
- ➢ Process A and Process B decide they want to queue a file for printing.



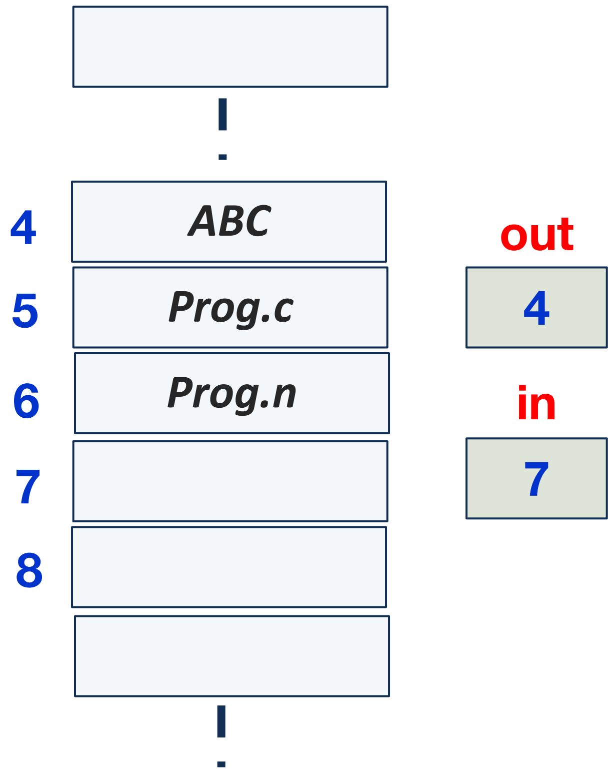
Process A

```
read in
store into A_NextFreeSlot
add file a into the buffer
 in = A_NextFreeSlot + 1
```

Process B

```
read in
store into B_NextFreeSlot
add file b into the buffer
 in = B_NextFreeSlot + 1
```

Spooler Directory/ buffer



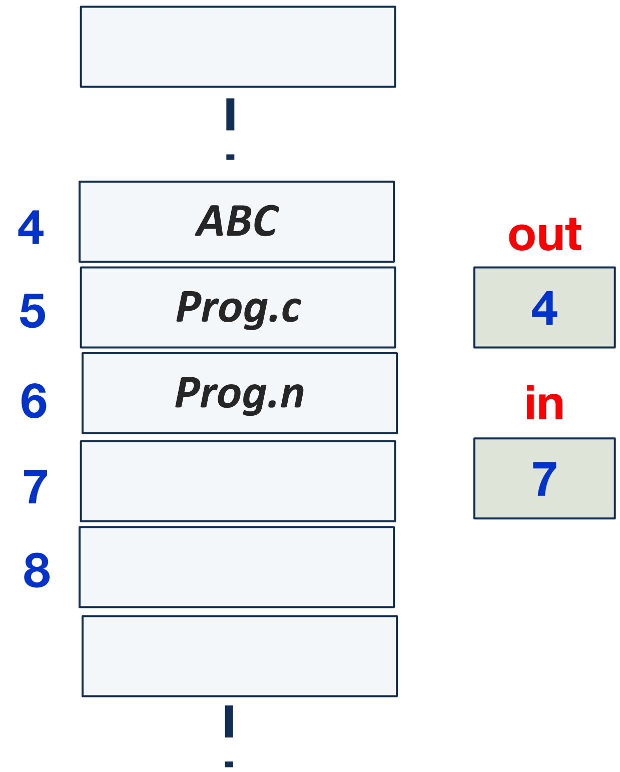
Process A

```
read in
                           7
store into A_NextFreeSlot
add file a into the buffer
 in = A_NextFreeSlot + 1
```

Process B

```
read in
store into B_NextFreeSlot
add file b into the buffer
 in = B_NextFreeSlot + 1
```

Spooler Directory/ buffer



Process A

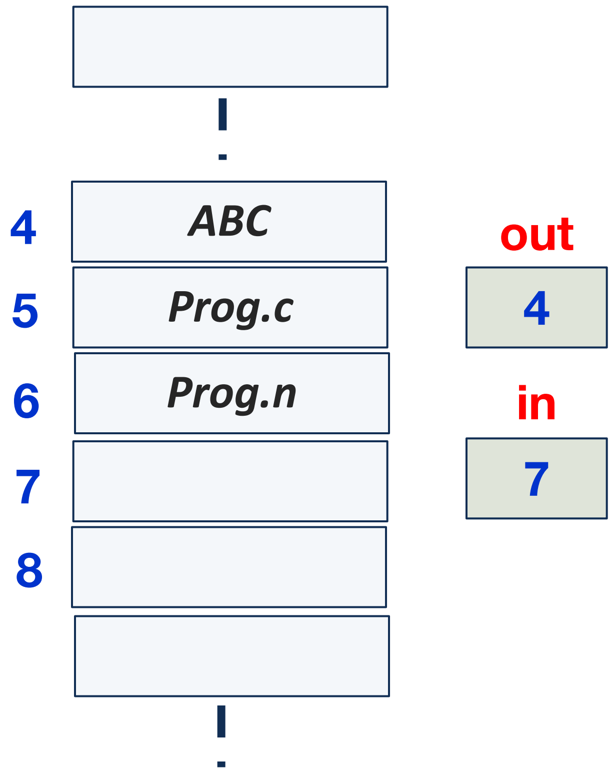
```
read in
store into A_NextFreeSlot
add file a into the buffer
 in = A_NextFreeSlot + 1
```

7

Process B

```
read in
store into B_NextFreeSlot
add file b into the buffer
 in = B_NextFreeSlot + 1
```

Spooler Directory/ buffer



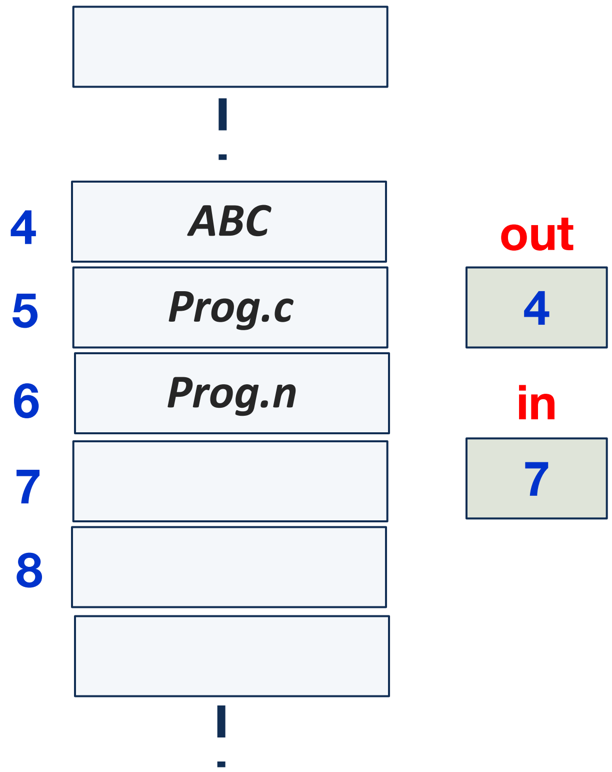
Process A

```
read in
                          7
store into A_NextFreeSlot
add file a into the buffer
 in = A_NextFreeSlot + 1
```

Process B

```
read in
store into B_NextFreeSlot
add file b into the buffer
 in = B_NextFreeSlot + 1
```

Spooler Directory/ buffer



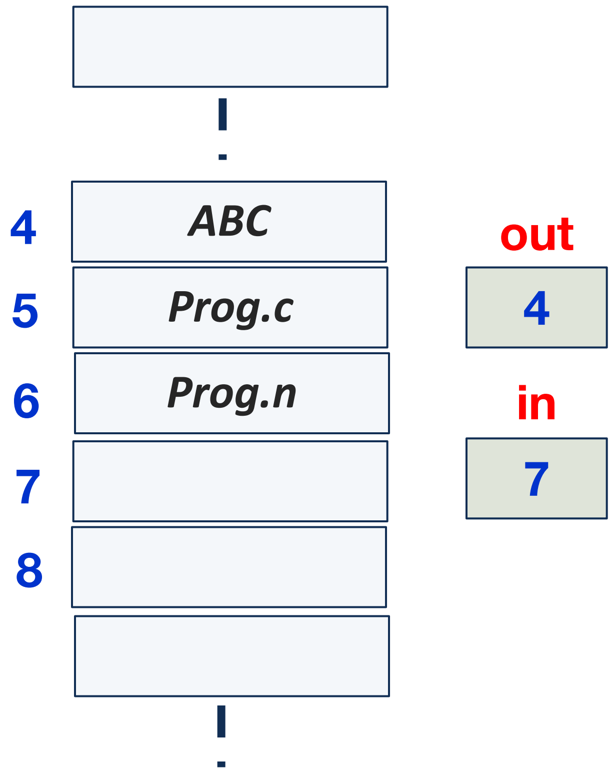
Process A

```
read in
                          7
store into A_NextFreeSlot
add file a into the buffer
 in = A_NextFreeSlot + 1
```

Process B

```
read in
                          7
store into B_NextFreeSlot
add file b into the buffer
 in = B_NextFreeSlot + 1
```

Spooler Directory/ buffer



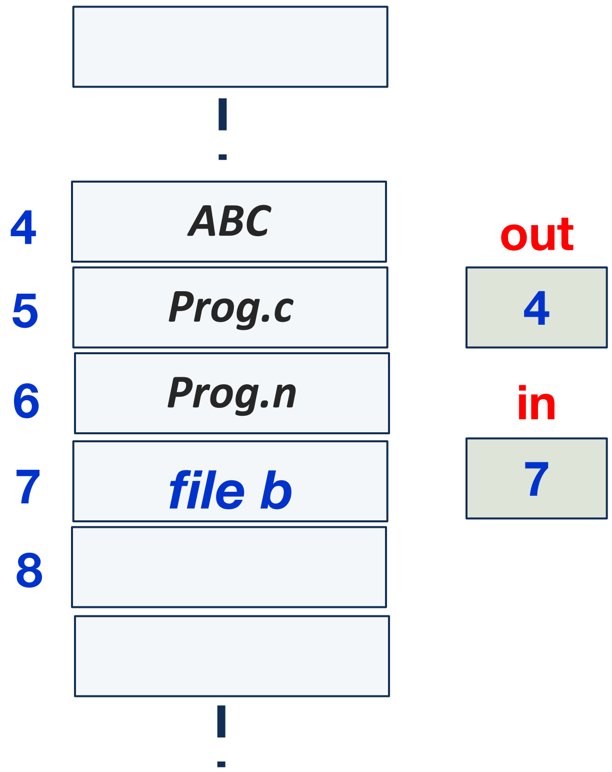
Process A

```
read in
                                       7
store into A_NextFreeSlot
add file a into the buffer
 in = A_NextFreeSlot + 1
```

Process B

```
read in
                                       7
store into B_NextFreeSlot
add file b into the buffer
 in = B_NextFreeSlot + 1
```

Spooler Directory/ buffer



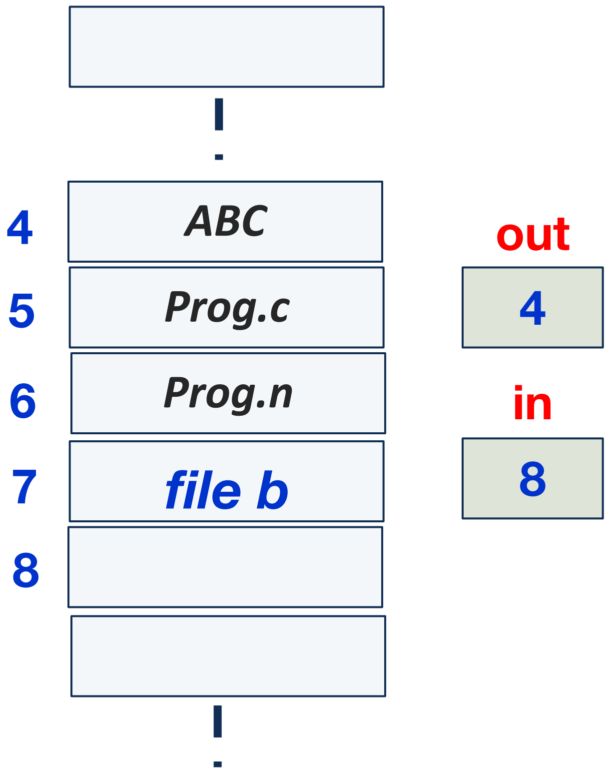
Process A

```
read in
                                  7
store into A_NextFreeSlot
add file a into the buffer
 in = A_NextFreeSlot + 1
```

Process B

```
read in
store into B_NextFreeSlot
add file b into the buffer
 in = B_NextFreeSlot + 1
```

Spooler Directory/ buffer



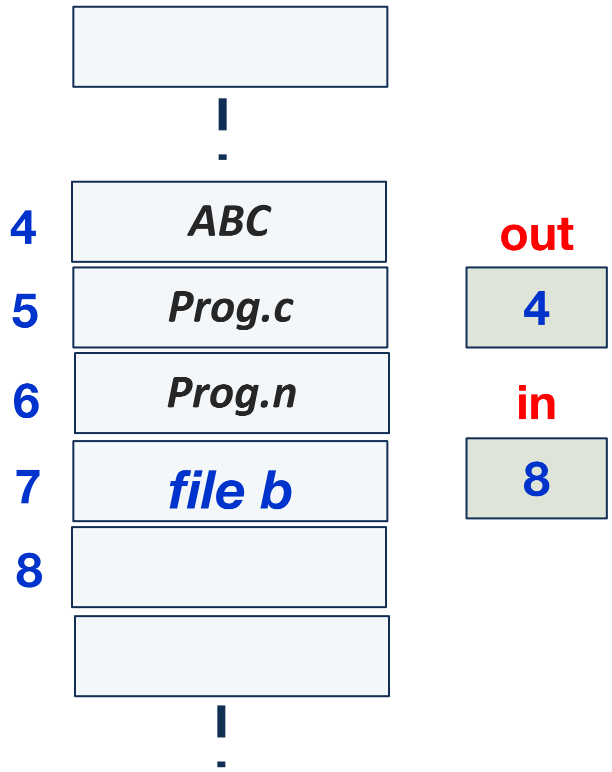
Process A

```
read in
store into A_NextFreeSlot   7
add file a into the buffer
 in = A_NextFreeSlot + 1
```

Process B

```
read in
store into B_NextFreeSlot
add file b into the buffer
 in = B_NextFreeSlot + 1
```

Spooler Directory/ buffer



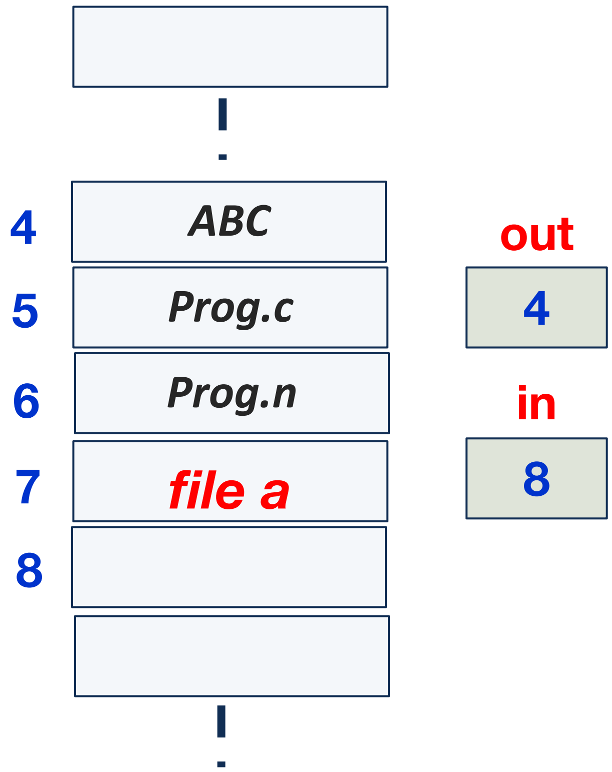
Process A

```
read in
                              7
store into A_NextFreeSlot
add file a into the buffer
 in = A_NextFreeSlot + 1
```

Process B

```
read in
store into B_NextFreeSlot
add file b into the buffer
 in = B_NextFreeSlot + 1
```

Spooler Directory/ buffer



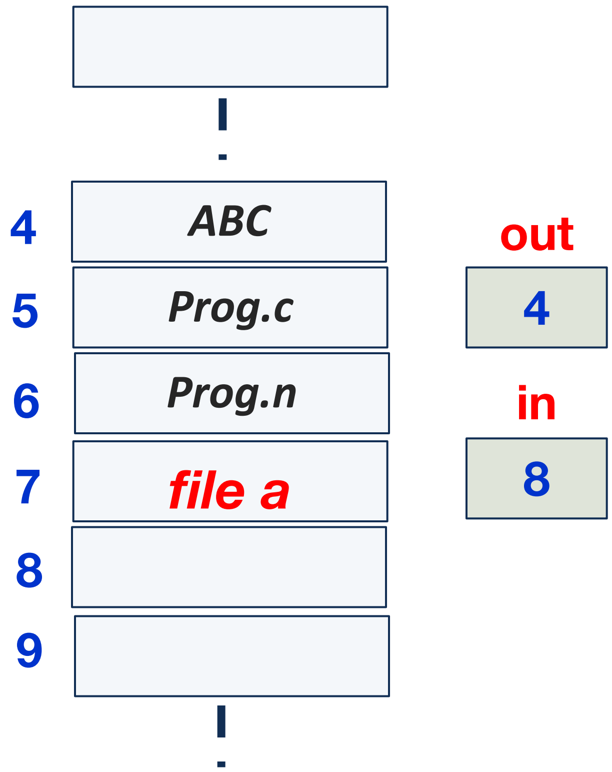
Process A

```
read in
store into A_NextFreeSlot
add file a into the buffer
 in = A_NextFreeSlot + 1
```

Process B

```
read in
store into B_NextFreeSlot
add file b into the buffer
 in = B_NextFreeSlot + 1
```

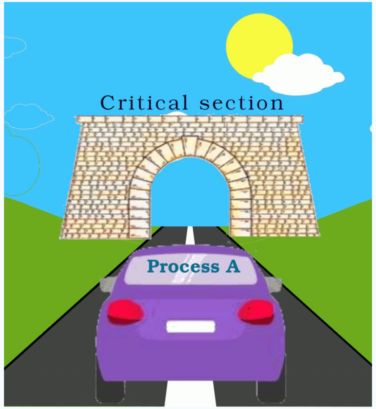
Spooler Directory/ buffer



Process B will never receive any output

The Critical Section Problem

- Each process has a segment of code called a critical section, in which the process may be changing common variables, updating a table, writing a file and so on.
- The important feature of the system is have to ensure that no other processes are executing its critical section when one particular process is executing its critical section.
- Processes cooperation protocol, whereby each process must request permission to enter its critical section. There is the entry section, then followed by exit section after that do the remaining code in the remainder section.



The Critical Section Problem

Example:

Shared data:

Where is the Critical-section ?

```
int Balance;
```

Process A:

```
……
enter section

Balance = Balance - 200;

exit section
remainder section
```

Process B:

```
……
entry section

Balance = Balance - 100;

exit section
remainder section
```

The Critical Section Problem

Example:

Shared data:

```
int Balance;
```

Process A:	**Process B:**
……	……
enter section	**entry section**
Balance = Balance - 200;	**Balance = Balance - 100;**
exit section	**exit section**
remainder section	remainder section

Critical-section

Synchronization solutions

Solution to Critical-Section requirements

1. **Mutual Exclusion** - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.
2. **Progress** - If no process is executing in its critical section and there exists some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely (unclear reasons).
3. **Bounded Waiting** - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted
   - Assume that each process executes at a nonzero speed
   - No assumption concerning relative speed of the N processes

Peterson's Algorithm

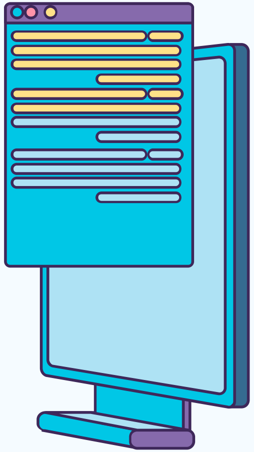
In Process synchronization

- Two process solution
- Assume that the LOAD and STORE instructions are atomic; that is, cannot be interrupted (privileged instructions).
- The two processes share two variables:
  - int turn;
  - Boolean flag[2]
- The variable turn indicates whose turn it is to enter the critical section.
- The flag array is used to indicate if a process is ready to enter the critical section. flag[i] = true implies that process P_i is ready!

Check the resources on C : geeksforgeeks.org

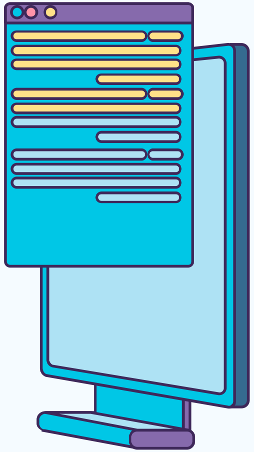
Critical Section

```
do {

    Entry section

    Critical section

    Exit section

    Remainder section

} while (TRUE);
```



Algorithm for Process P_i

```
do {
    flag[i] = TRUE;                       Entry section
    turn = j;
    while (flag[j] && turn == j);
    Critical section
    flag[i] = FALSE;                      Exit section
    Remainder section
} while (TRUE);
```



Algorithm for Process P_i

flag | FALSE | FALSE |

```
do {
      flag[i] = TRUE;
      turn = j;
      while (flag[j] && turn == j);

      Critical section

      flag[i] = FALSE;

      Remainder section

} while (TRUE);
```

P_0

```
do {
      flag[i] = TRUE;
      turn = j;
      while (flag[j] && turn == j);

      Critical section

      flag[i] = FALSE;

      Remainder section

} while (TRUE);
```

P_1

Algorithm for Process P_i

flag	TRUE	FALSE

```
do {
      flag[i] = TRUE;
      turn = j;
      while (flag[j] && turn == j);
      Critical section
      flag[i] = FALSE;
      Remainder section
} while (TRUE);
```

P_0

```
do {
      flag[i] = TRUE;
      turn = j;
      while (flag[j] && turn == j);
      Critical section
      flag[i] = FALSE;
      Remainder section
} while (TRUE);
```

P_1

Algorithm for Process P_i

flag	TRUE	FALSE

```
do {
      flag[i] = TRUE;
      turn = j;
      while (flag[j] && turn == j);

      Critical section

      flag[i] = FALSE;

      Remainder section

} while (TRUE);
```

P_0

```
do {
      flag[i] = TRUE;
      turn = j;
      while (flag[j] && turn == j);

      Critical section

      flag[i] = FALSE;

      Remainder section

} while (TRUE);
```

P_1

Algorithm for Process P_i

flag

TRUE	TRUE

```
do {
    flag[i] = TRUE;
    turn = j;
    while (flag[j] && turn == j);
    Critical section
    flag[i] = FALSE;
    Remainder section
} while (TRUE);
```

P_0

```
do {
    flag[i] = TRUE;
    turn = j;
    while (flag[j] && turn == j);
    Critical section
    flag[i] = FALSE;
    Remainder section
} while (TRUE);
```

P_1

Algorithm for Process P_i

flag | TRUE | TRUE |

```
do {
    flag[i] = TRUE;
    turn = j;
    while (flag[j] && turn == j);
    Critical section
    flag[i] = FALSE;
    Remainder section
} while (TRUE);
```

P_0

```
do {
    flag[i] = TRUE;
    turn = j;
    while (flag[j] && turn == j);
    Critical section
    flag[i] = FALSE;
    Remainder section
} while (TRUE);
```

P_1

Algorithm for Process P_i

flag	TRUE	TRUE

```
do {
      flag[i] = TRUE;
      turn = j;
      while (flag[j] && turn == j);
      Critical section
      flag[i] = FALSE;
      Remainder section
} while (TRUE);
```

P_0

```
do {
      flag[i] = TRUE;
      turn = j;
      while (flag[j] && turn == j);
      Critical section
      flag[i] = FALSE;
      Remainder section
} while (TRUE);
```

P_1

Algorithm for Process P_i

flag	FALSE	TRUE

```
do {
      flag[i] = TRUE;
      turn = j;
      while (flag[j] && turn == j);

      Critical section

      flag[i] = FALSE;

      Remainder section

} while (TRUE);
```

P_0

```
do {
      flag[i] = TRUE;
      turn = j;
      while (flag[j] && turn == j);

      Critical section

      flag[i] = FALSE;

      Remainder section

} while (TRUE);
```

P_1

Algorithm for Process P_i

flag

FALSE	TRUE

```
do {
      flag[i] = TRUE;
      turn = j;
      while (flag[j] && turn == j);

      Critical section

      flag[i] = FALSE;

      Remainder section

} while (TRUE);
```

P_0

```
do {
      flag[i] = TRUE;
      turn = j;
      while (flag[j] && turn == j);

      Critical section

      flag[i] = FALSE;

      Remainder section

} while (TRUE);
```

P_1

Algorithm for Process P_i

flag		
flag	FALSE	TRUE

```
do {
      flag[i] = TRUE;
      turn = j;
      while (flag[j] && turn == j);
      Critical section
      flag[i] = FALSE;
      Remainder section
} while (TRUE);
```

P_0

```
do {
      flag[i] = TRUE;
      turn = j;
      while (flag[j] && turn == j);
      Critical section
      flag[i] = FALSE;
      Remainder section
} while (TRUE);
```

P_1

Algorithm for Process P_i

flag	FALSE	FALSE

```
do {
      flag[i] = TRUE;
      turn = j;
      while (flag[j] && turn == j);

      Critical section

      flag[i] = FALSE;

      Remainder section

} while (TRUE);
```

P_0

```
do {
      flag[i] = TRUE;
      turn = j;
      while (flag[j] && turn == j);

      Critical section

      flag[i] = FALSE;

      Remainder section

} while (TRUE);
```

P_1

Algorithm for Process P_i

flag	FALSE	FALSE

```
do {
      flag[i] = TRUE;
      turn = j;
      while (flag[j] && turn == j);
      Critical section
      flag[i] = FALSE;
      Remainder section
} while (TRUE);
```

P_0

```
do {
      flag[i] = TRUE;
      turn = j;
      while (flag[j] && turn == j);
      Critical section
      flag[i] = FALSE;
      Remainder section
} while (TRUE);
```

P_1

Synchronization Hardware (1/3)

Disable Interrupt

- The critical-section problem could be solved simply in a uniprocessor environment, if we could prevent interrupts from occurring while a shared variable was being modified.
- The current sequence of instructions would be allowed to execute in order without preemption.
- This is often the approach taken by non preemptive kernel.

Synchronization Hardware (2/3)

Disable Interrupt

- a user-level program is given the ability to disable interrupts, then it can disable the timer interrupt and prevent context switching from taking place, thereby allowing it to use the processor without letting other processes execute.

- Unfortunately, this solution is not as feasible in a multiprocessors environment, because disabling interrupts on a multiprocessors can be time consuming as the message is passed to all the processors.

Synchronization Hardware (3/3)

Disable Interrupt

why not disable interrupts?

- This message passing delays entry into each critical-section and the system efficiency decreases.
- Furthermore, disabling interrupts affect only the CPU that executed the disable instruction, where the other processor will continue running and can access the shared memory.

02

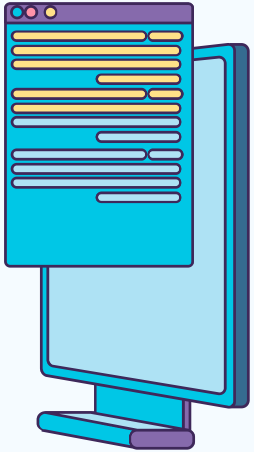
Locks (Spin Locks)

A synchronization primitive that enforces limits on access to a resource when there are many threads of executions.

Solution to Critical-section Problem Using Locks

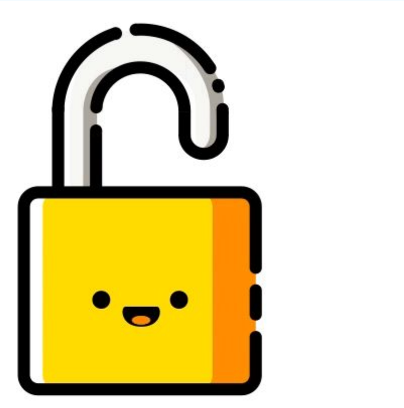
Many systems provide hardware support for critical section code:

```
do {
    acquire lock
    critical section
    release lock
    remainder section
} while (TRUE);
```



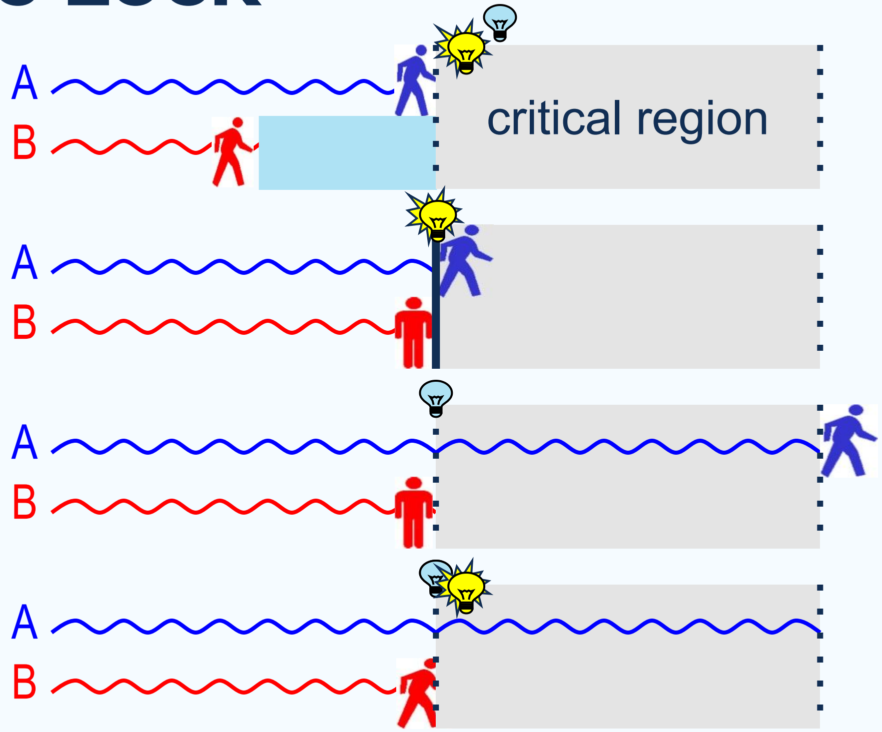
Lock

- A single shared variable (lock) initially 0.
- When the process wants to enter its critical-section, it first tests the lock. If the lock is 0, the process set it to 1 and enter the critical-section.
- If the lock is already 1, the process just wait until it becomes 0.
- Disadvantages: Some threads/processes have to wait until a lock is released. If one of the threads holding a lock dies, stalls, blocks, or enters an infinite loop, other threads waiting for the lock may wait forever.



Atomic Lock

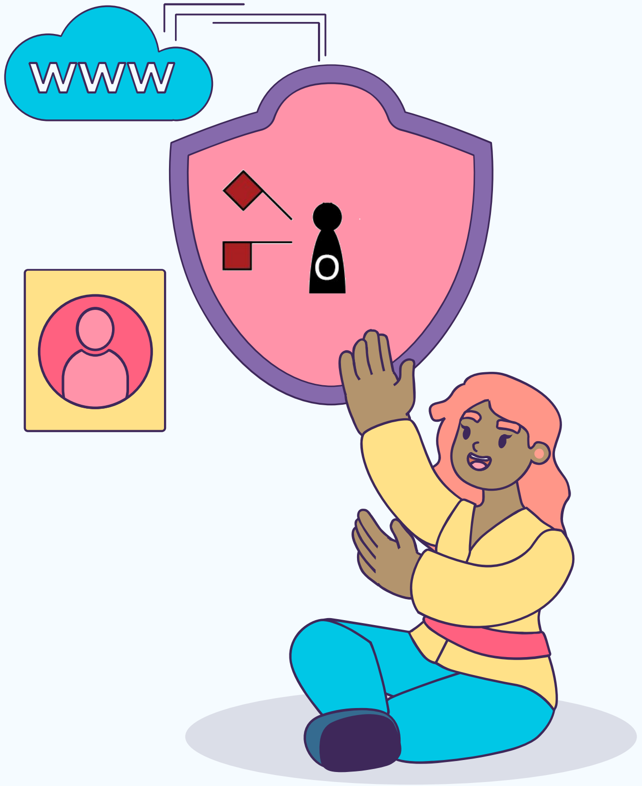
1. thread A reaches CR and <u>finds the lock at 0 *and* sets it in one shot</u>, then enters
2. even if B comes right behind A, it will find that the lock is already at 1
3. thread A exits CR, then resets lock to 0
4. thread B finds the lock at 0 and sets it to 1 in one shot, just before entering CR



03

Semaphore

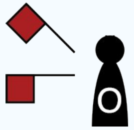
A variable or abstract data type used to control access to a common resource by multiple threads and avoid critical section problems.



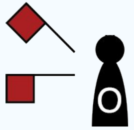
Semaphore

- Semaphore *S* – integer variable
- Two standard operations modify S: wait() and signal()
  - Originally called P() and V()
- Less complicated
- Can only be accessed via two indivisible (atomic) operations

```
wait (S) {
    while S ≤ 0;
     // no-op
    S--;
}
```

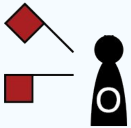


```
signal (S) {
    S++;
}
```

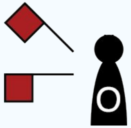


Semaphore as General Synchronization Tool

- Provide mutual exclusion



```
Semaphore S; //initialized to 1

wait (S);
Critical Section
signal (S);
```



Semaphore with two individual atomic operation

S 1

```
int balance, S;
S = 1;
// Process A
...
wait(S);
balance = balance - 200;
signal(S);
remainder sessions
```

```
int balance, S;
S = 1;
// Process B
...
wait(S);
balance = balance - 100;
signal(S);
remainder sessions
```

```
wait (S) {
    while S ≤ 0 ; // no-op
    S--;
}
```

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

Semaphore with two individual atomic operation

S 0

```
int balance, S;
S = 1;
// Process A
...
wait(S);
balance = balance - 200;
signal(S);
remainder sessions
```

```
int balance, S;
S = 1;
// Process B
...
wait(S);
balance = balance - 100;
signal(S);
remainder sessions
```

```
wait (S) {
    while S ≤ 0 ; // no-op
    S--;
}
```

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

Semaphore with two individual atomic operation

S 0

```
int balance, S;
S = 1;
// Process A
...
wait(S);
balance = balance - 200;
signal(S);
remainder sessions
```

```
int balance, S;
S = 1;
// Process B
...
wait(S);
balance = balance - 100;
signal(S);
remainder sessions
```

```
wait (S) {
    while S ≤ 0 ; // no-op
    S--;
}
```

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

Semaphore with two individual atomic operation

S 1

```
int balance, S;
S = 1;
// Process A
...
wait(S);
balance = balance - 200;
signal(S);
remainder sessions
```

```
int balance, S;
S = 1;
// Process B
...
wait(S);
balance = balance - 100;
signal(S);
remainder sessions
```

```
wait (S) {
    while S ≤ 0 ; // no-op
    S--;
}
```

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

Semaphore with two individual atomic operation

S 1

```
int balance, S;
S = 1;
// Process A
...
wait(S);
balance = balance - 200;
signal(S);
remainder sessions
```

```
int balance, S;
S = 1;
// Process B
...
wait(S);
balance = balance - 100;
signal(S);
remainder sessions
```

```
wait (S) {
    while S ≤ 0 ; // no-op
    S--;
}
```

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

Semaphore with two individual atomic operation

S 0

```
int balance, S;
S = 1;
// Process A
...
wait(S);
balance = balance - 200;
signal(S);
remainder sessions
```

```
int balance, S;
S = 1;
// Process B
...
wait(S);
balance = balance - 100;
signal(S);
remainder sessions
```

```
wait (S) {
    while S ≤ 0 ; // no-op
    S--;
}
```

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

Semaphore with two individual atomic operation

S 0

```
int balance, S;
S = 1;
// Process A
...
wait(S);
balance = balance - 200;
signal(S);
remainder sessions
```

```
int balance, S;
S = 1;
// Process B
...
wait(S);
balance = balance - 100;
signal(S);
remainder sessions
```

```
wait (S) {
    while S ≤ 0 ; // no-op
    S--;
}
```

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

Semaphore with two individual atomic operation

S 1

```
int balance, S;
S = 1;
// Process A
...
wait(S);
balance = balance - 200;
signal(S);
remainder sessions
```

```
int balance, S;
S = 1;
// Process B
...
wait(S);
balance = balance - 100;
signal(S);
remainder sessions
```

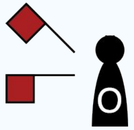
```
wait (S) {
    while S ≤ 0 ; // no-op
    S--;
}
```

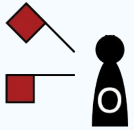
```
signal (S) {
    S++;
}
```

Semaphore Implementation

- Must guarantee that no two processes can execute wait () and signal () on the same semaphore at the same time

- Thus, implementation becomes the critical section problem where the wait and signal code are placed in the critical section.
    - Could now have busy waiting in critical section implementation
        - But implementation code is short
        - Little busy waiting if critical section rarely occupied





- Note that applications may spend lots of time in critical sections and therefore this is not a good solution.

Semaphore Bounded-Buffer

N buffers, each can hold one item

Semaphore Mutex	Semaphore Full	Semaphore Empty
Initialise to the value 1	Initialise to the value 0	Initialise to the value N

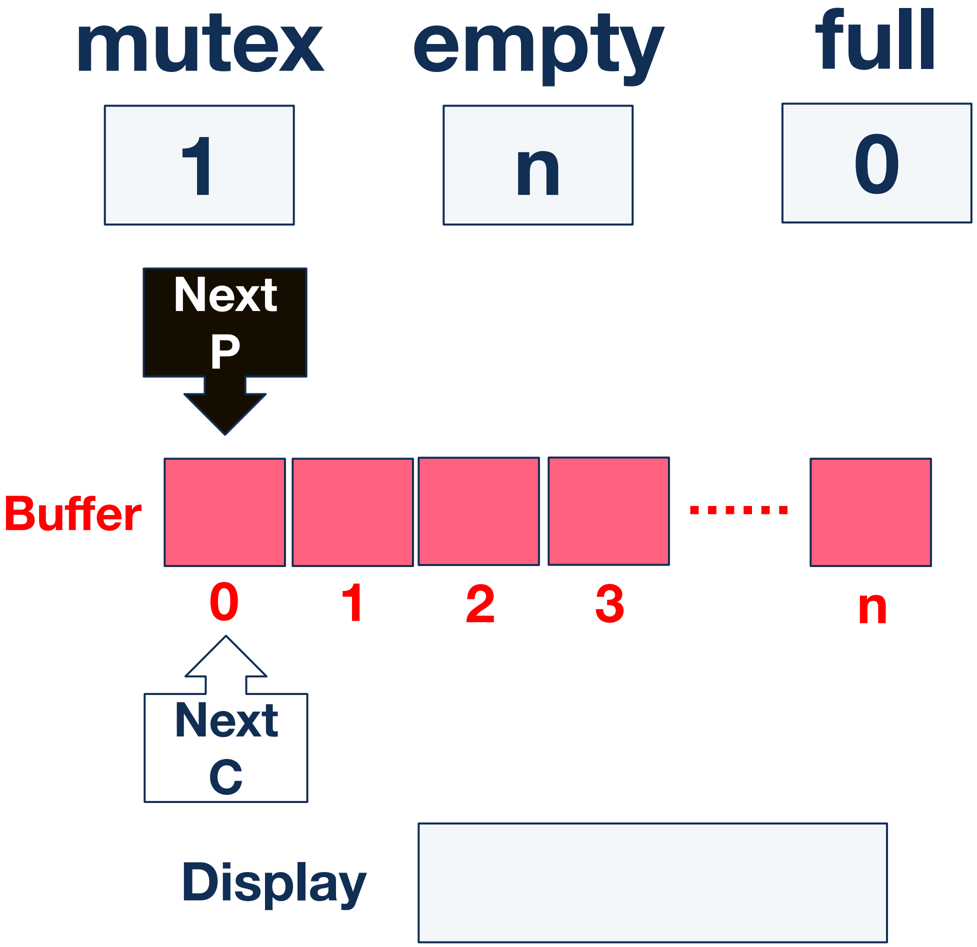
Semaphore in Producer-Consumer

Structure Process of Producer

```
do {
     //   produce an item
     wait (empty);
     wait (mutex);

     //  add item to buffer

     signal (mutex);
     signal (full);
} while (true);
```

Structure Process of Consumer

```
do {
     wait (full);
     wait (mutex);
     //  remove item from  buffer

     signal (mutex);
     signal (empty);

     // consume the removed item
} while (true);
```

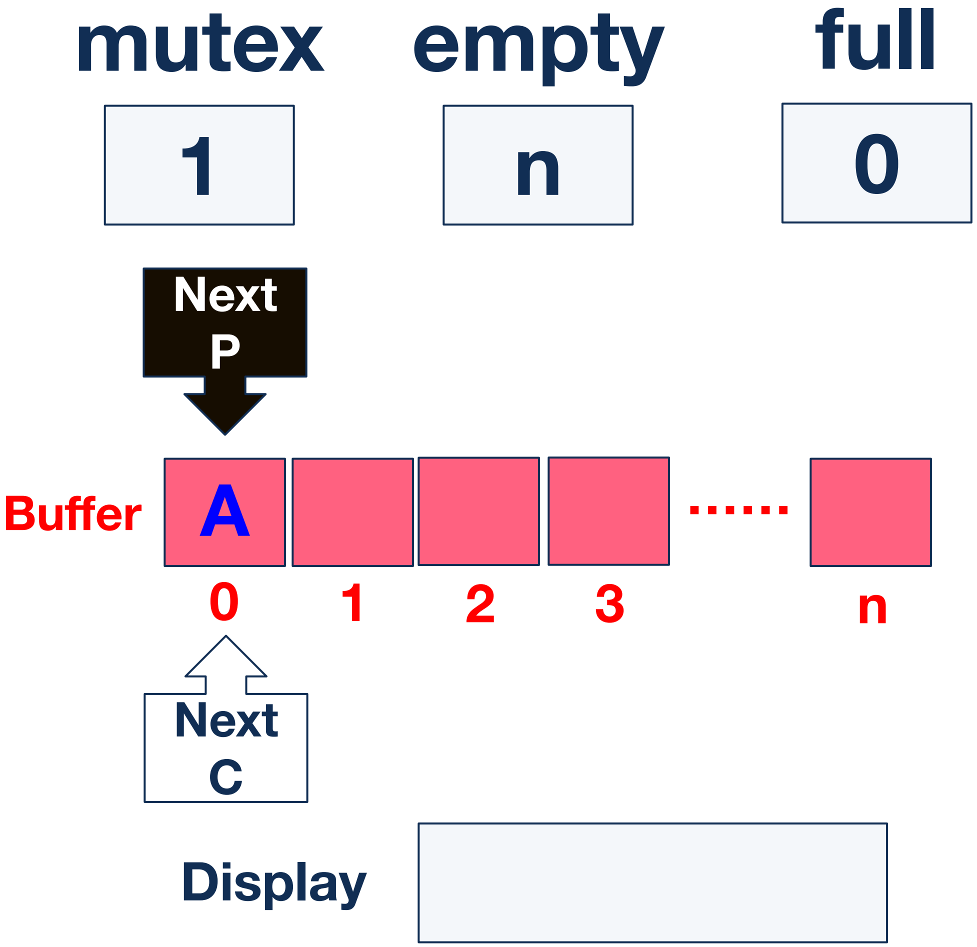


Producer

```
do{
    //produce an item in nextp
    Wait(empty);
    Wait(mutex);
    //add nextp to buffer
    signal(Mutex);
    signal(full);
}(true)
```

Consumer

```
do{
    Wait(full);
    Wait(mutex);
    //remove item from buffer to nextc
    Signal(mutex);
    Signal(empty);
    //consume the item in nextc
}(true)
```

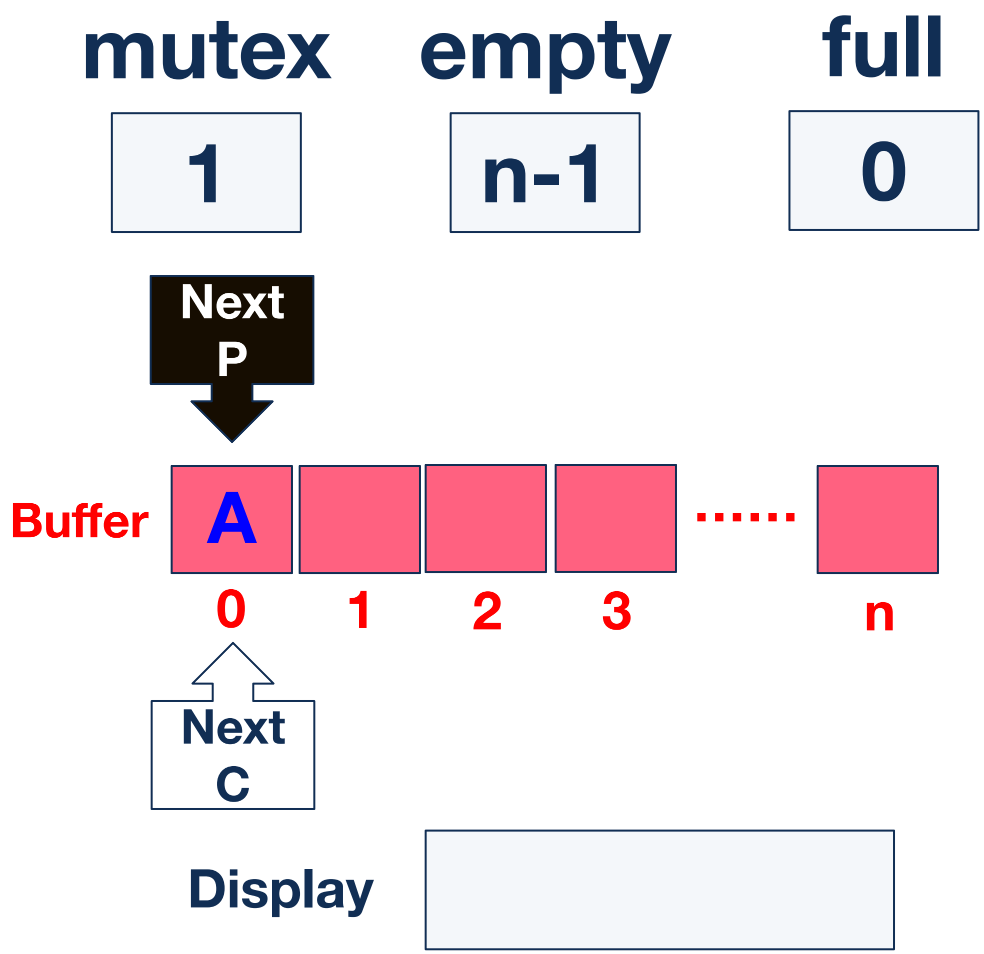


Producer

```
do{
    //produce an item in nextp
    Wait(empty);
    Wait(mutex);
    //add nextp to buffer
    signal(Mutex);
    signal(full);
}(true)
```

Consumer

```
do{
    wait(full);
    wait(mutex);
    //remove item from buffer to nextc
    signal(mutex);
    signal(empty);
    //consume the item in nextc
}(true)
```

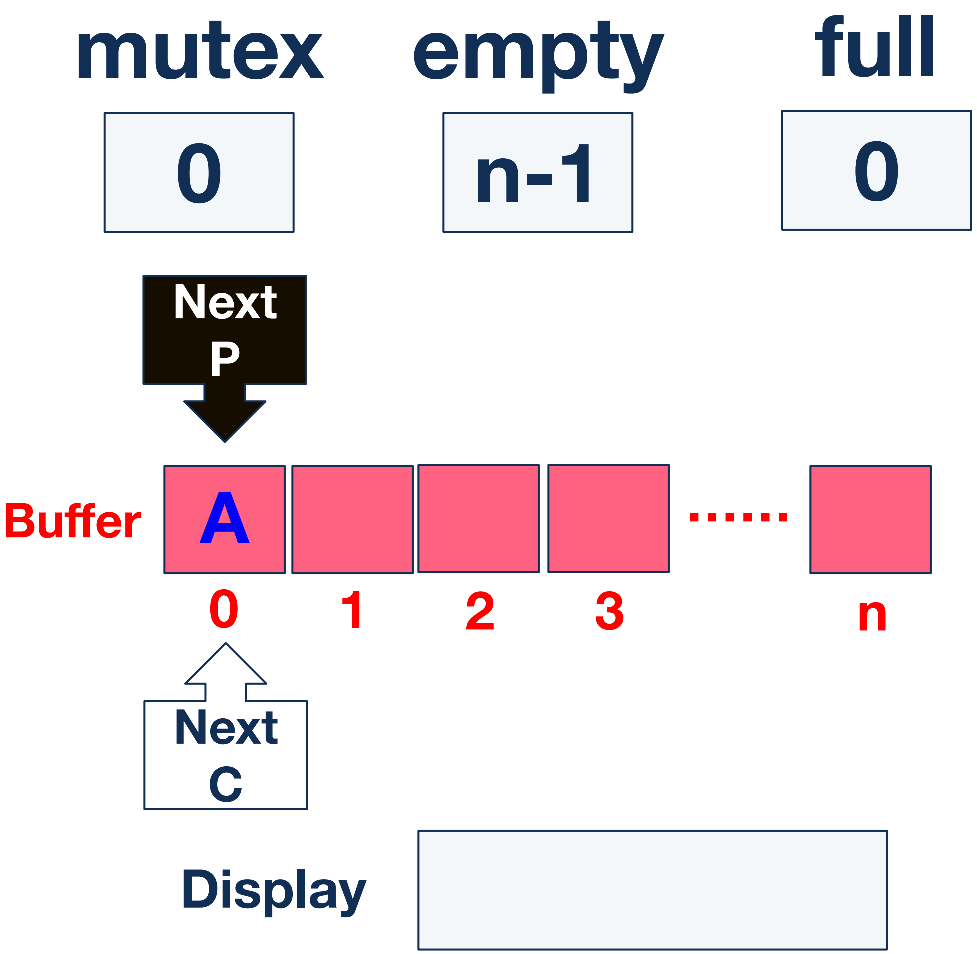


Producer

```
do{
    //produce an item in nextp
    Wait(empty);
    Wait(mutex);
    //add nextp to buffer
    signal(Mutex);
    signal(full);
}(true)
```

Consumer

```
do{
    wait(full);
    wait(mutex);
    //remove item from buffer to nextc
    signal(mutex);
    signal(empty);
    //consume the item in nextc
}(true)
```

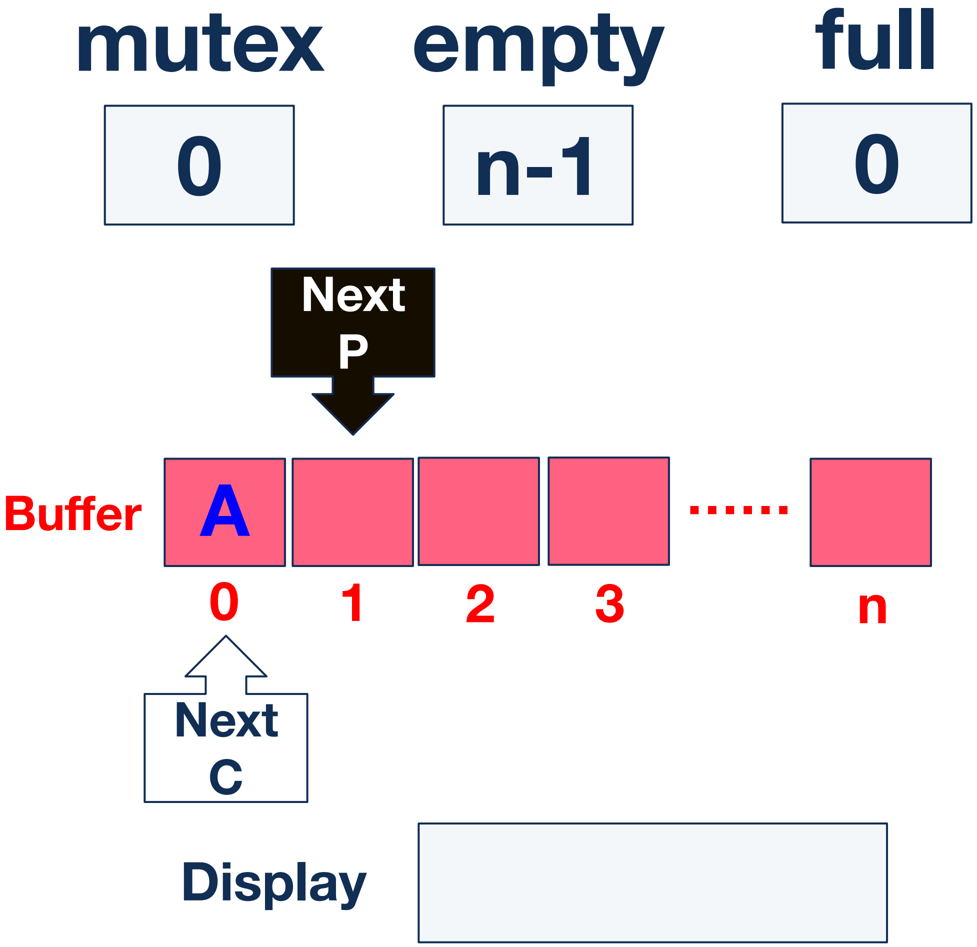


Producer

```
do{
     //produce an item in nextp
     Wait(empty);
     Wait(mutex);
     //add nextp to buffer
     signal(Mutex);
     signal(full);
}(true)
```

Consumer

```
do{
     wait(full);
     wait(mutex);
     //remove item from buffer to nextc
     signal(mutex);
     signal(empty);
     //consume the item in nextc
}(true)
```

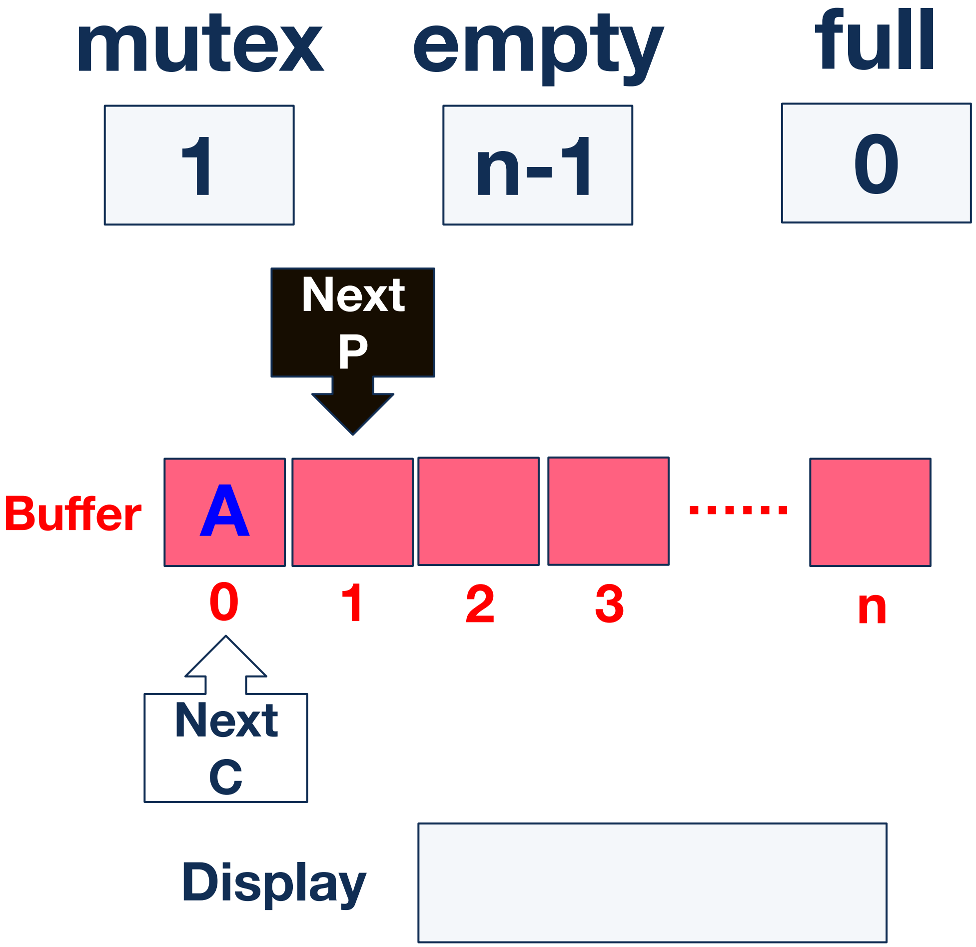


Producer

```
do{
    //produce an item in nextp
    Wait(empty);
    Wait(mutex);
    //add nextp to buffer
    signal(Mutex);
    signal(full);
}(true)
```

Consumer

```
do{
    wait(full);
    wait(mutex);
    //remove item from buffer to nextc
    signal(mutex);
    signal(empty);
    //consume the item in nextc
}(true)
```

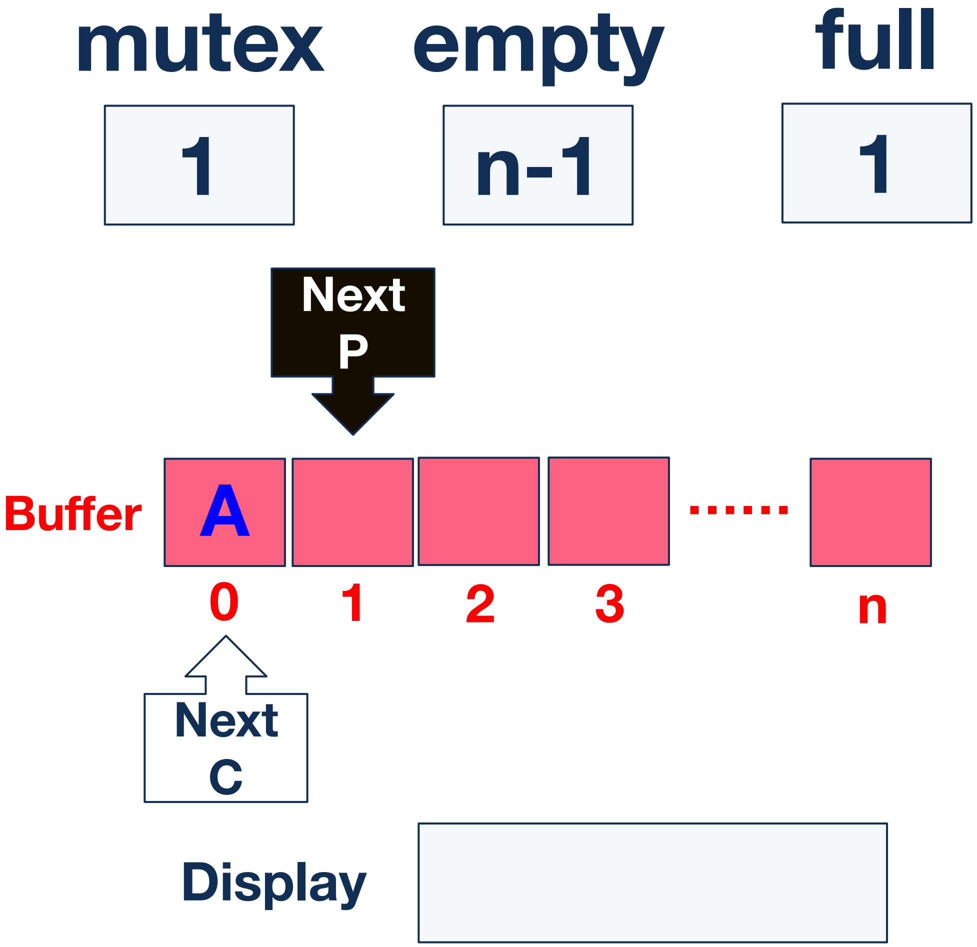


Producer

```
do{
    //produce an item in nextp
    Wait(empty);
    Wait(mutex);
    //add nextp to buffer
    signal(Mutex);
    signal(full);
}(true)
```

Consumer

```
do{
    wait(full);
    wait(mutex);
    //remove item from buffer to nextc
    signal(mutex);
    signal(empty);
    //consume the item in nextc
}(true)
```

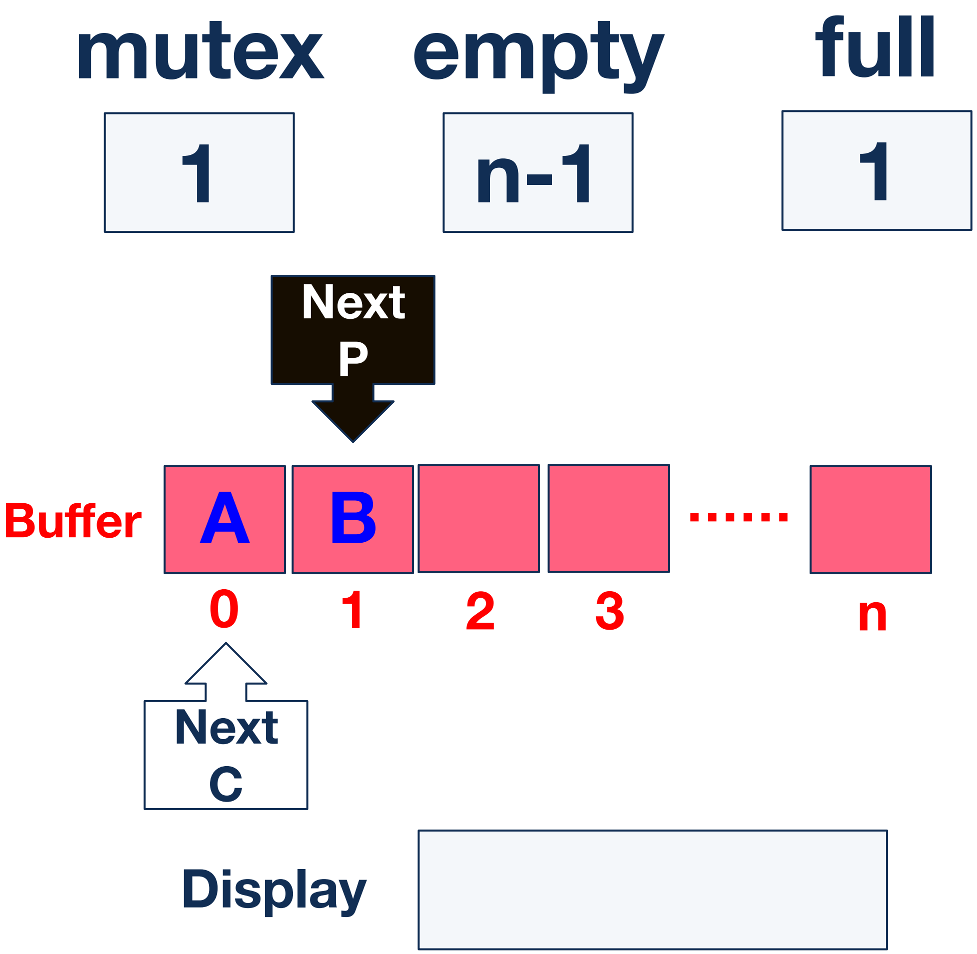


Producer

```
do{
      //produce an item in nextp
      wait(empty);
      wait(mutex);
      //add nextp to buffer
      signal(Mutex);
      signal(full);
}(true)
```

Consumer

```
do{
      wait(full);
      wait(mutex);
      //remove item from buffer to nextc
      signal(mutex);
      signal(empty);
      //consume the item in nextc
}(true)
```

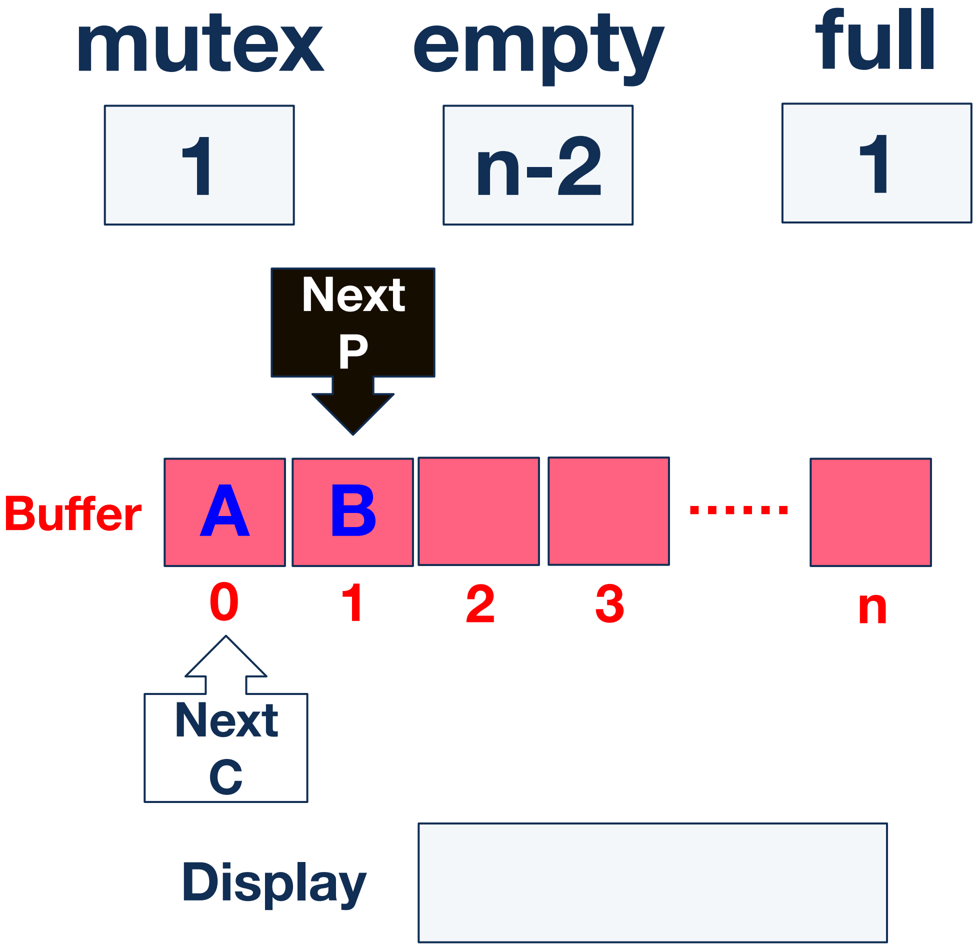


Producer

```
do{
    //produce an item in nextp
    wait(empty);
    wait(mutex);
    //add nextp to buffer
    signal(Mutex);
    signal(full);
}(true)
```

Consumer

```
do{
    wait(full);
    wait(mutex);
    //remove item from buffer to nextc
    signal(mutex);
    signal(empty);
    //consume the item in nextc
}(true)
```

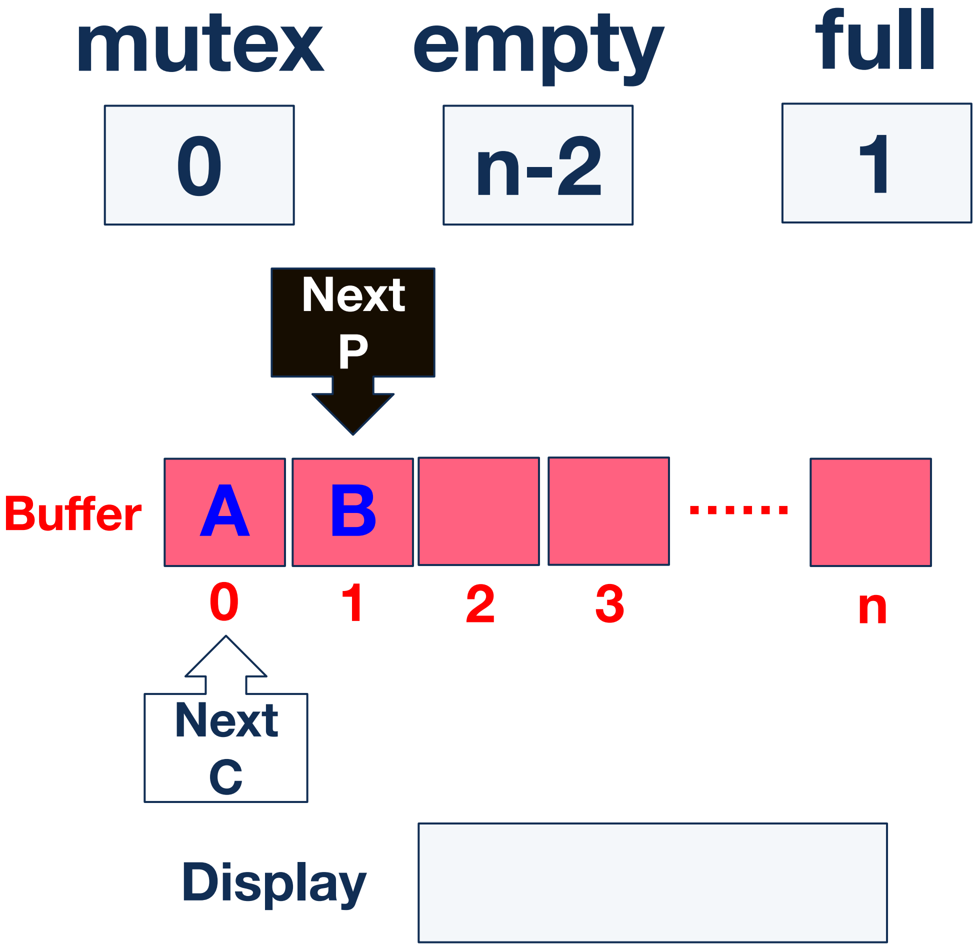


Producer

```
do{
    //produce an item in nextp
    wait(empty);
    wait(mutex);
    //add nextp to buffer
    signal(Mutex);
    signal(full);
}(true)
```

Consumer

```
do{
    wait(full);
    wait(mutex);
    //remove item from buffer to nextc
    signal(mutex);
    signal(empty);
    //consume the item in nextc
}(true)
```

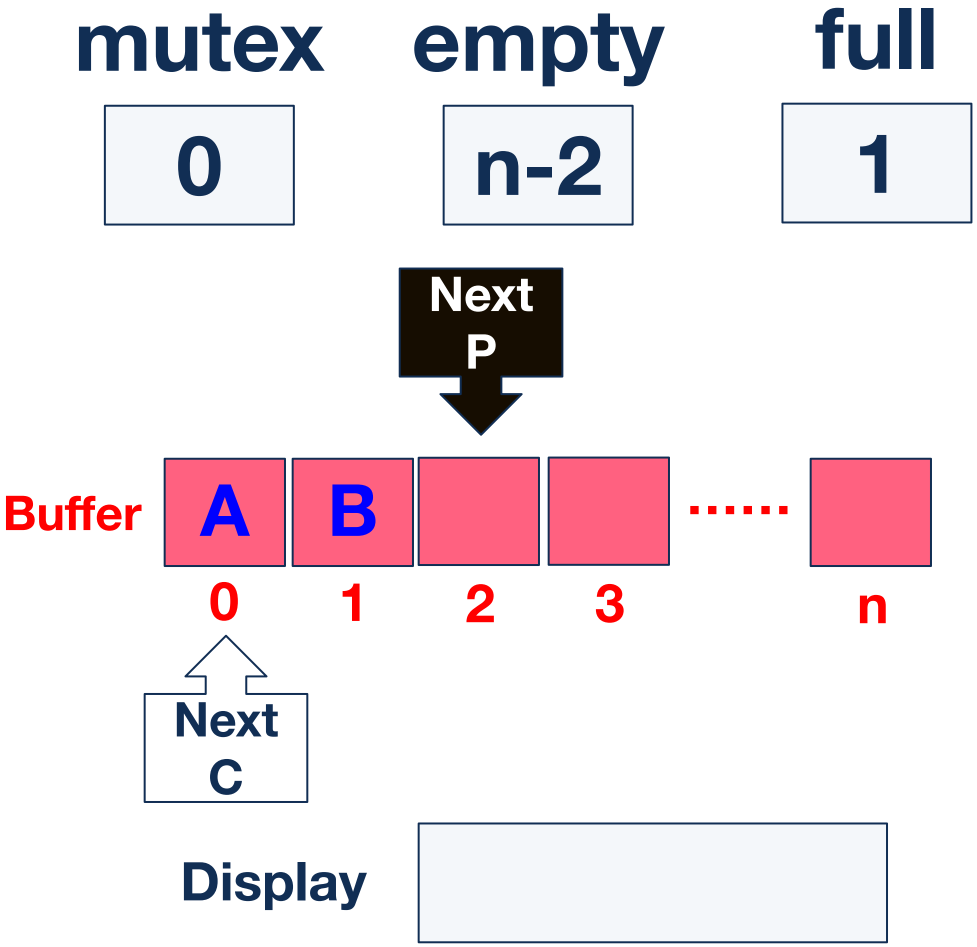


Producer

```
do{
      //produce an item in nextp
      wait(empty);
      wait(mutex);
      //add nextp to buffer
      signal(Mutex);
      signal(full);
}(true)
```

Consumer

```
do{
      wait(full);
      wait(mutex);
      //remove item from buffer to nextc
      signal(mutex);
      signal(empty);
      //consume the item in nextc
}(true)
```

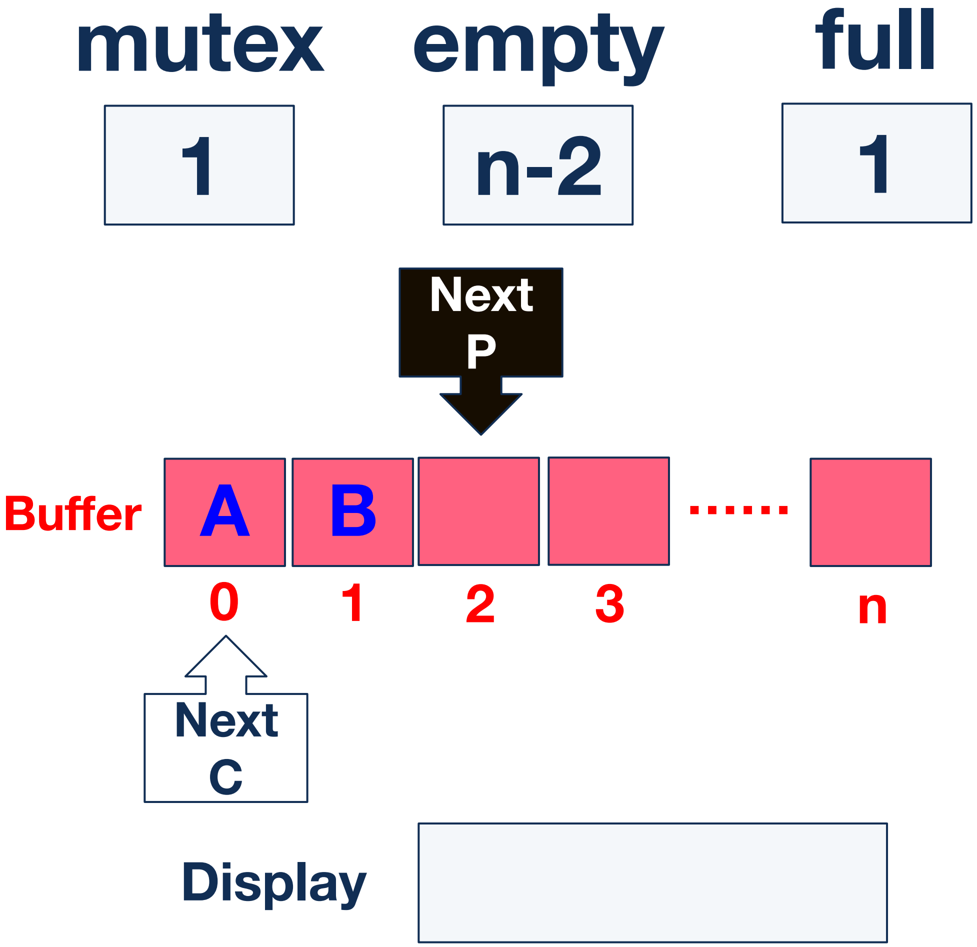


Producer

```
do{
    //produce an item in nextp
    wait(empty);
    wait(mutex);
    //add nextp to buffer
    signal(Mutex);
    signal(full);
}(true)
```

Consumer

```
do{
    wait(full);
    wait(mutex);
    //remove item from buffer to nextc
    signal(mutex);
    signal(empty);
    //consume the item in nextc
}(true)
```

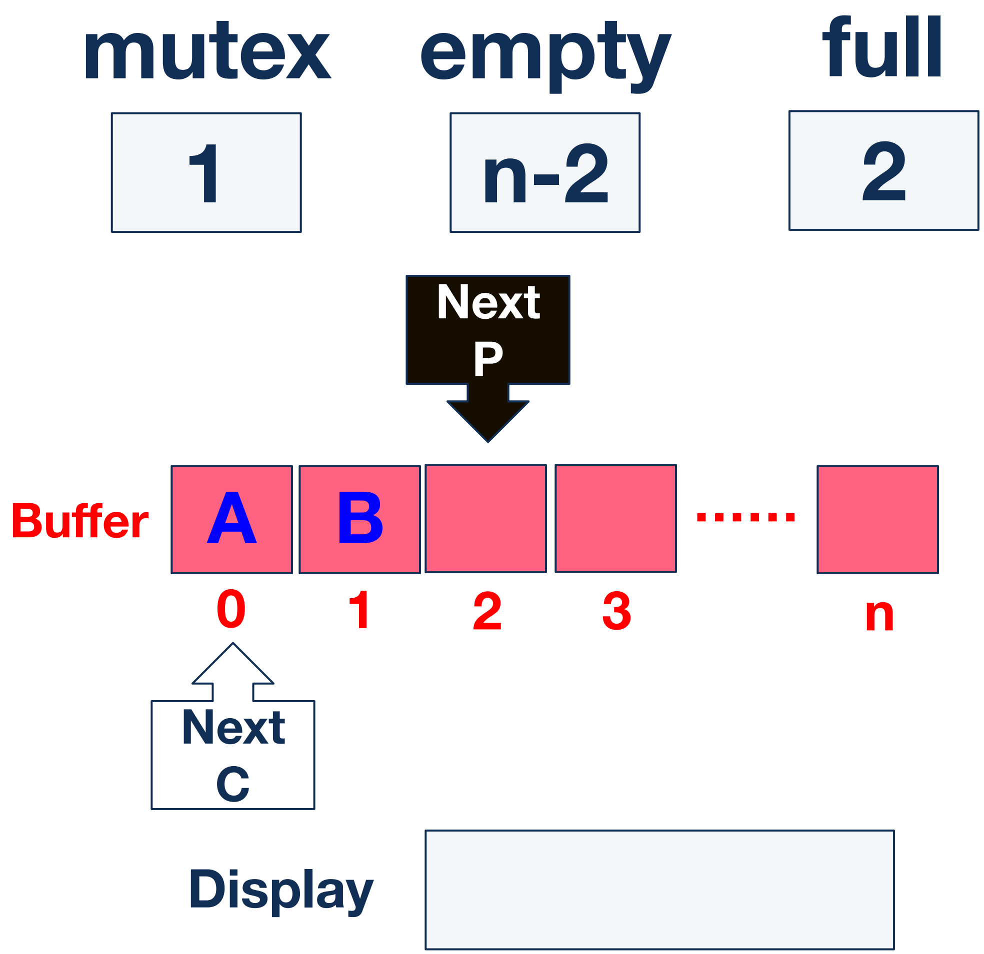


Producer

```
do{
     //produce an item in nextp
     wait(empty);
     wait(mutex);
     //add nextp to buffer
     signal(Mutex);
     signal(full);
}(true)
```

Consumer

```
do{
     wait(full);
     wait(mutex);
     //remove item from buffer to nextc
     signal(mutex);
     signal(empty);
     //consume the item in nextc
}(true)
```

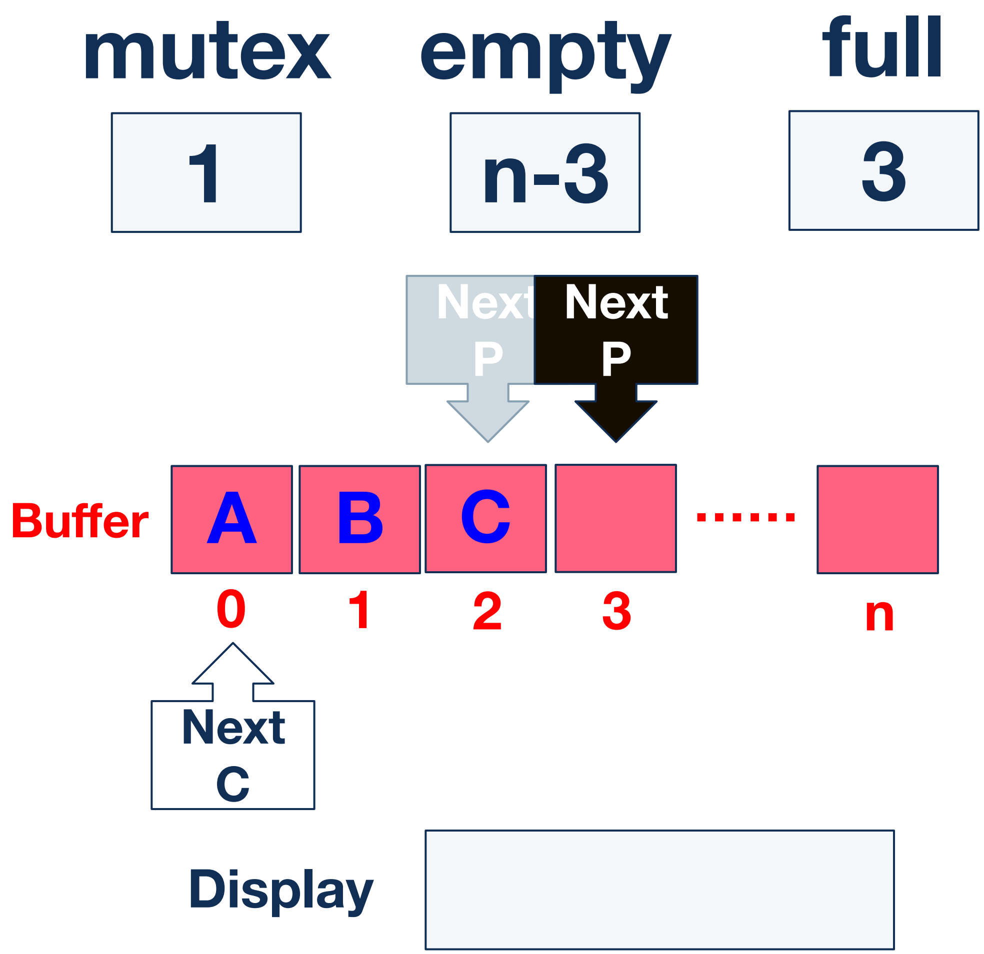


Producer

```
do{
    //produce an item in nextp
    wait(empty);
    wait(mutex);
    //add nextp to buffer
    signal(Mutex);
    signal(full);
}(true)
```

Consumer

```
do{
    wait(full);
    wait(mutex);
    //remove item from buffer to nextc
    signal(mutex);
    signal(empty);
    //consume the item in nextc
}(true)
```

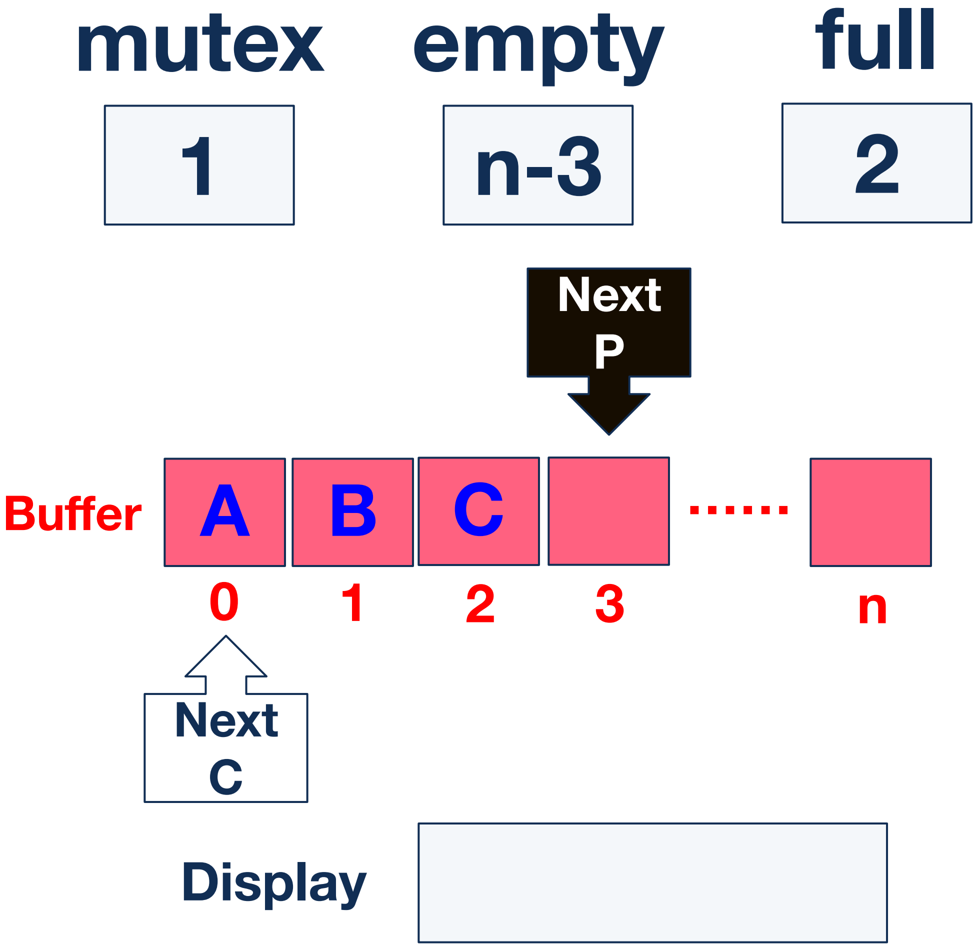


Producer

```
do{
    //produce an item in nextp
    wait(empty);
    wait(mutex);
    //add nextp to buffer
    signal(Mutex);
    signal(full);
}(true)
```

Consumer

```
do{
    wait(full);
    wait(mutex);
    //remove item from buffer to nextc
    signal(mutex);
    signal(empty);
    //consume the item in nextc
}(true)
```

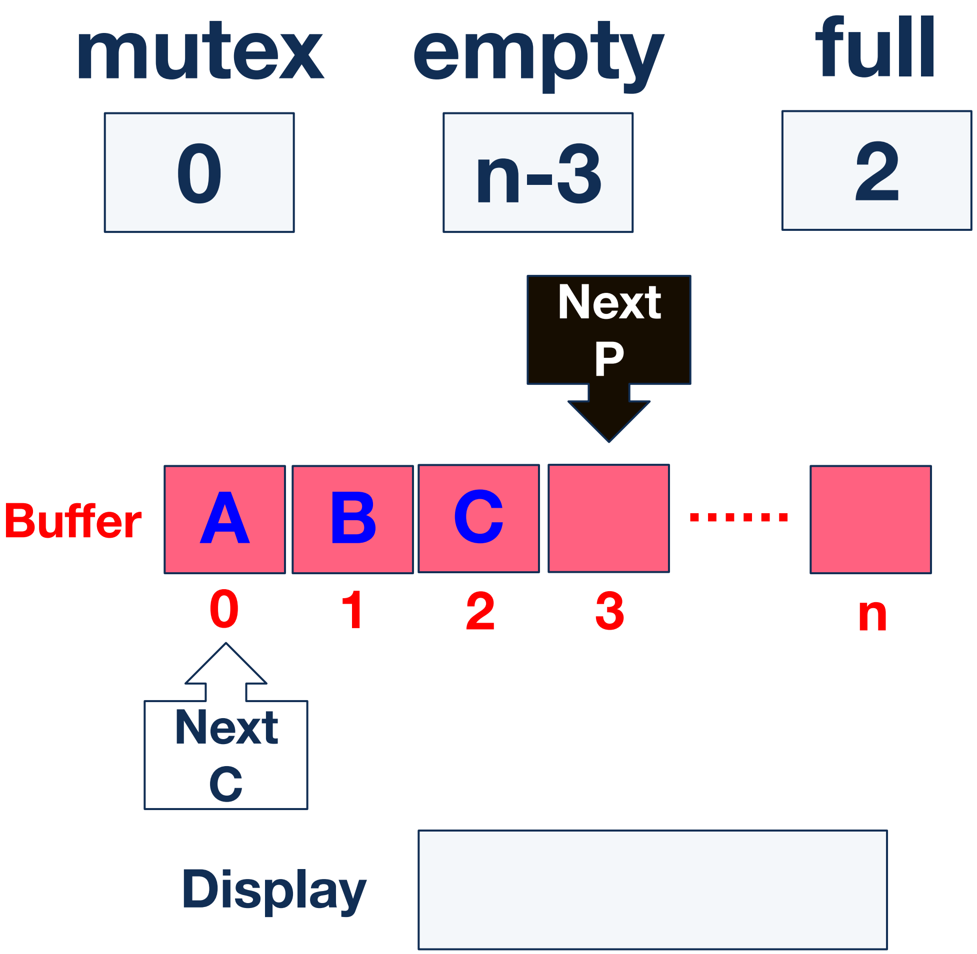


Producer

```
do{
     //produce an item in nextp
     wait(empty);
     wait(mutex);
     //add nextp to buffer
     signal(Mutex);
     signal(full);
}(true)
```

Consumer

```
do{
     Wait(full);
     Wait(mutex);
     //remove item from buffer to nextc
     Signal(mutex);
     Signal(empty);
     //consume the item in nextc
}(true)
```

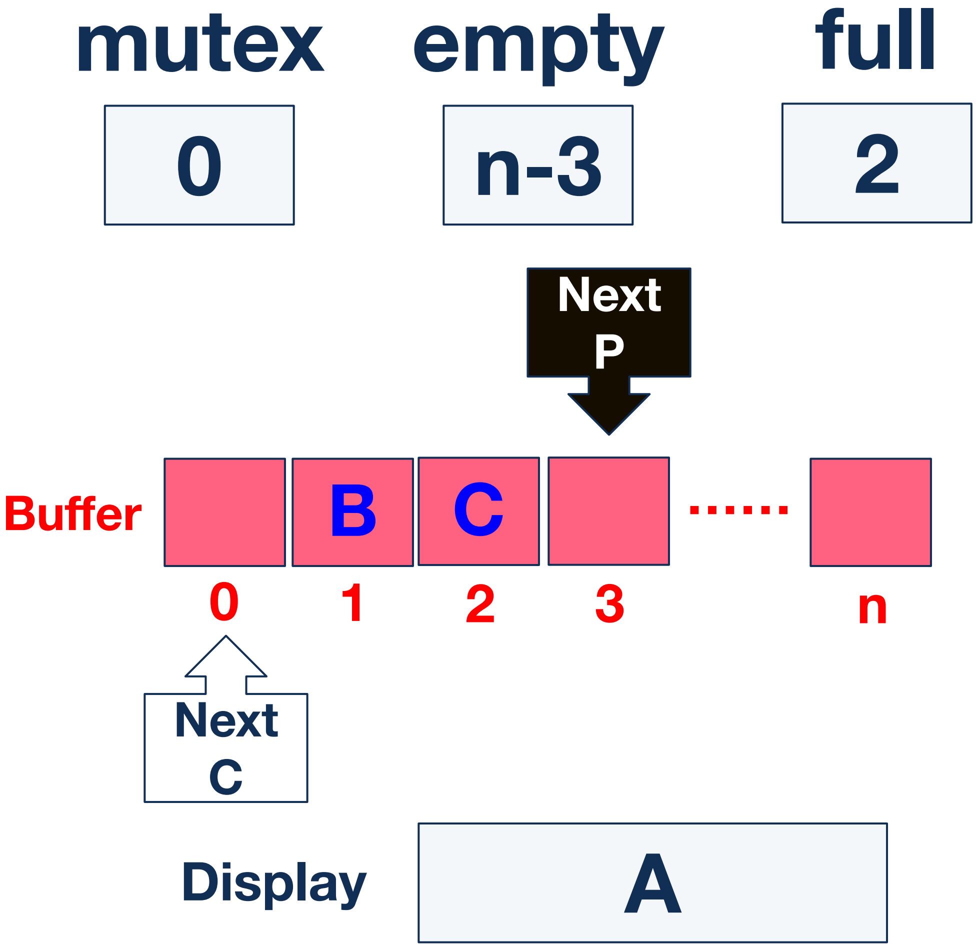


Producer

```
do{
      //produce an item in nextp
      wait(empty);
      wait(mutex);
      //add nextp to buffer
      signal(Mutex);
      signal(full);
}(true)
```

Consumer

```
do{
      Wait(full);
      Wait(mutex);
      //remove item from buffer to nextc
      Signal(mutex);
      Signal(empty);
      //consume the item in nextc
}(true)
```

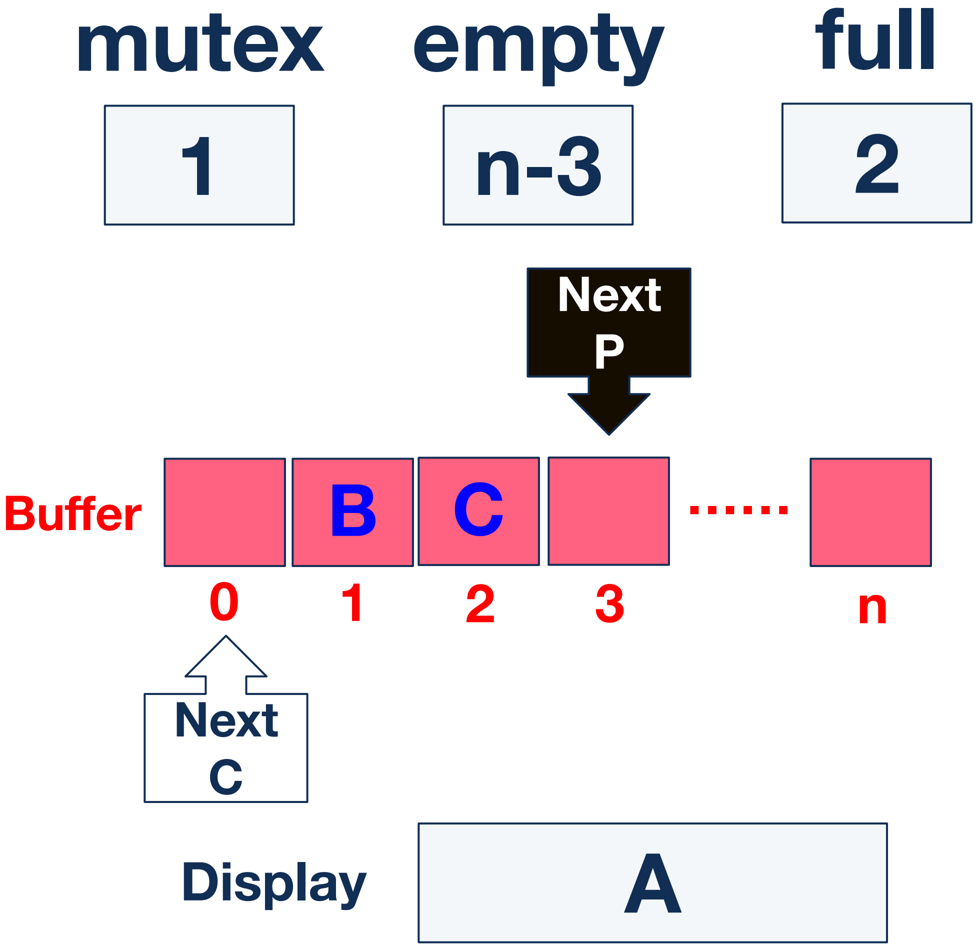


Producer

```
do{
      //produce an item in nextp
      wait(empty);
      wait(mutex);
      //add nextp to buffer
      signal(Mutex);
      signal(full);
}(true)
```

Consumer

```
do{
      Wait(full);
      Wait(mutex);
      //remove item from buffer to nextc
      Signal(mutex);
      Signal(empty);
      //consume the item in nextc
}(true)
```

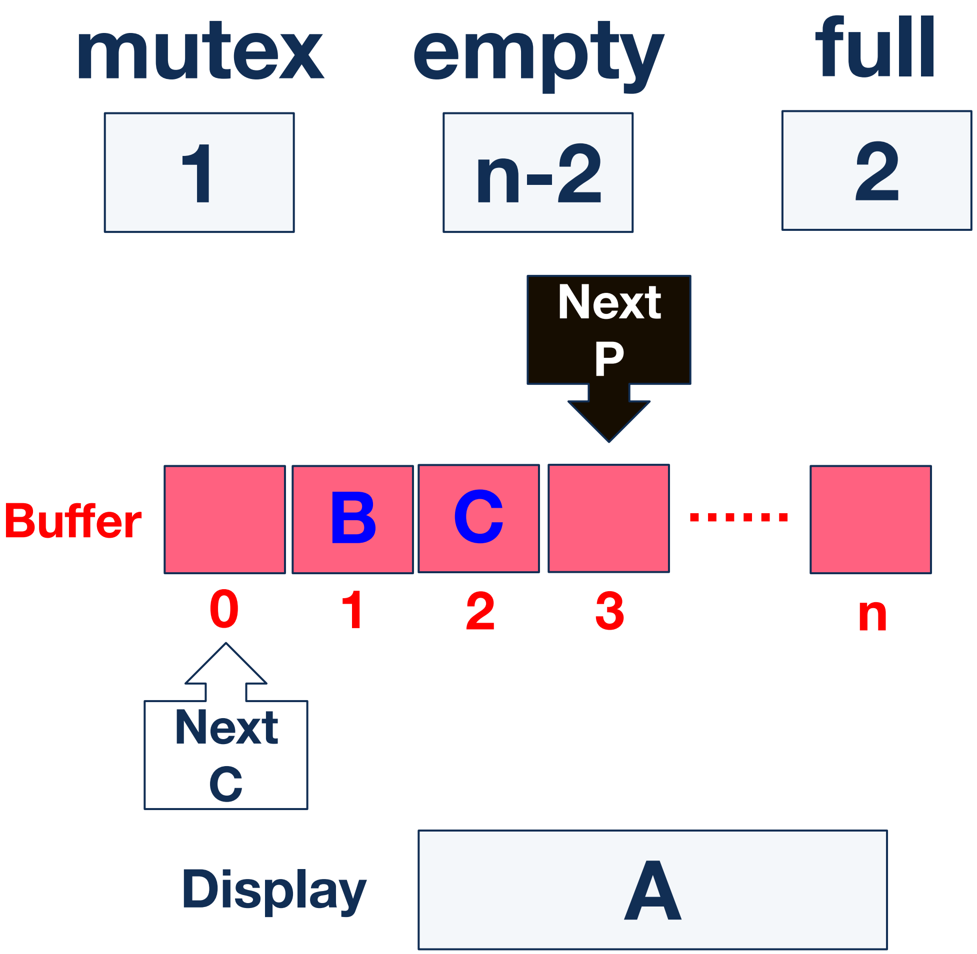


Producer

```
do{
    //produce an item in nextp
    wait(empty);
    wait(mutex);
    //add nextp to buffer
    signal(Mutex);
    signal(full);
}(true)
```

Consumer

```
do{
    Wait(full);
    Wait(mutex);
    //remove item from buffer to nextc
    Signal(mutex);
    Signal(empty);
    //consume the item in nextc
}(true)
```

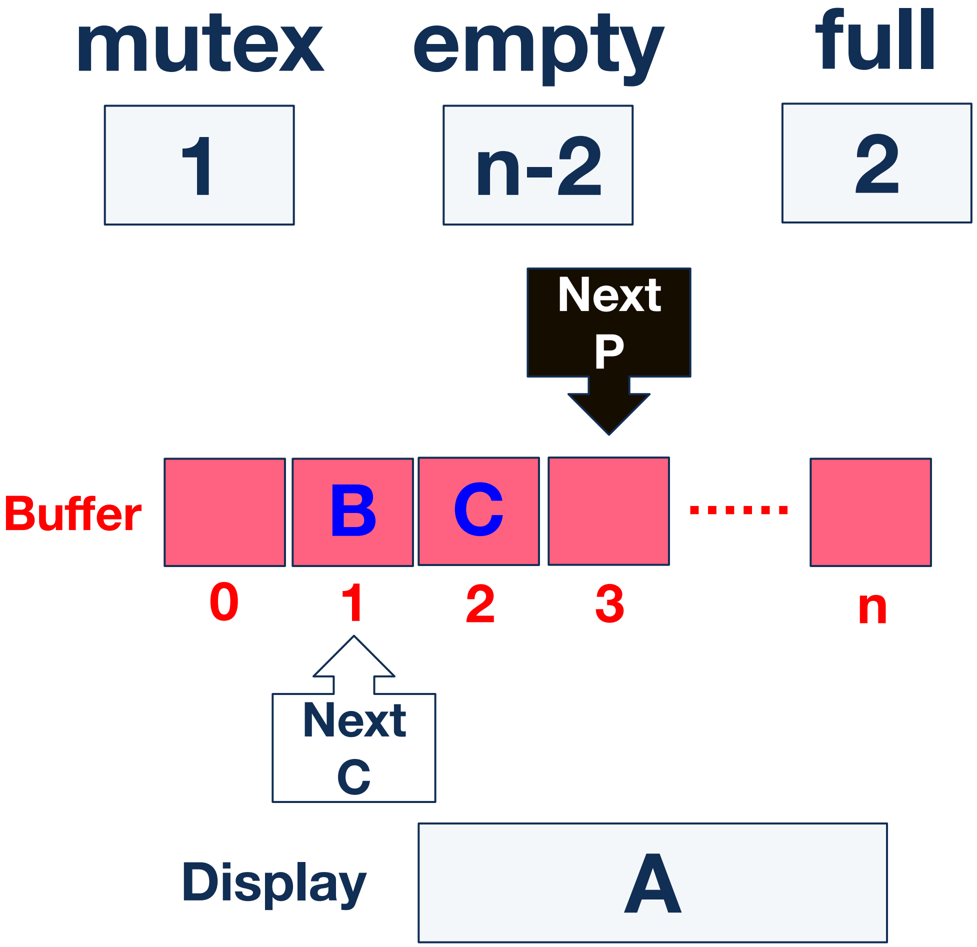


Producer

```
do{
    //produce an item in nextp
    wait(empty);
    wait(mutex);
    //add nextp to buffer
    signal(Mutex);
    signal(full);
}(true)
```

Consumer

```
do{
    Wait(full);
    Wait(mutex);
    //remove item from buffer to nextc
    Signal(mutex);
    Signal(empty);
    //consume the item in nextc
}(true)
```

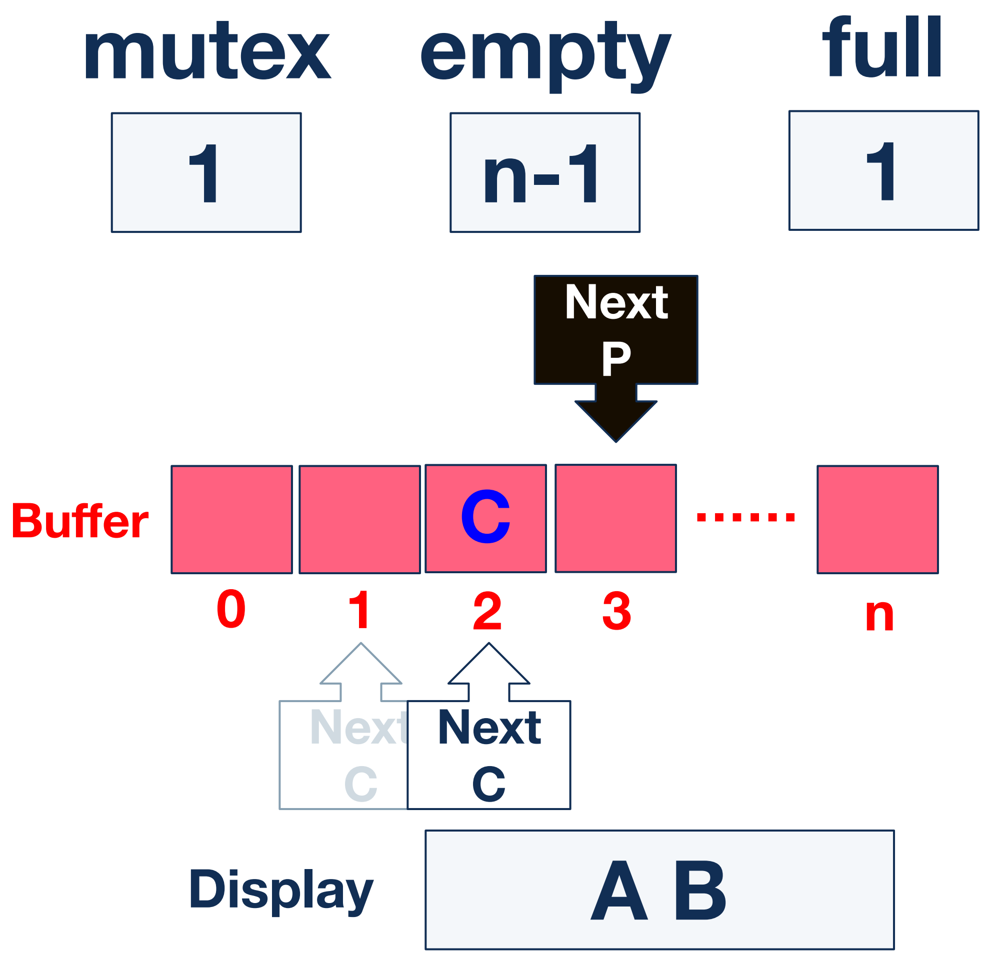


Producer

```
do{
    //produce an item in nextp
    wait(empty);
    wait(mutex);
    //add nextp to buffer
    signal(Mutex);
    signal(full);
}(true)
```

Consumer

```
do{
    Wait(full);
    Wait(mutex);
    //remove item from buffer to nextc
    Signal(mutex);
    Signal(empty);
    //consume the item in nextc
}(true)
```

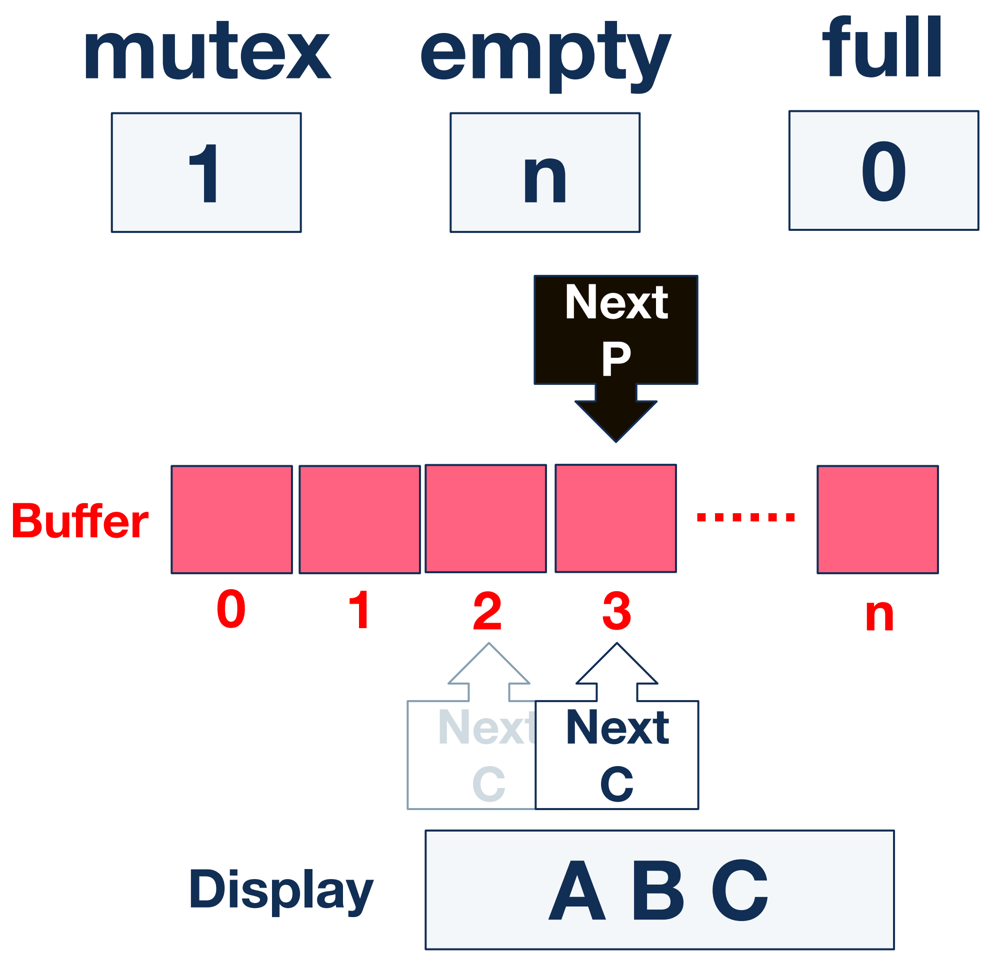


Producer

```
do{
    //produce an item in nextp
    wait(empty);
    wait(mutex);
    //add nextp to buffer
    signal(Mutex);
    signal(full);
}(true)
```

Consumer

```
do{
    Wait(full);
    Wait(mutex);
    //remove item from buffer to nextc
    Signal(mutex);
    Signal(empty);
    //consume the item in nextc
}(true)
```



Producer

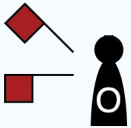
```
do{
      //produce an item in nextp
      wait(empty);
      wait(mutex);
      //add nextp to buffer
      signal(Mutex);
      signal(full);
}(true)
```

Consumer

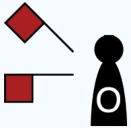
```
do{
      Wait(full);
      Wait(mutex);
      //remove item from buffer to nextc
      Signal(mutex);
      Signal(empty);
      //consume the item in nextc
}(true)
```

Semaphore Implementation
With No Busy Waiting

- With each semaphore there is an associated waiting queue. Each entry in a waiting queue has two data items:
  - value (of type integer)
  - pointer to next record in the list



- Two operations:
  - Block() – place the process invoking the operation on the appropriate waiting queue.
  - Wakeup() – remove one of processes in the waiting queue and place it in the ready queue.



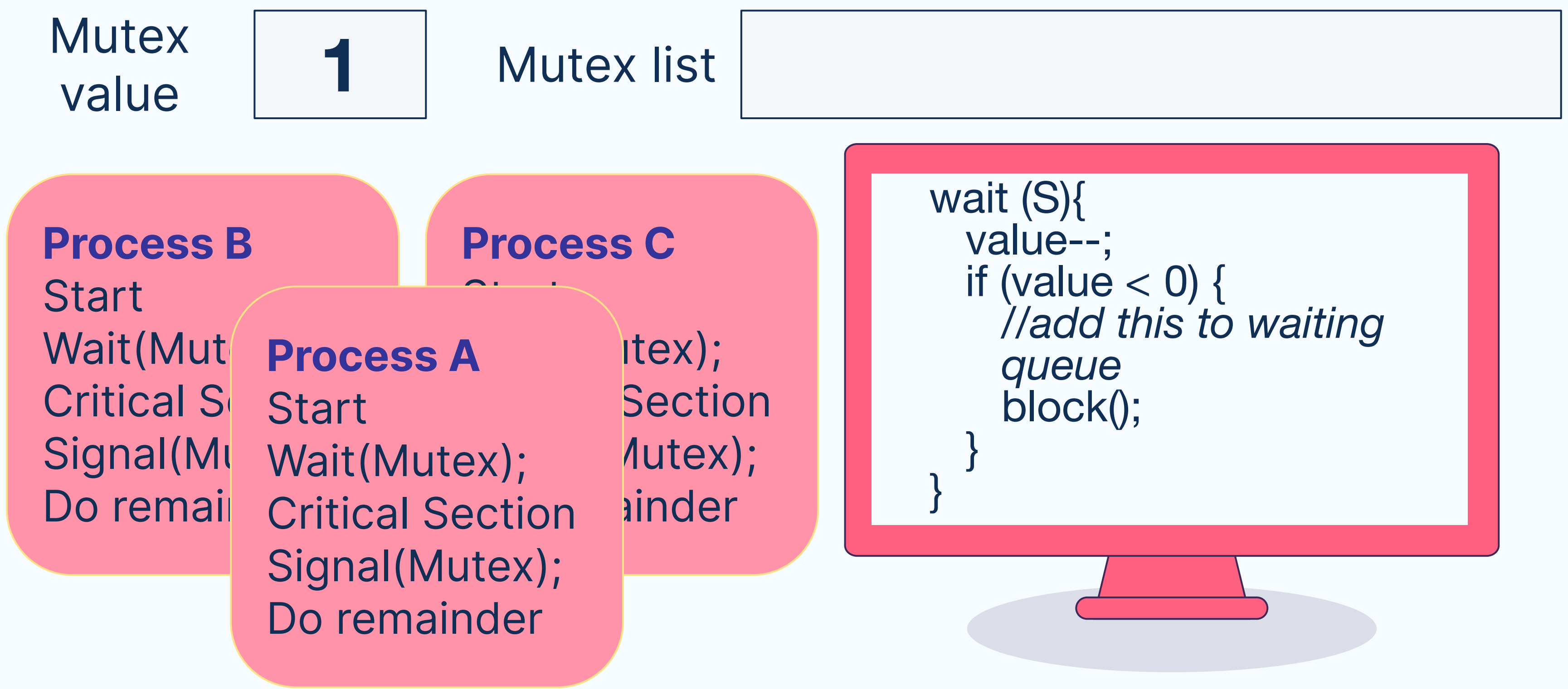
Semaphore implementation

Implementation of Wait()

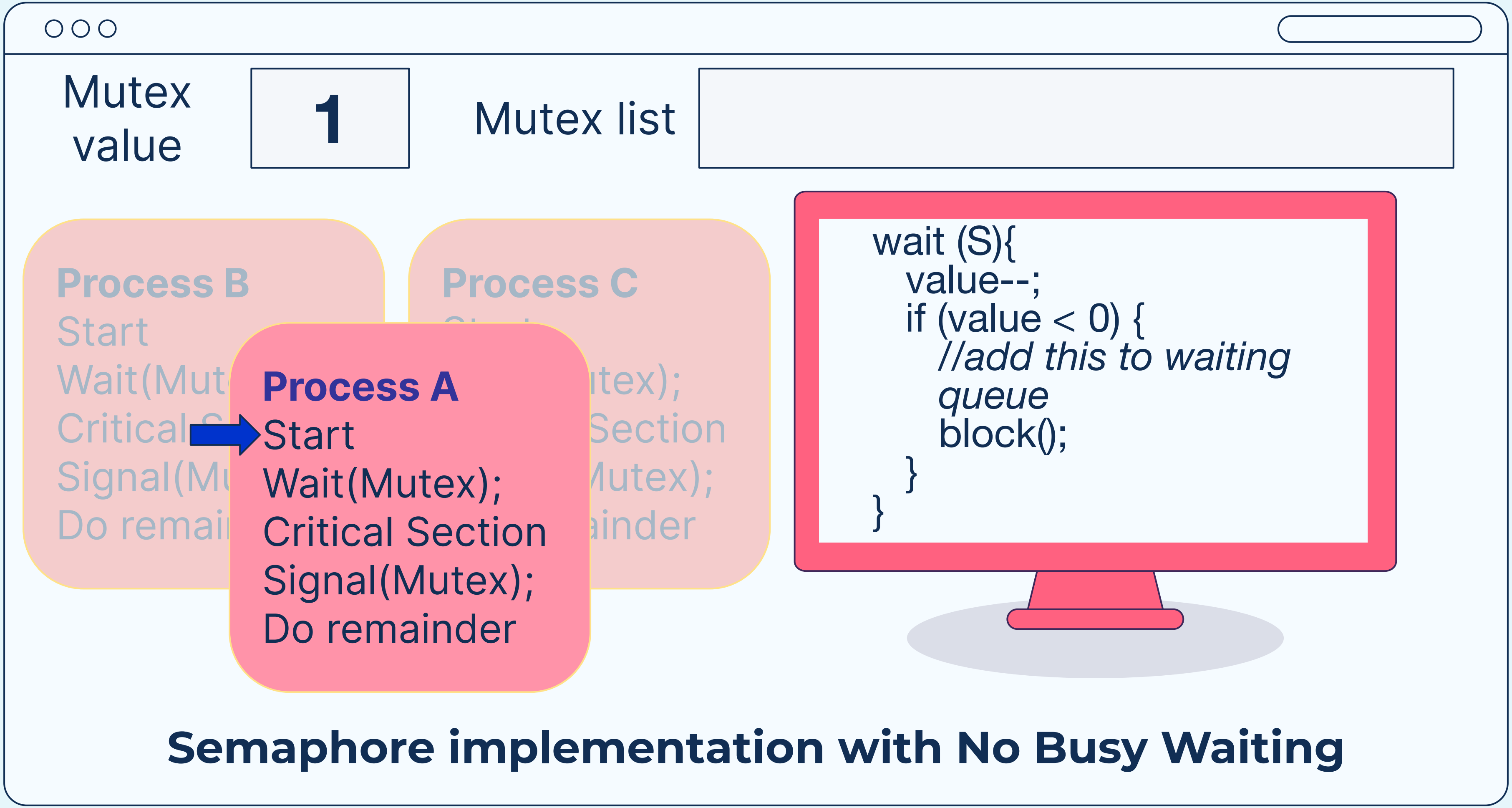
```
wait (S){
value--;

if (value < 0) {
block();}

-- add this process to
   waiting queue
}
```

Implementation of Signal()

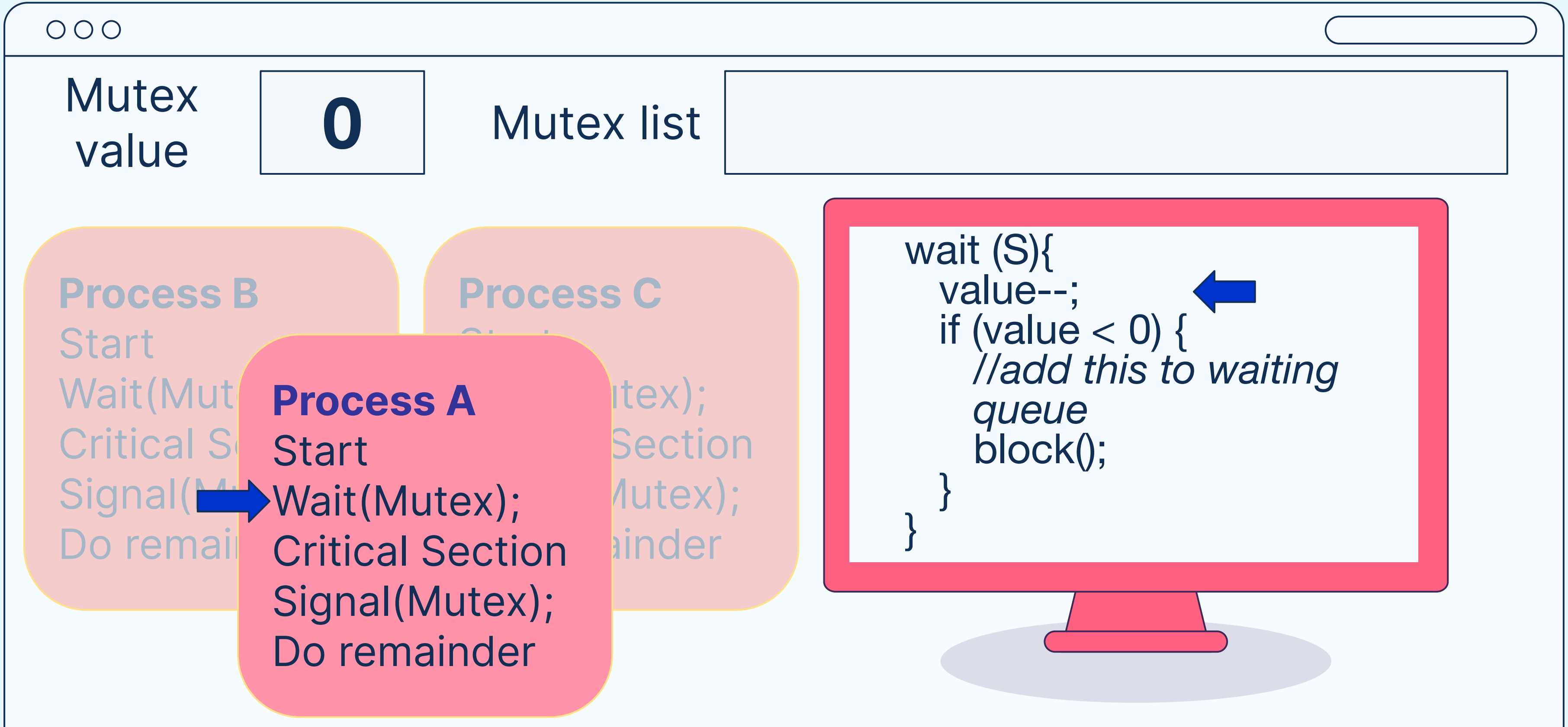
```
Signal (S){
value++;

if (value ≤ 0) {
wakeup(P);  }

-- remove a process P from
the waiting queue
}
```



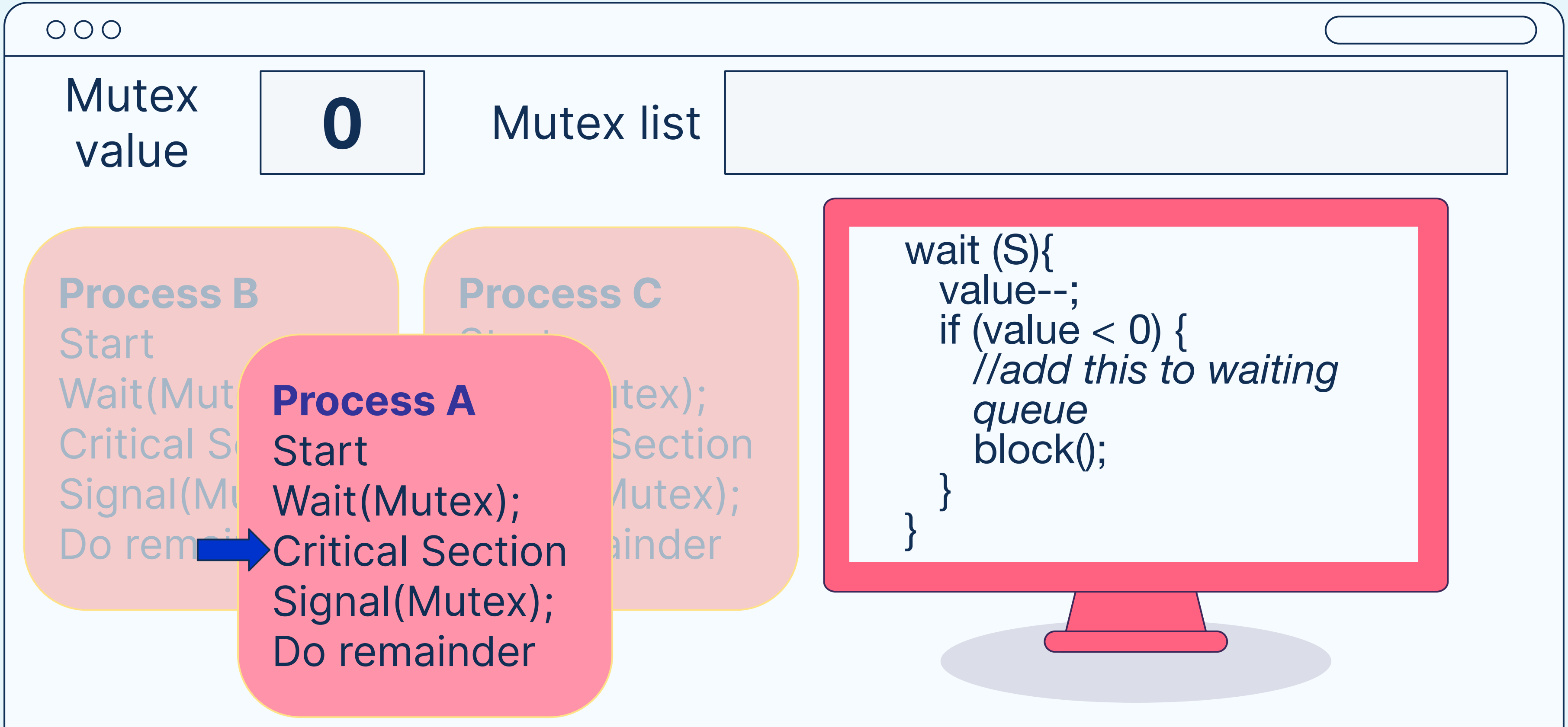
Semaphore implementation with No Busy Waiting



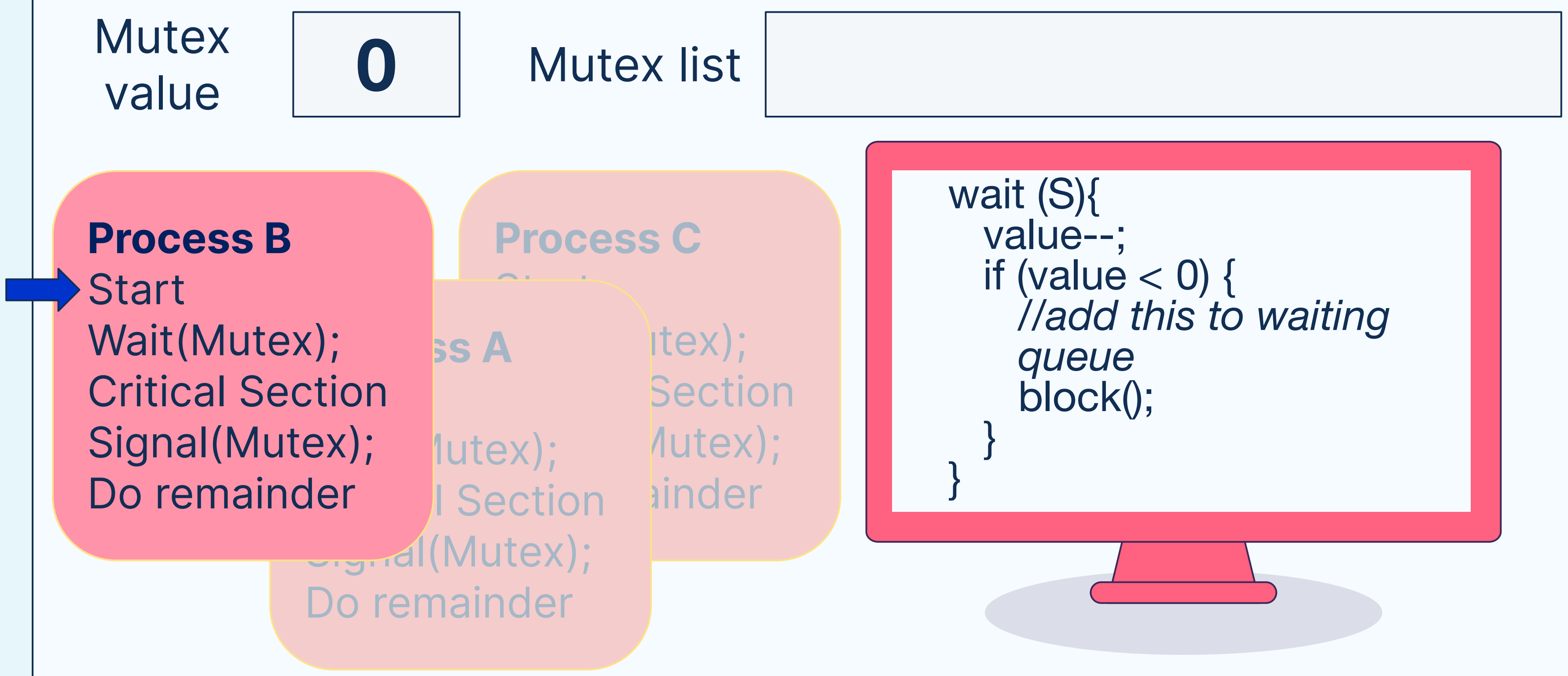
Mutex value
1
Mutex list
Process B
Start
Process C
Process A
Start
Wait(Mutex);
Critical Section
Signal(Mutex);
Do remainder
wait (S){
value--;
if (value < 0) {
//add this to waiting queue
block();
}
}
Semaphore implementation with No Busy Waiting



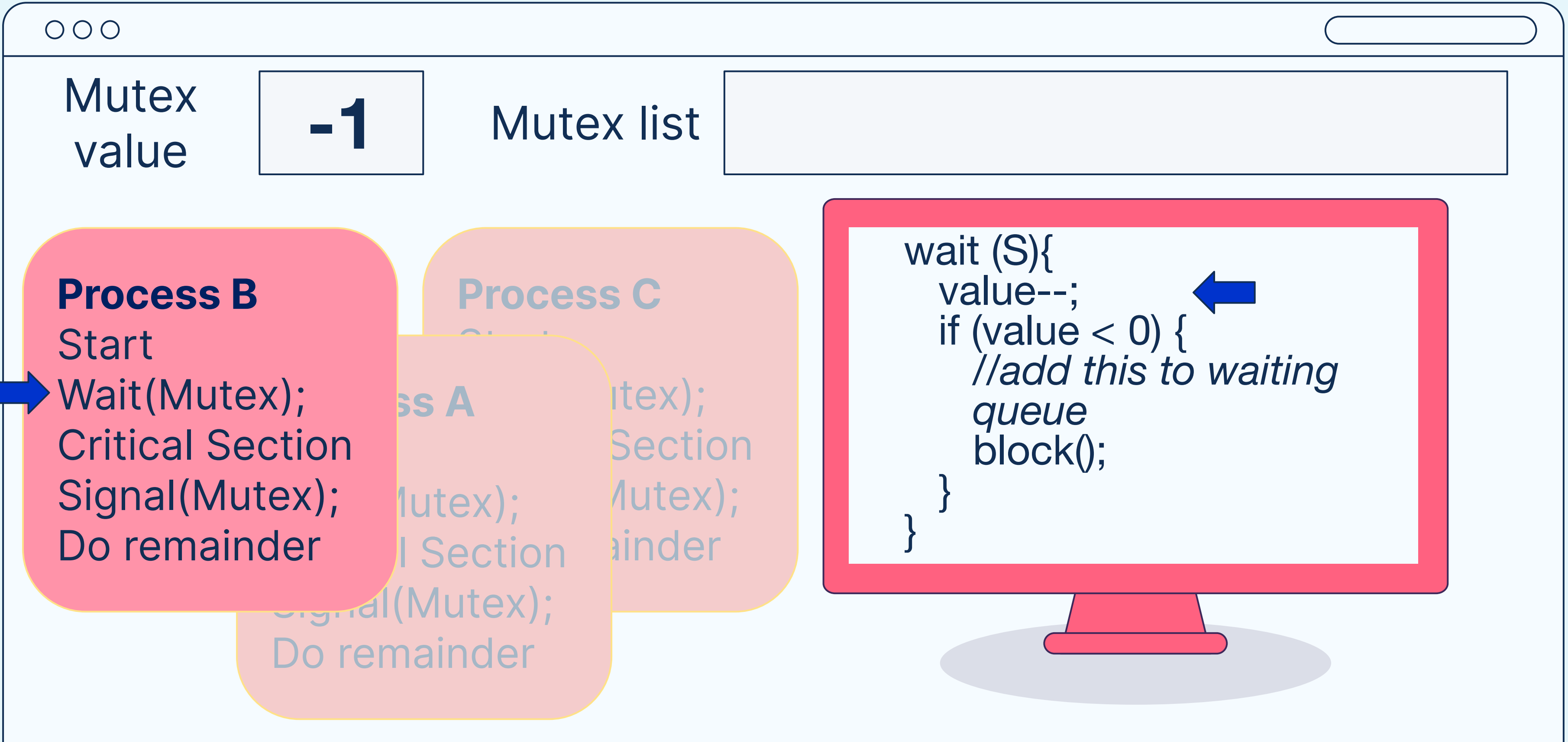
Semaphore implementation with No Busy Waiting



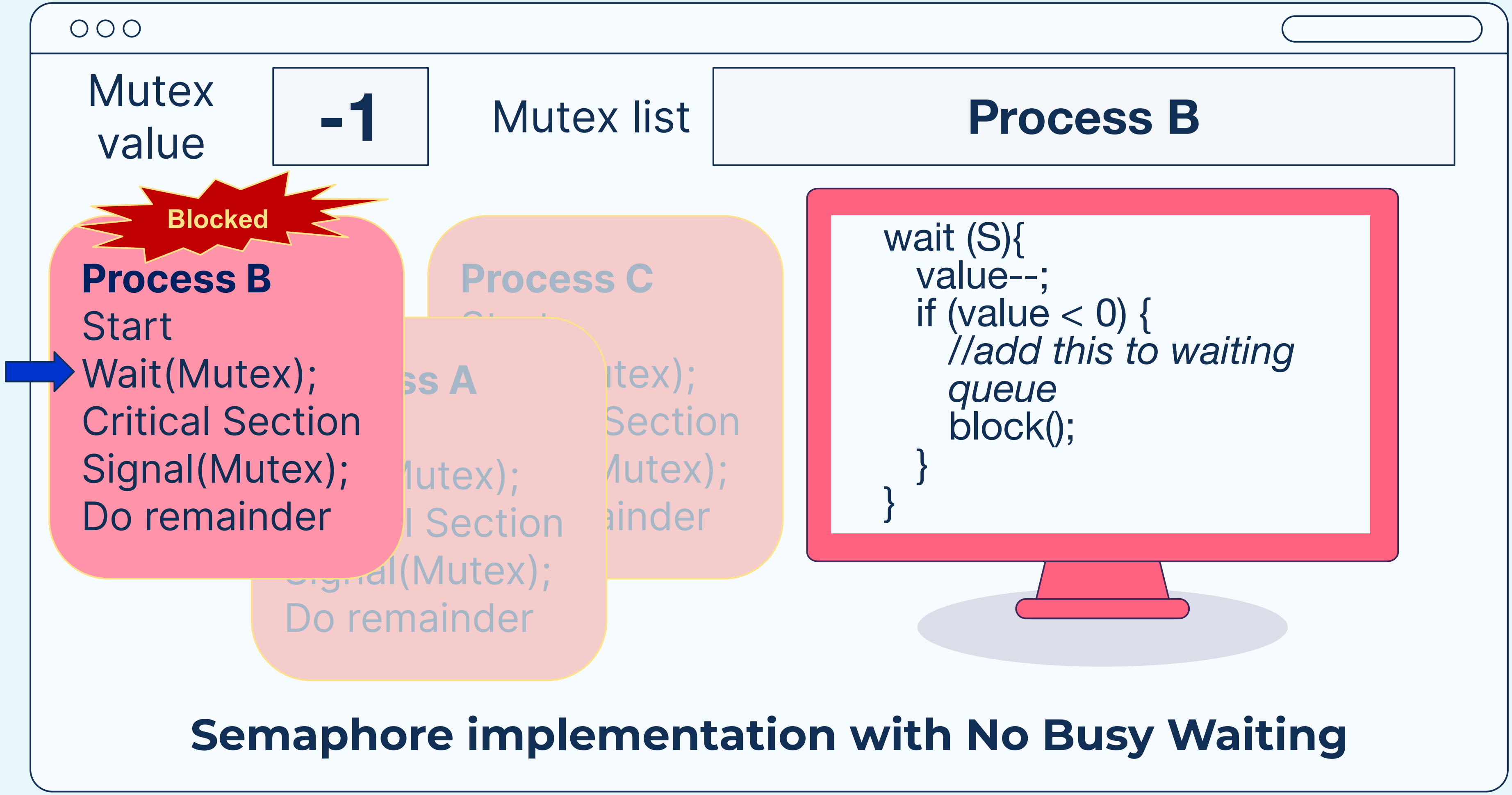
Semaphore implementation with No Busy Waiting



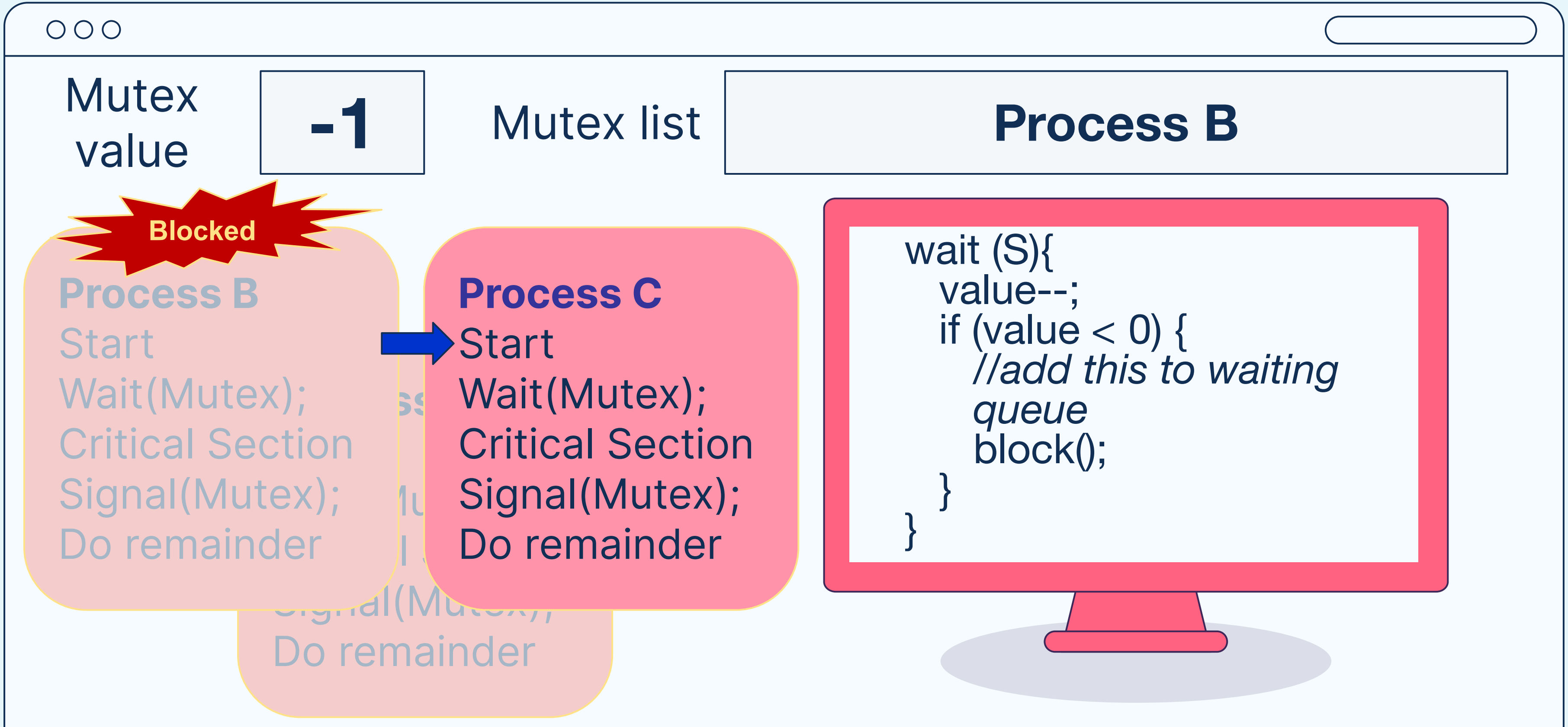
Semaphore implementation with No Busy Waiting



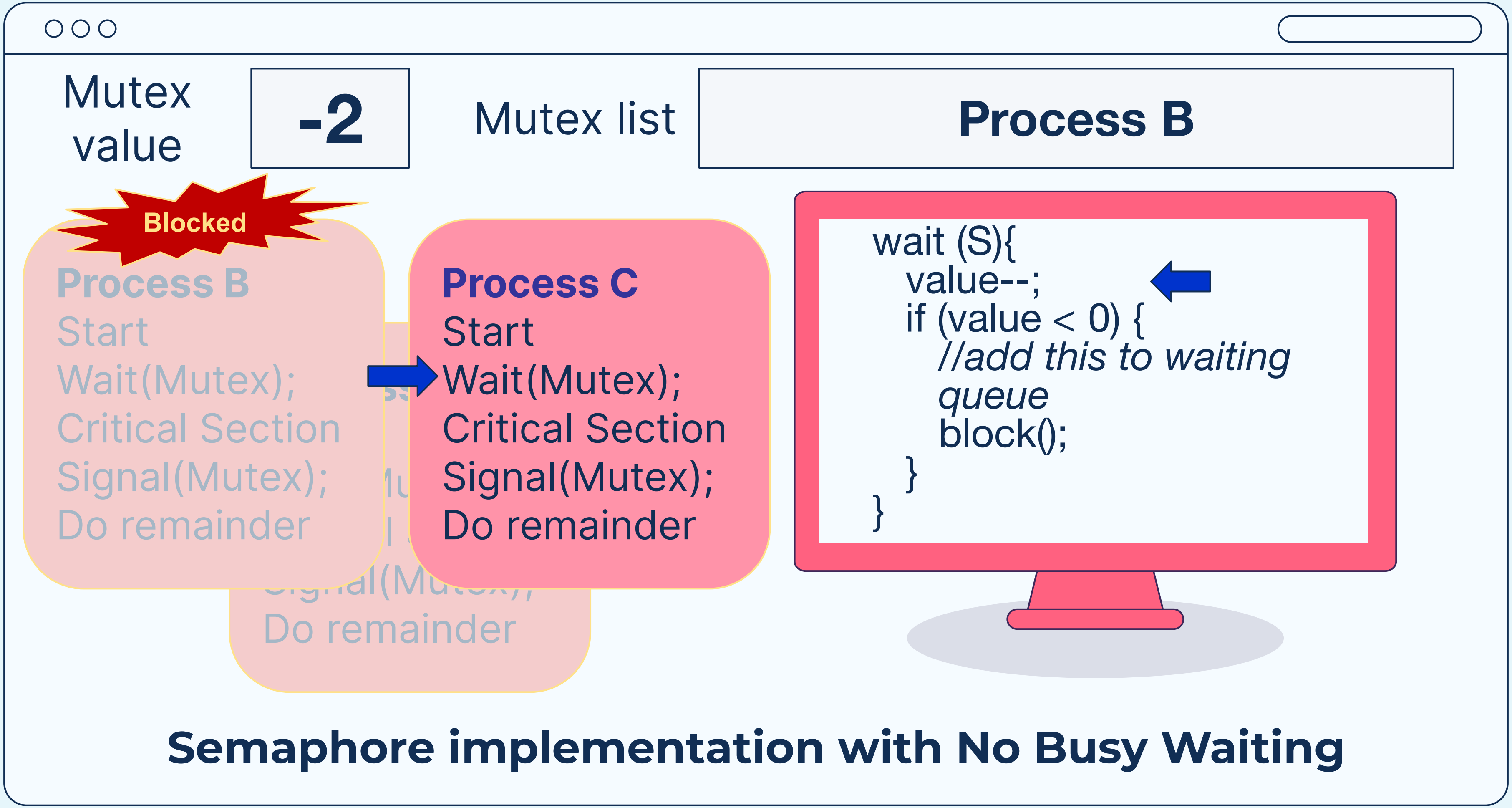
Semaphore implementation with No Busy Waiting



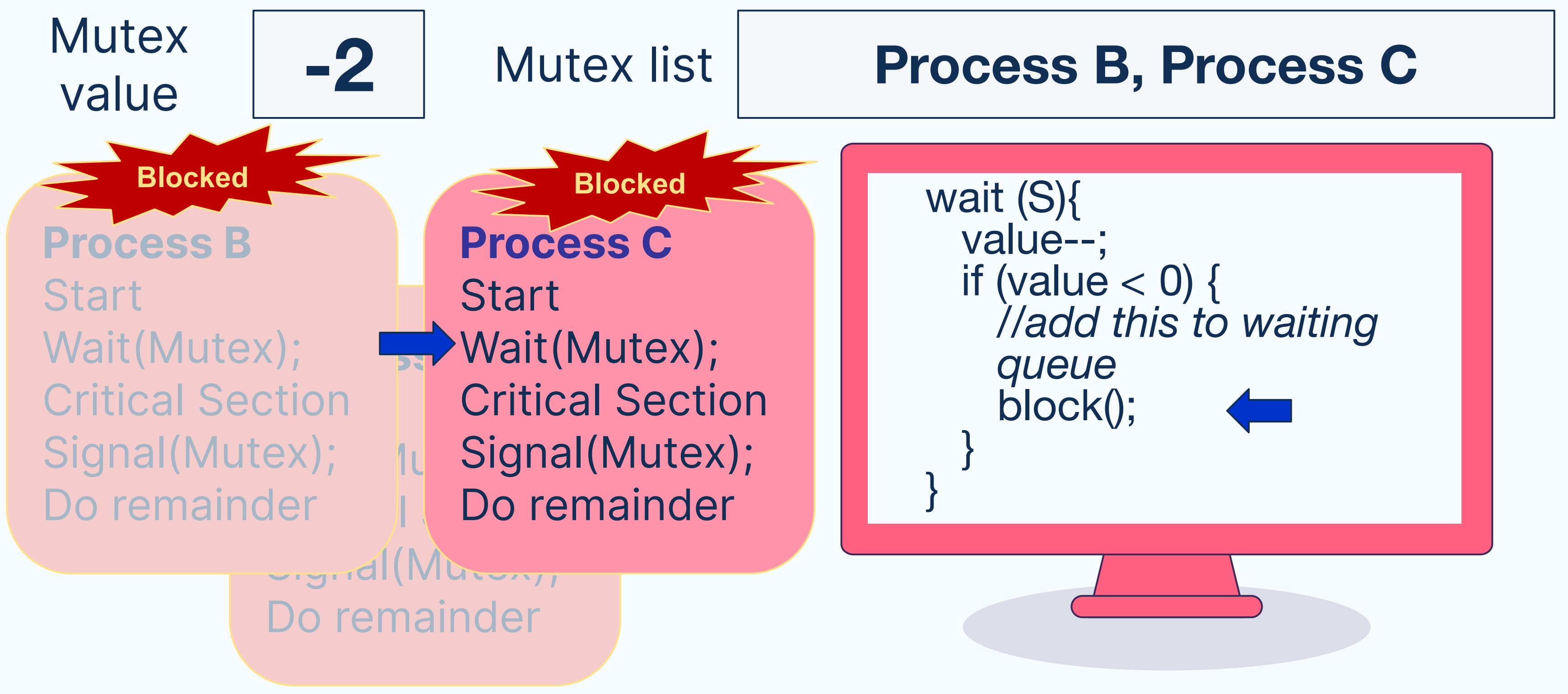
Mutex value
-1
Mutex list
Process B
Blocked
Process B
Start
Wait(Mutex);
Critical Section
Signal(Mutex);
Do remainder
Process C
Do remainder
wait (S){
value--;
if (value < 0) {
//add this to waiting queue
block();
}
}
Semaphore implementation with No Busy Waiting



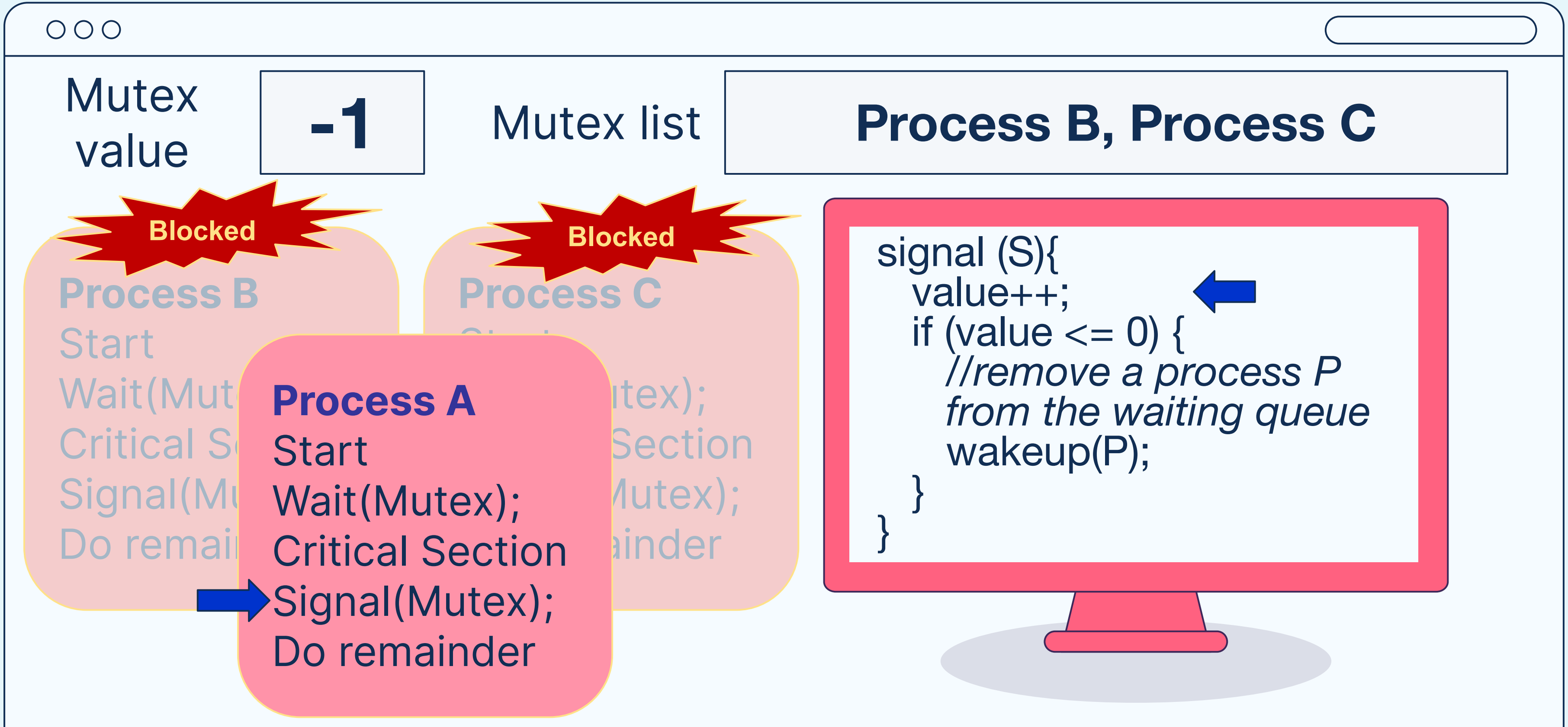
Semaphore implementation with No Busy Waiting



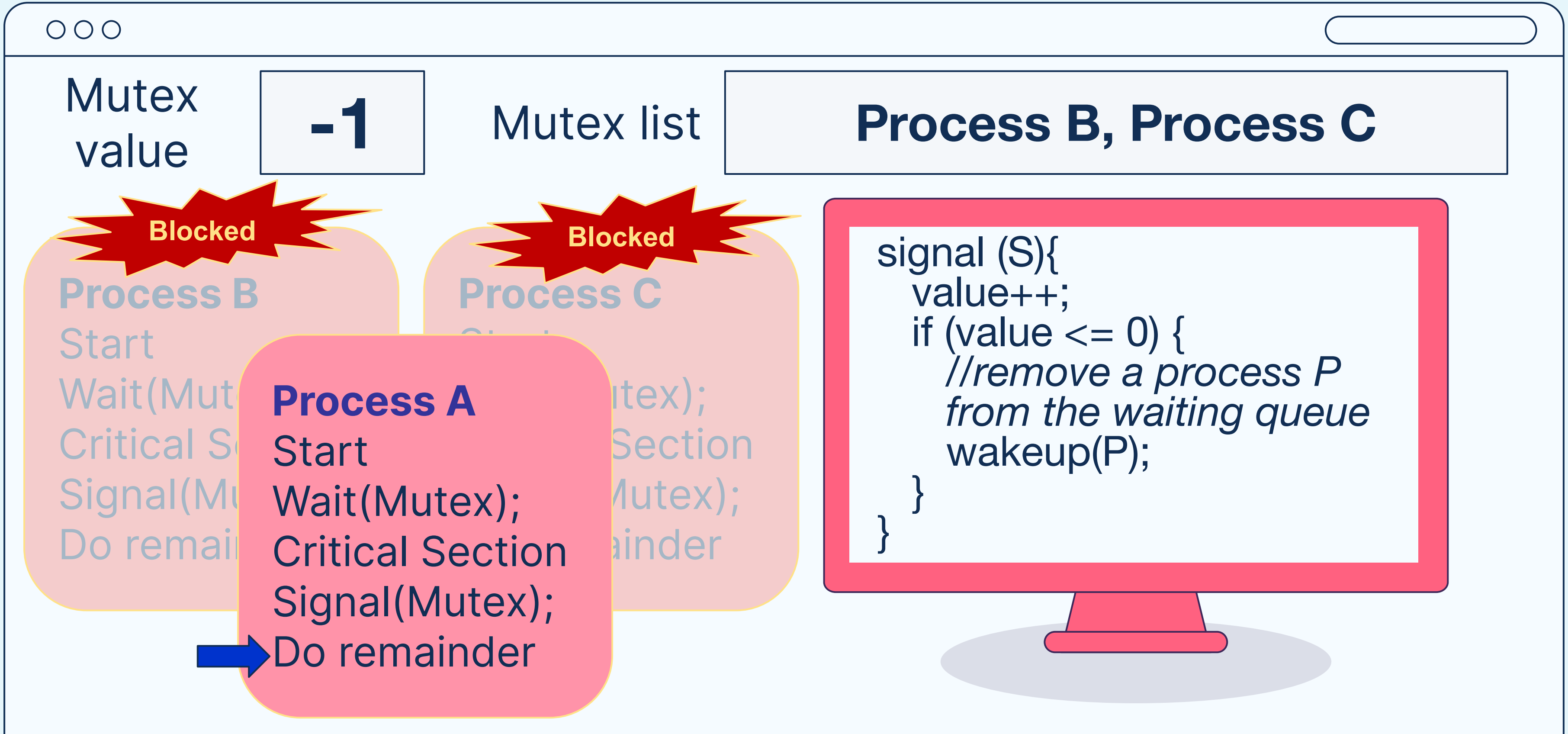
Mutex value
-2
Mutex list
Process B
Blocked
Process B
Start
Wait(Mutex);
Critical Section
Signal(Mutex);
Do remainder
Do remainder
Process C
Start
Wait(Mutex);
Critical Section
Signal(Mutex);
Do remainder
wait (S){
value--;
if (value < 0) {
//add this to waiting queue
block();
}
}
Semaphore implementation with No Busy Waiting



Semaphore implementation with No Busy Waiting



Semaphore implementation with No Busy Waiting



Semaphore implementation with No Busy Waiting

Mutex value **0**

Mutex list **Process C**

Blocked

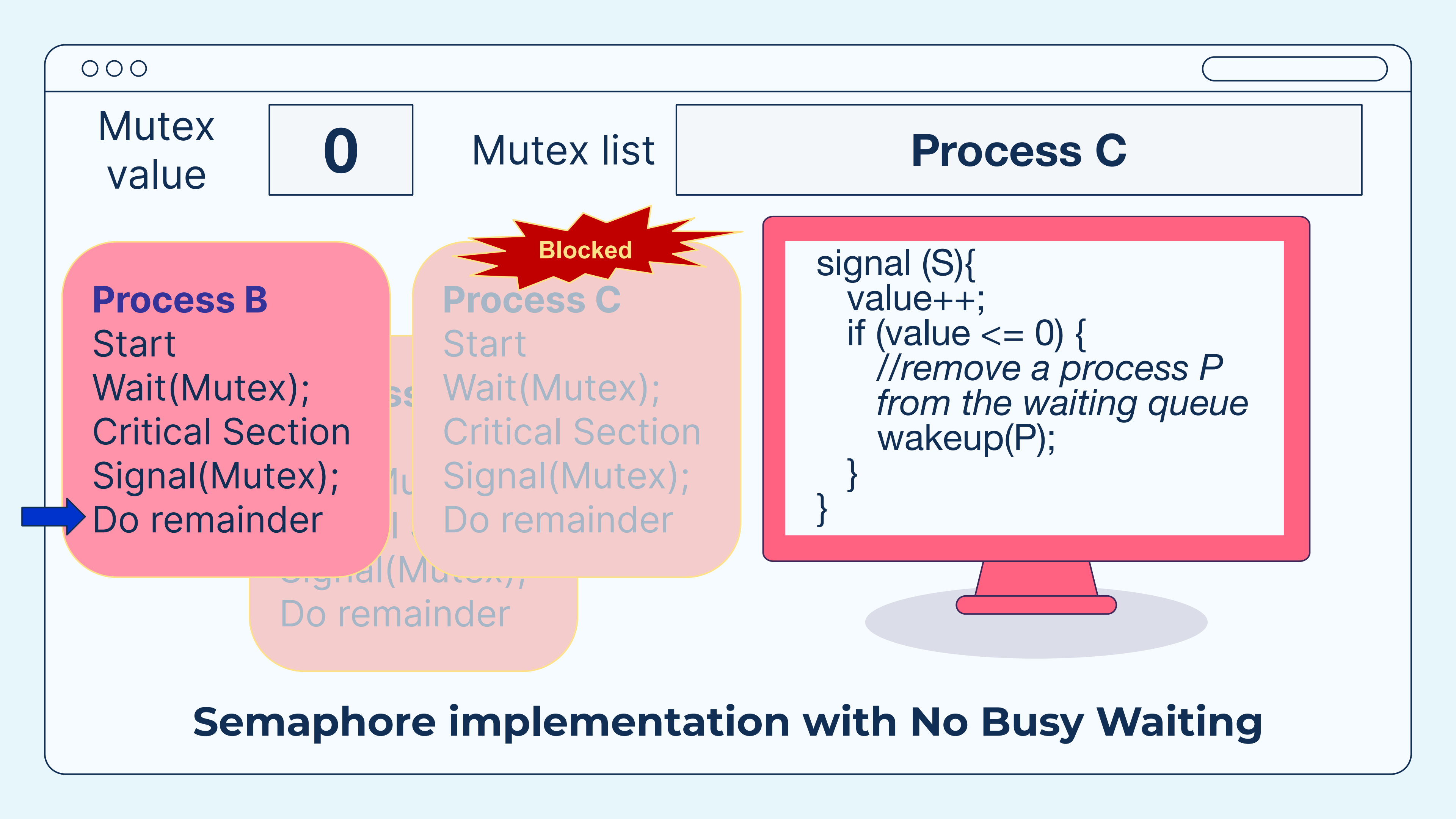
Process B
Start
Wait(Mutex);
Critical Section
Signal(Mutex);
Do remainder

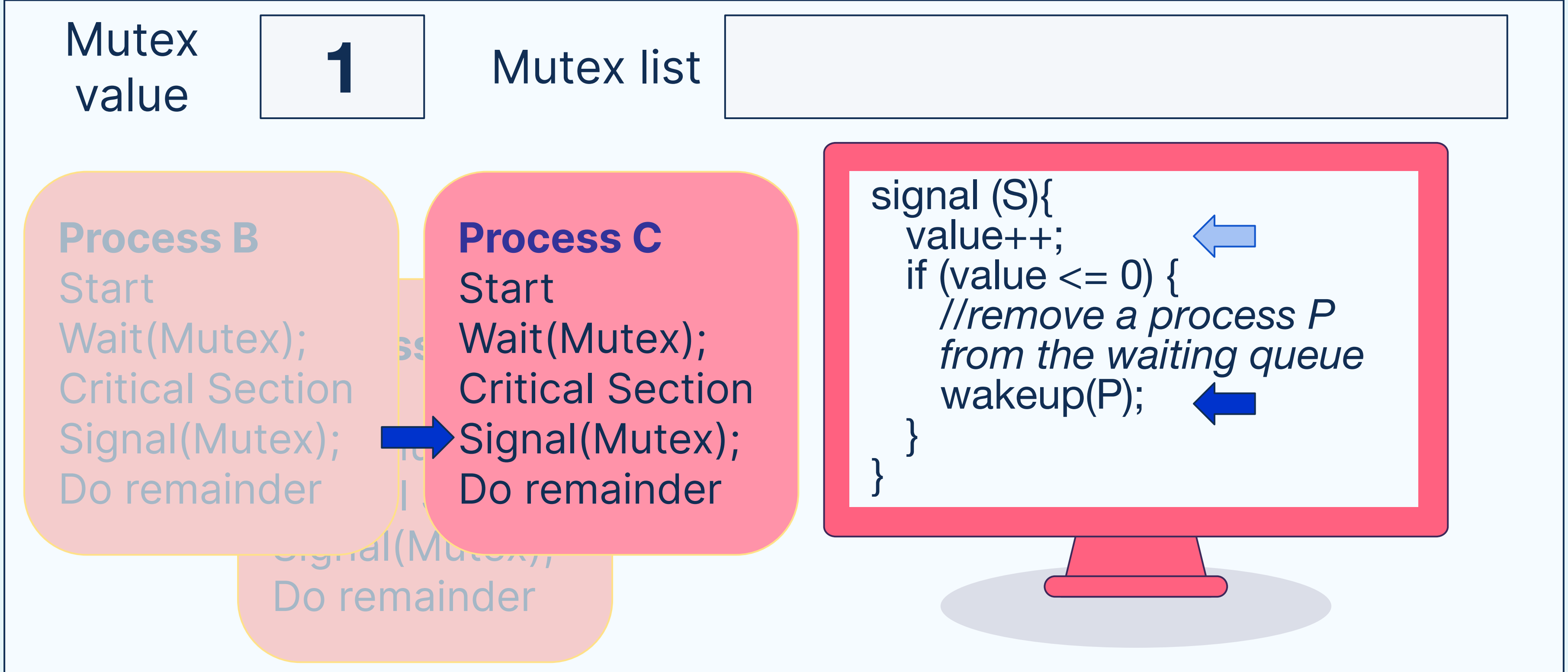
Process C
Start
Wait(Mutex);
Critical Section
Signal(Mutex);
Do remainder

Do remainder

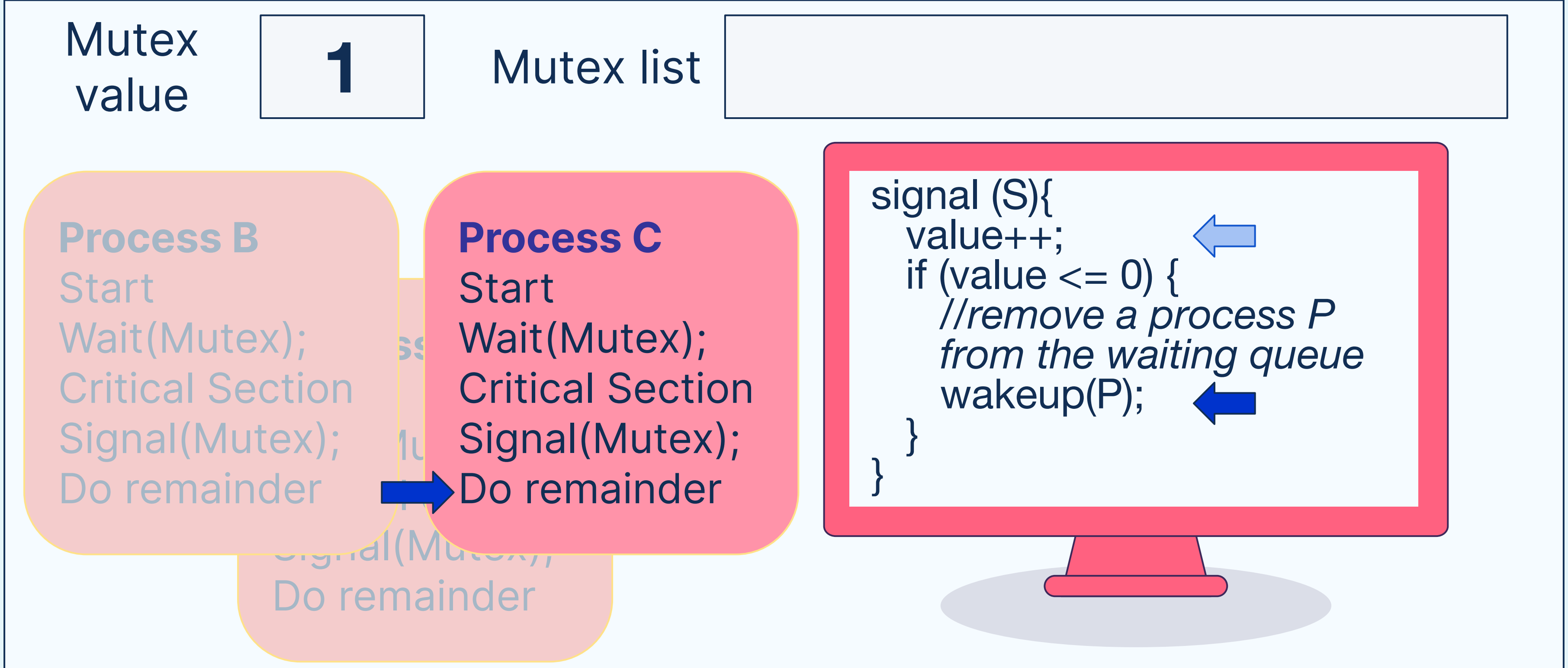
```
signal (S){
   value++;
   if (value <= 0) {
      //remove a process P
      from the waiting queue
      wakeup(P);
   }
}
```

Semaphore implementation with No Busy Waiting


Mutex value
0
Mutex list
Process C
Blocked
Process B
Start
Wait(Mutex);
Critical Section
Signal(Mutex);
Do remainder
Process C
Start
Wait(Mutex);
Critical Section
Signal(Mutex);
Do remainder
Do remainder
signal (S){
value++;
if (value <= 0) {
//remove a process P
from the waiting queue
wakeup(P);
}
}
Semaphore implementation with No Busy Waiting



Semaphore implementation with No Busy Waiting



Semaphore implementation with No Busy Waiting

Deadlock and Starvation

The implementation of a semaphore with a **waiting queue** may result in a situation where two or more processes are waiting for each other.

Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes

Let S and Q be two semaphores initialized to 1

	S	Q
value	1	1

```
wait (S){
     value--;

     if (value < 0) {
          //add this process to
          waiting queue

          block();
    }
}
```

P_0	P_1
wait (S);	wait (Q);
wait (Q);	wait (S);
.	.
.	.
signal (S);	signal (Q);
signal (Q);	signal (S);

Deadlock and Starvation

The implementation of a semaphore with a **waiting queue** may result in a situation where two or more processes are waiting for each other.

Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes

Let S and Q be two semaphores initialized to 1

	S	Q
value	0	1

```
wait (S){
    value--;

    if (value < 0) {
        //add this process to
        waiting queue

        block();
    }
}
```

P_0

```
wait (S);
wait (Q);
    .
    .
signal (S);
signal (Q);
```

P_1

```
wait (Q);
wait (S);
    .
    .
signal (Q);
signal (S);
```

Deadlock and Starvation

The implementation of a semaphore with a **waiting queue** may result in a situation where two or more processes are waiting for each other.

Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes

Let S and Q be two semaphores initialized to 1

	S	Q
value	0	0

```
wait (S){
    value--;

    if (value < 0) {
        //add this process to
        waiting queue

        block();
    }
}
```

```
P0                  P1
wait (S);           wait (Q);
wait (Q);           wait (S);
   .                   .
   .                   .
signal (S);         signal (Q);
signal (Q);         signal (S);
```

Deadlock and Starvation

The implementation of a semaphore with a **waiting queue** may result in a situation where two or more processes are waiting for each other.

Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes

Let S and Q be two semaphores initialized to 1

	S	Q
value	0	-1

```
wait (S){
     value--;

     if (value < 0) {
          //add this process to
          waiting queue

          block();
    }
}
```

P_0	P_1
`wait (S);`	`wait (Q);`
`wait (Q);`	`wait (S);`
.	.
.	.
`signal (S);`	`signal (Q);`
`signal (Q);`	`signal (S);`

Deadlock and Starvation

The implementation of a semaphore with a **waiting queue** may result in a situation where two or more processes are waiting for each other.

Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes

Let S and Q be two semaphores initialized to 1

	S	Q
value	-1	-1

```
wait (S){
     value--;

     if (value < 0) {
          //add this process to
          waiting queue

          block();
     }
}
```

P_0	P_1
`wait (S);`	`wait (Q);`
`wait (Q);`	`wait (S);`
.	.
.	.
`signal (S);`	`signal (Q);`
`signal (Q);`	`signal (S);`

Deadlock and Starvation

The implementation of a semaphore with a **waiting queue** may result in a situation where two or more processes are waiting for each other.

Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes

Let S and Q be two semaphores initialized to 1

	S	Q
value	-1	-1

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

    if (value <= 0) {
        //remove a process P
        from the waiting queue

        wakeup(P);
    }
}
```

P_0	P_1
wait (S);	wait (Q);
wait (Q);	wait (S);
.	.
.	.
signal (S);	signal (Q);
signal (Q);	signal (S);

Deadlock and Starvation

The implementation of a semaphore with a **waiting queue** may result in a situation where two or more processes are waiting for each other.

Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes

Let S and Q be two semaphores initialized to 1

	S	Q
value	1	1

```
wait (S) {
    while S ≤ 0 ; // no-op
    S--;
}
```

P_0	P_1
wait (S);	wait (Q);
wait (Q);	wait (S);
.	.
.	.
signal (S);	signal (Q);
signal (Q);	signal (S);

Deadlock and Starvation

The implementation of a semaphore with a **waiting queue** may result in a situation where two or more processes are waiting for each other.

Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes

Let S and Q be two semaphores initialized to 1

	S	Q
value	0	1

```
wait (S) {
    while S ≤ 0 ; // no-op
    S--;
}
```

P_0	P_1
wait (S);	wait (Q);
wait (Q);	wait (S);
.	.
.	.
signal (S);	signal (Q);
signal (Q);	signal (S);

Deadlock and Starvation

The implementation of a semaphore with a **waiting queue** may result in a situation where two or more processes are waiting for each other.

Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes

Let S and Q be two semaphores initialized to 1

	S	Q
value	0	0

```
wait (S) {
    while S ≤ 0 ; // no-op
    S--;
}
```

P_0	P_1
wait (S);	wait (Q);
wait (Q);	wait (S);
.	.
.	.
signal (S);	signal (Q);
signal (Q);	signal (S);

Deadlock and Starvation

The implementation of a semaphore with a **waiting queue** may result in a situation where two or more processes are waiting for each other.

Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes

Let S and Q be two semaphores initialized to 1

	S	Q
value	0	0

```
wait (S) {
    while S ≤ 0 ; // no-op
    S--;
}
```

P_0	P_1
wait (S);	wait (Q);
wait (Q);	wait (S);
.	.
.	.
signal (S);	signal (Q);
signal (Q);	signal (S);

Deadlock and Starvation

The implementation of a semaphore with a **waiting queue** may result in a situation where two or more processes are waiting for each other.

Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes

Let S and Q be two semaphores initialized to 1

	S	Q
value	0	0

```
wait (S) {
    while S ≤ 0 ; // no-op
    S--;
}
```

P_0	P_1
wait (S);	wait (Q);
wait (Q);	wait (S);
.	.
.	.
signal (S);	signal (Q);
signal (Q);	signal (S);

Deadlock and Starvation

The implementation of a semaphore with a **waiting queue** may result in a situation where two or more processes are waiting for each other.

Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes

Let S and Q be two semaphores initialized to 1

	S	Q
value	0	0

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

P_0	P_1
`wait (S);`	`wait (Q);`
`wait (Q);`	`wait (S);`
.	.
.	.
`signal (S);`	`signal (Q);`
`signal (Q);`	`signal (S);`

Explaination

Suppose that P0 executes wait(S) and then P1 executes wait(Q).

When P0 executes wait(Q), it must wait until P1 executes signal(Q).

Similarly, when P1 executes wait(S), it must wait P0 executes signal(S).

Since these signal() operations cannot be executed, P0 and P1 are deadlock.

Problems with semaphore

Incorrect use of semaphore operations:

01 **signal (mutex) .... wait (mutex)**

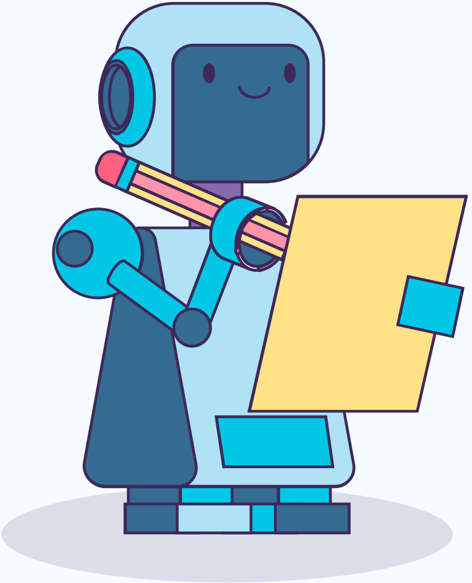
Several processes may execute in the CS

02 **wait (mutex) ... wait (mutex)**

Deadlock will occur

03 **Omitting of wait (mutex) or/and signal (mutex)**

Several processes may execute in the CS



Thank you

CREDITS: This presentation template was created by **Slidesgo**, and includes icons by **Flaticon**, and infographics & images by **Freepik**