Chapter 5: Process Synchronization (Part 3)

Chapter 5: Process Synchronization

- Background
- The Critical-Section Problem
- Peterson's Solution
- Synchronization Hardware
- Mutex Locks
- **Semaphores**
- Classic Problems of Synchronization
- Monitors
- Condition Variables

Semaphore

- Synchronization tool that provides more sophisticated ways (than **Mutex locks**) for process to synchronize their activities.
- originally, semaphores were **flags or lights** for signaling between ships
- In computer science, a **semaphore** can be used to solve several different types of synchronization problems.

- **Semaphore *S*** – integer variable used for signaling between processes

- A semaphore S is an integer variable that, *apart from initialization*, can only be modified through **two** atomic and mutually exclusive operations:
  - **Wait (S)**
  - **Signal (S)**
    - Originally called **`P()`** and **`V()`**
- If a process is waiting for a signal, it is suspended until that signal is sent

Definition of Wait() and Signal()

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

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

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

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

Semaphore Usage

- **Counting semaphore** – integer value can range over an unrestricted domain
  - Can be used to control access to resources (semaphore S initialized to number of resources)
  - a resource such as a shared data structure is protected by a semaphore. You must **acquire** the semaphore before using the resource and **release** the semaphore when you are done with the shared resource.
    - Process that wishes to use resource will execute wait(S)
    - Process that releases a resource will execute signal(S)
    - If S = 0 then all resources are being used, and, Processes block until S > 0;
- **Binary semaphore** – integer value can range only between 0 and 1
  - Same as a **mutex lock**
- Can solve various synchronization problems (See next slide to know how)

Semaphore Usage (Cont'd)

- **Process Synchronization with Semaphores** - Consider concurrent processes P_1 and P_2 that require statement S_1 to happen before S_2

  Create a semaphore "**synch**" shared by P_1 and P_2 and initialized to 0

  Process P1:

  ```
  S1;                 //execute statement S1
  signal(synch); //increment semaphore synch
  ```

  Process P2:

  ```
  wait(synch); //decrement Semaphore synch
  S2; //execute statement S2
  ```

- Can implement a counting semaphore ***S*** as a binary semaphore if ***S*** is initialized to 1

Semaphore Implementation (wait and signal must be atomic)

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

Semaphore Implementation with no Busy loop

- For no busy waiting we associate a waiting queue with each semaphore
- 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
-

```
typedef struct{

int value;

struct process *list; //list of processes waiting on the semaphore S

  } semaphore;
```

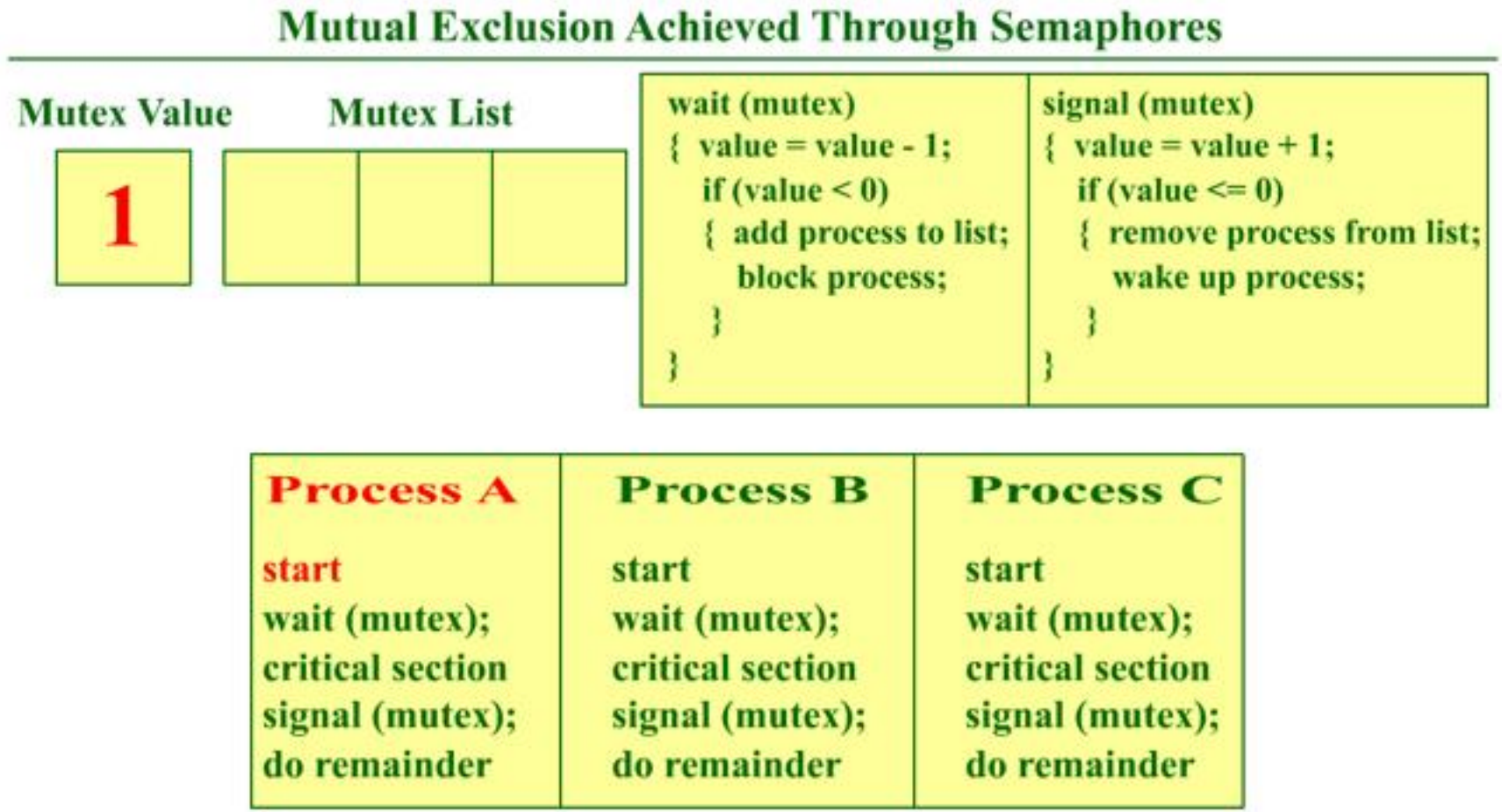
Implementation with no Busy waiting (Cont.)

```
wait(semaphore *S) {
    S->value--;
    if (S->value < 0) {
        add this process to S->list;
        block();
    }
} //wait() operation adds a process to the S queue and suspends it
signal(semaphore *S) {
    S->value++;
    if (S->value <= 0) {
        remove a process P from S->list;
        wakeup(P);
    }
}//signal() operation removes one process from S queue and awakens it
```

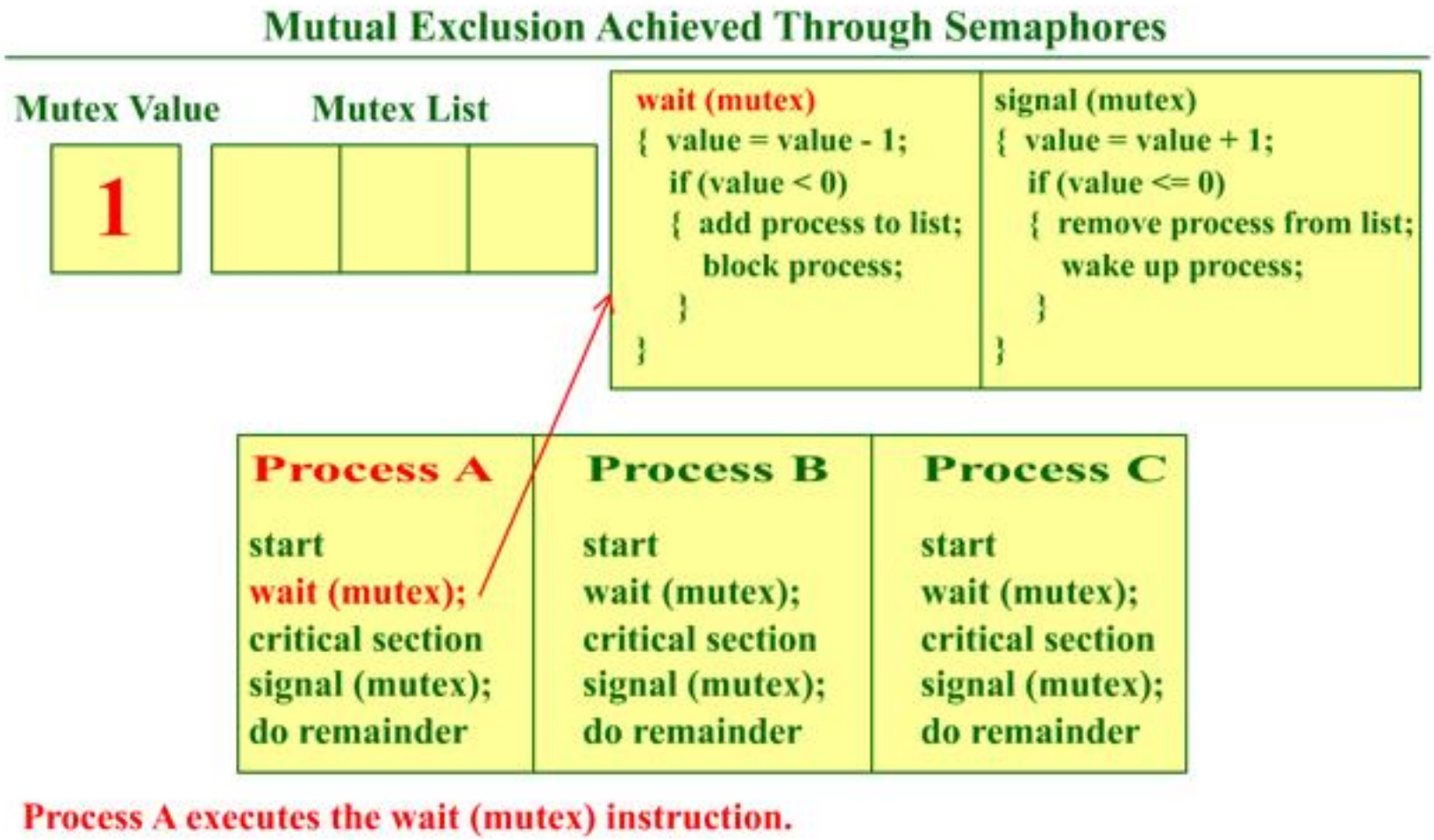
Example: Mutual Exclusion with semaphore

- Suppose there are three processes, **A**, **B**, and **C**.
- Every process wants to execute Critical Section.
- But CS is protected by a shared binary Semaphore called Mutex.
- Any process trying to execute CS will first call **wait(mutex)** and if not blocked will execute CS.
- Similarly, any process that executes CS will call **signal(mutex)** and will unblock the next blocked process waiting in the Mutex List.

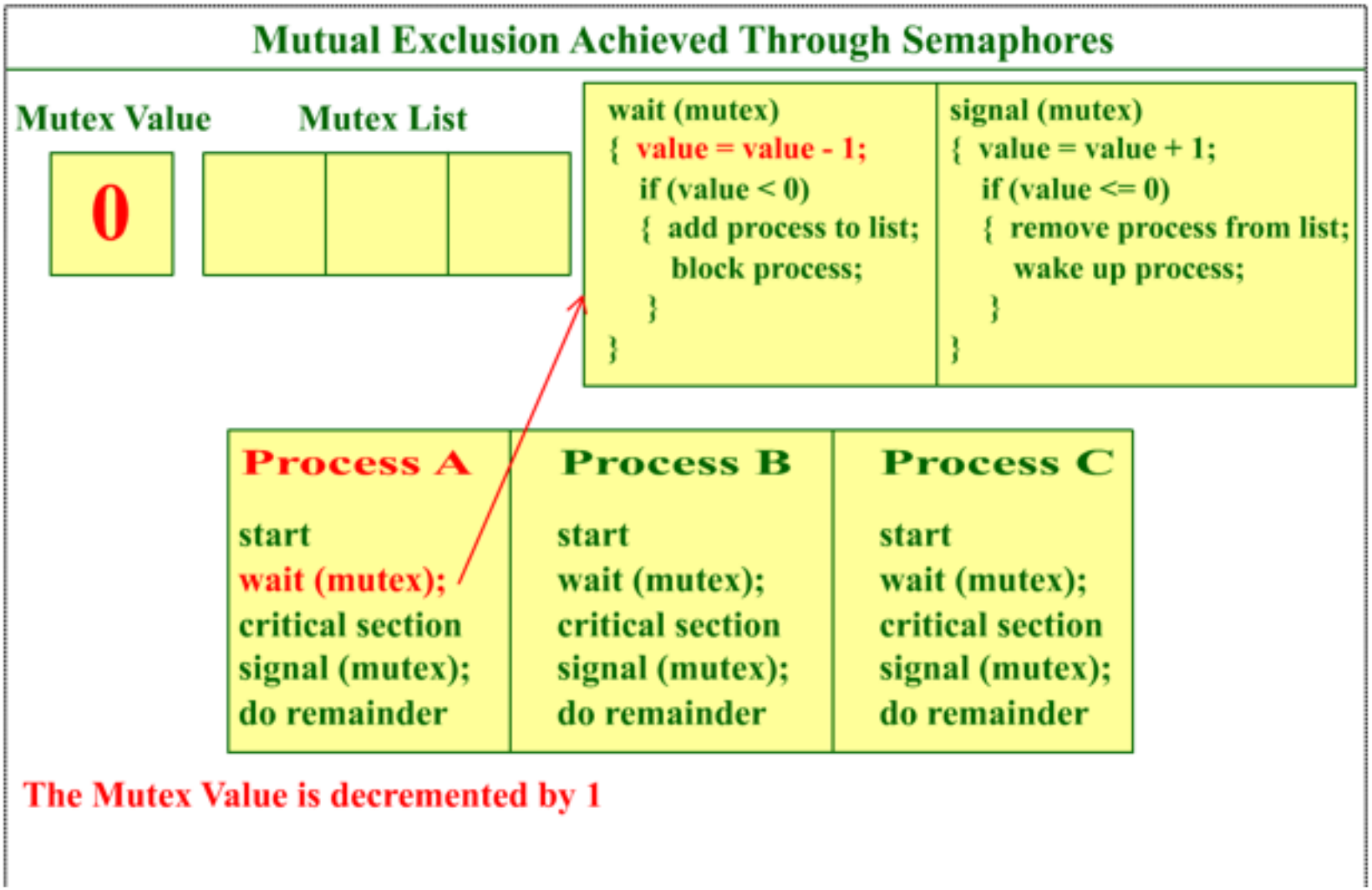
Mutual Exclusion with Semaphores (1/34)



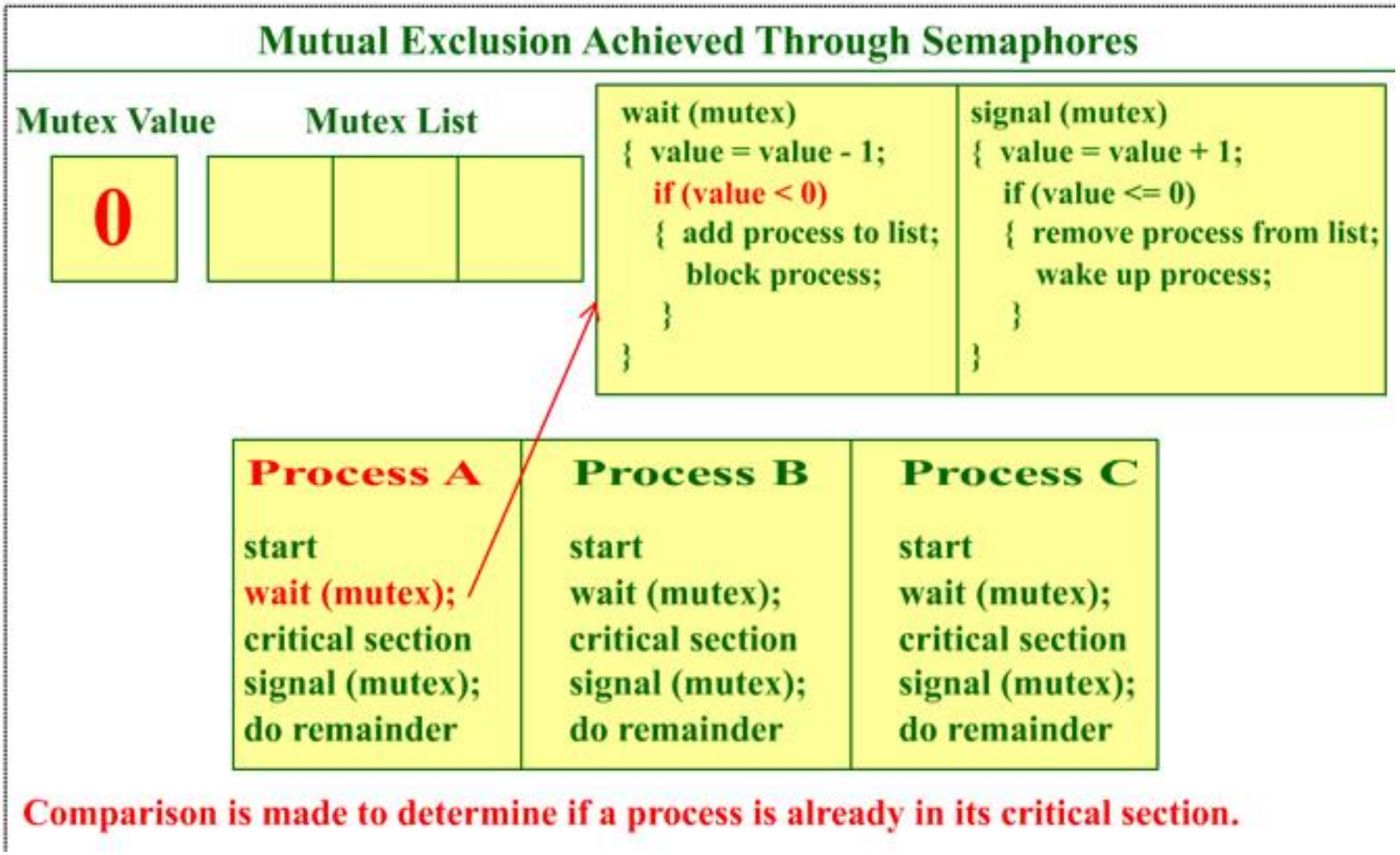
Mutual Exclusion with Semaphores (2/34)



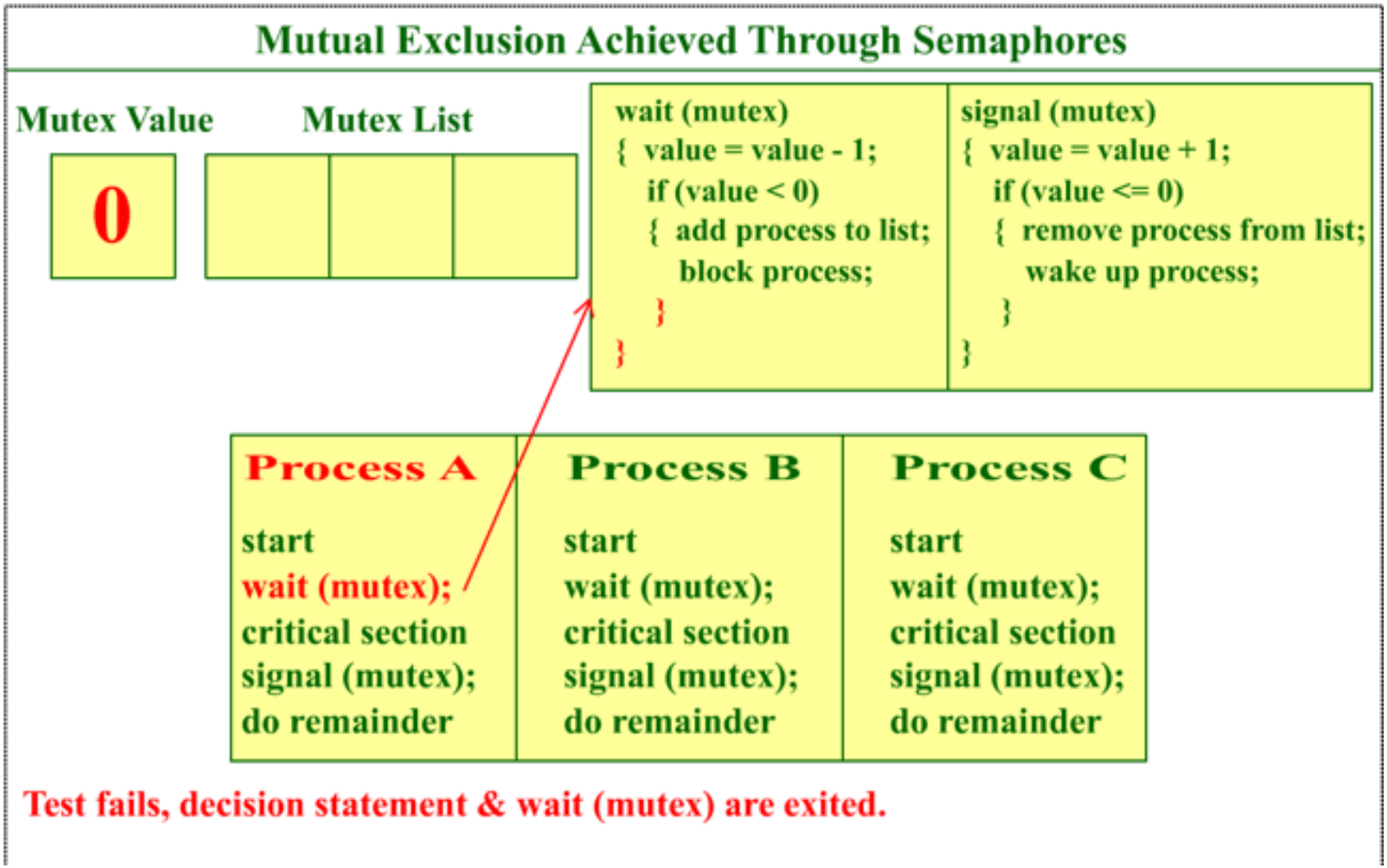
Mutual Exclusion with Semaphores (3/34)



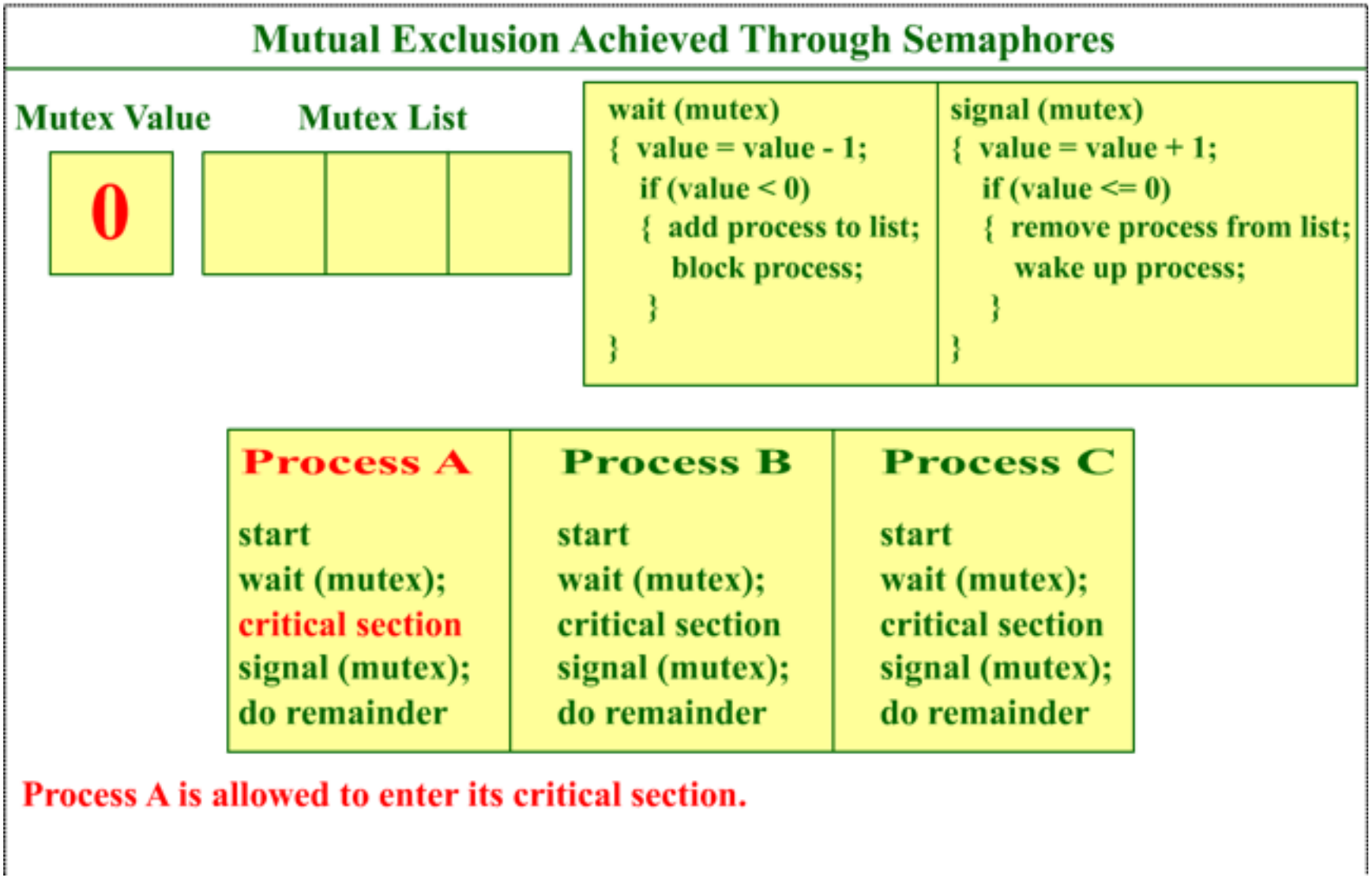
Mutual Exclusion with Semaphores (4/34)



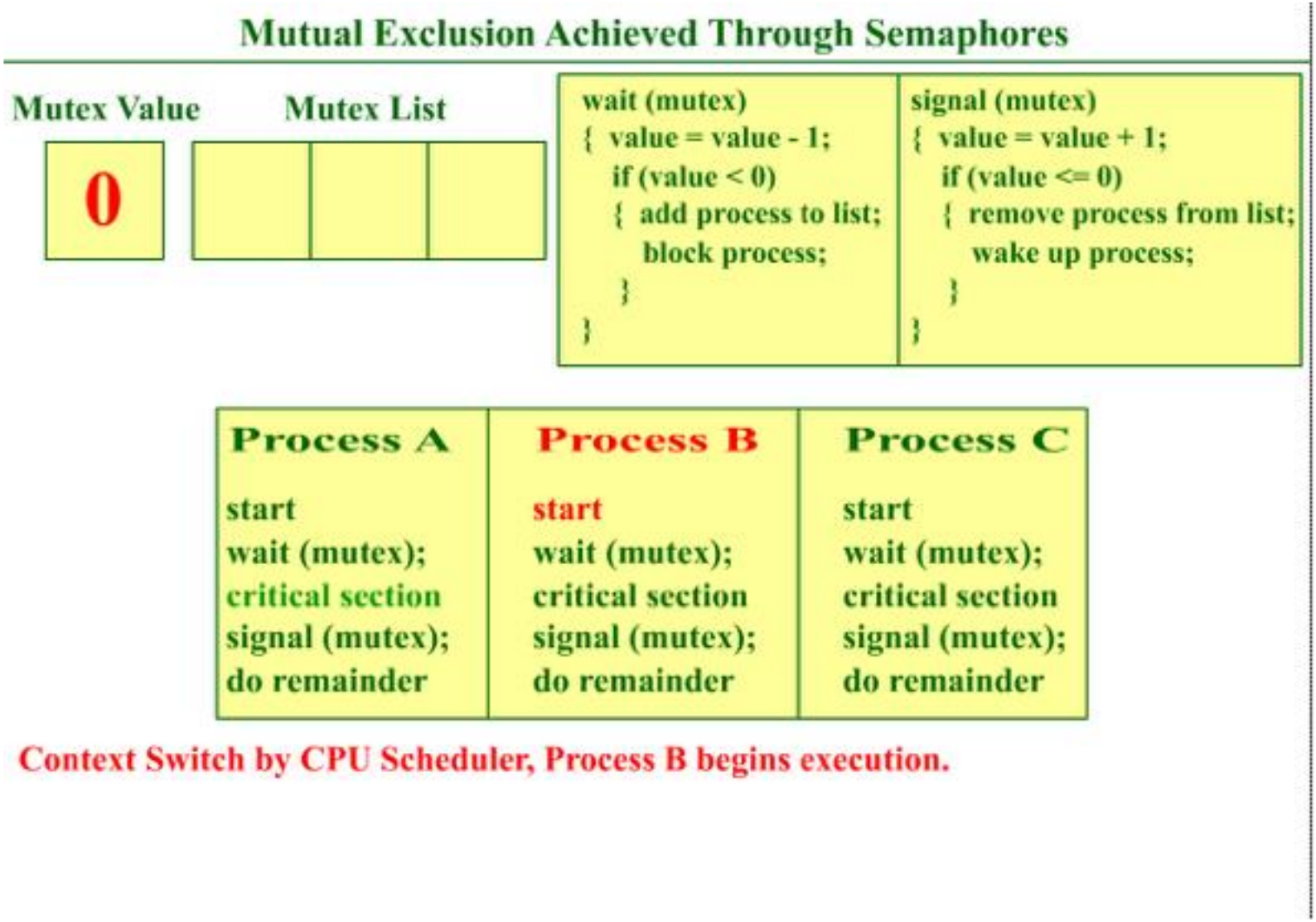
Mutual Exclusion with Semaphores (5/34)



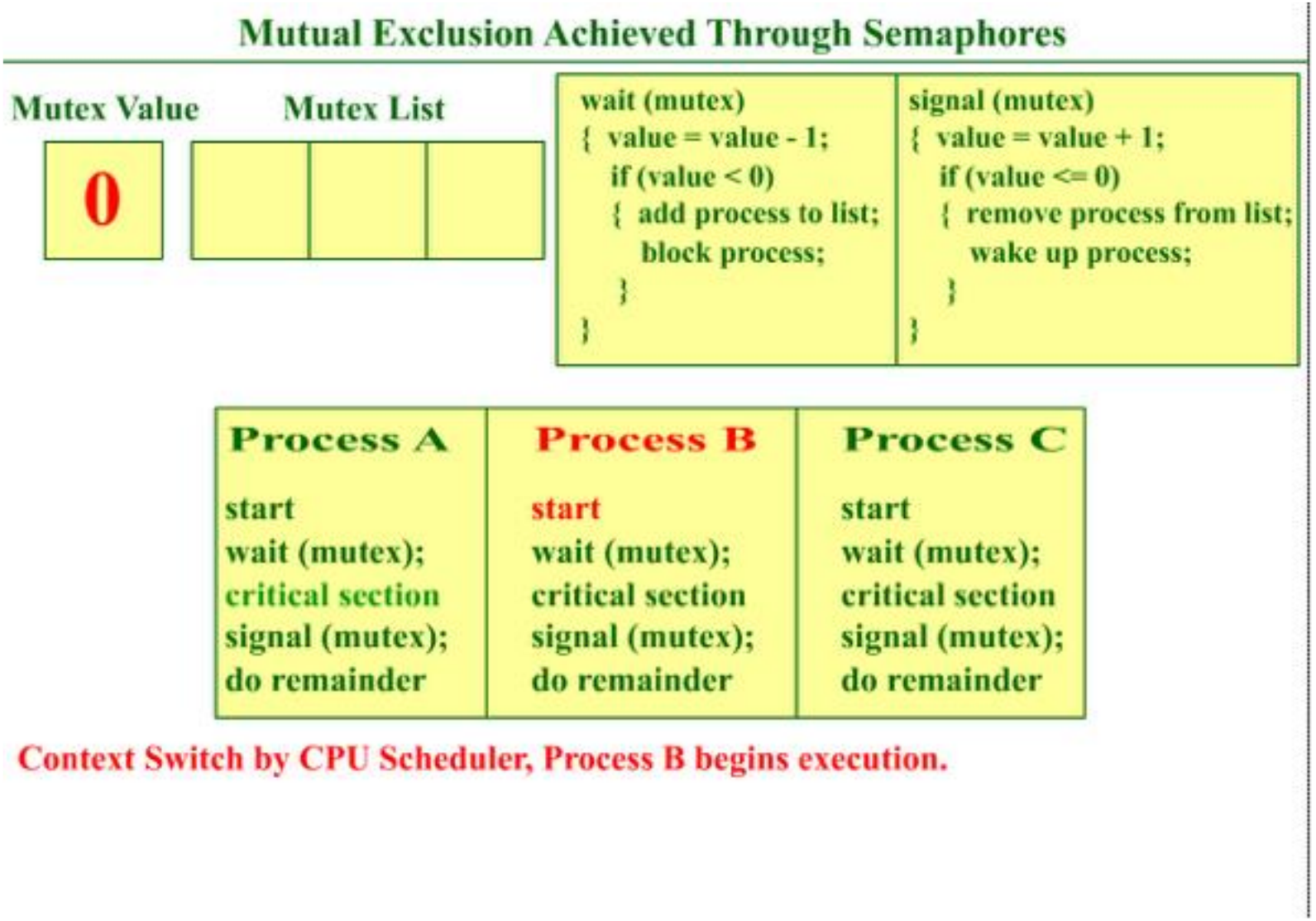
Mutual Exclusion with Semaphores (6/34)



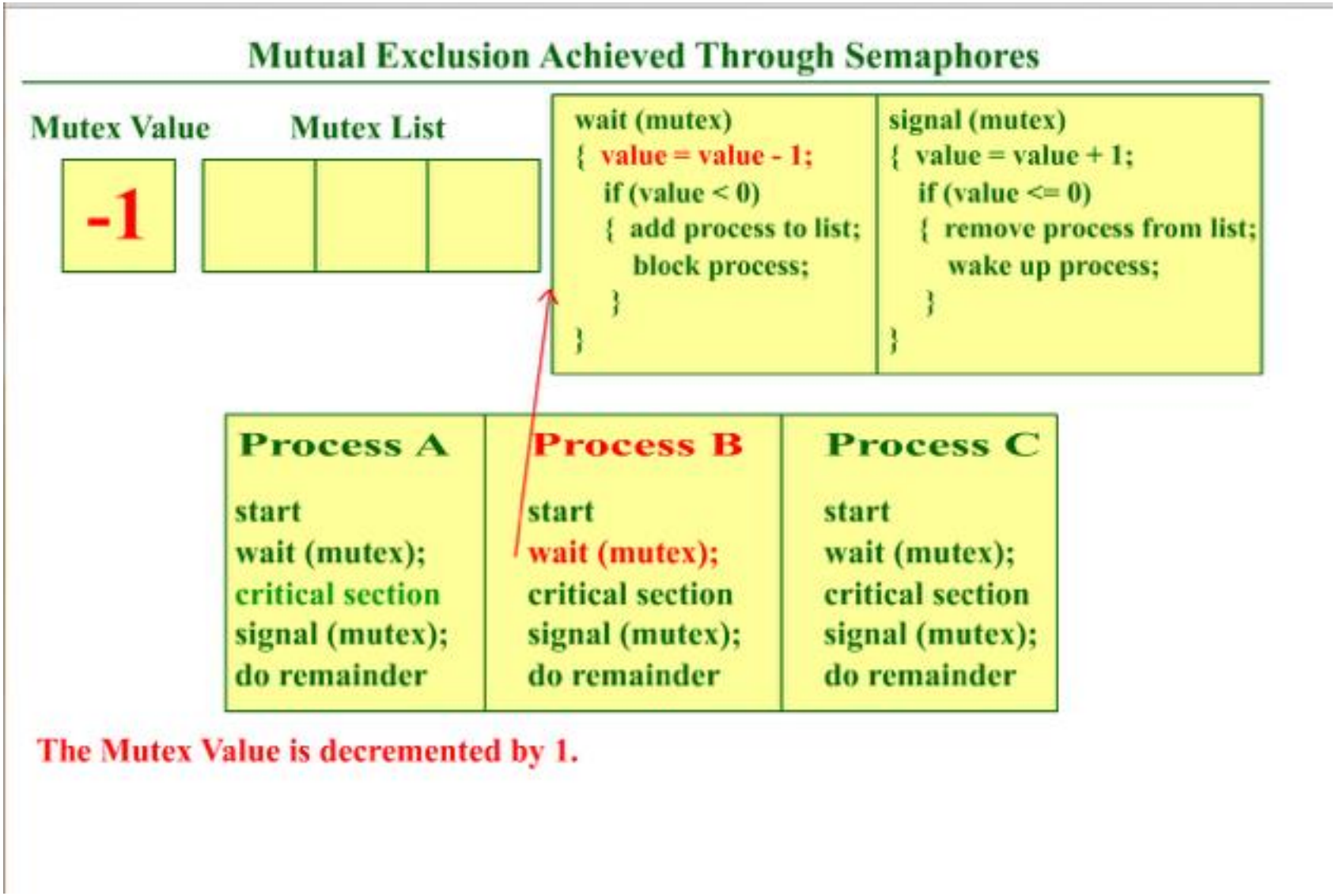
Mutual Exclusion with Semaphores (7/34)



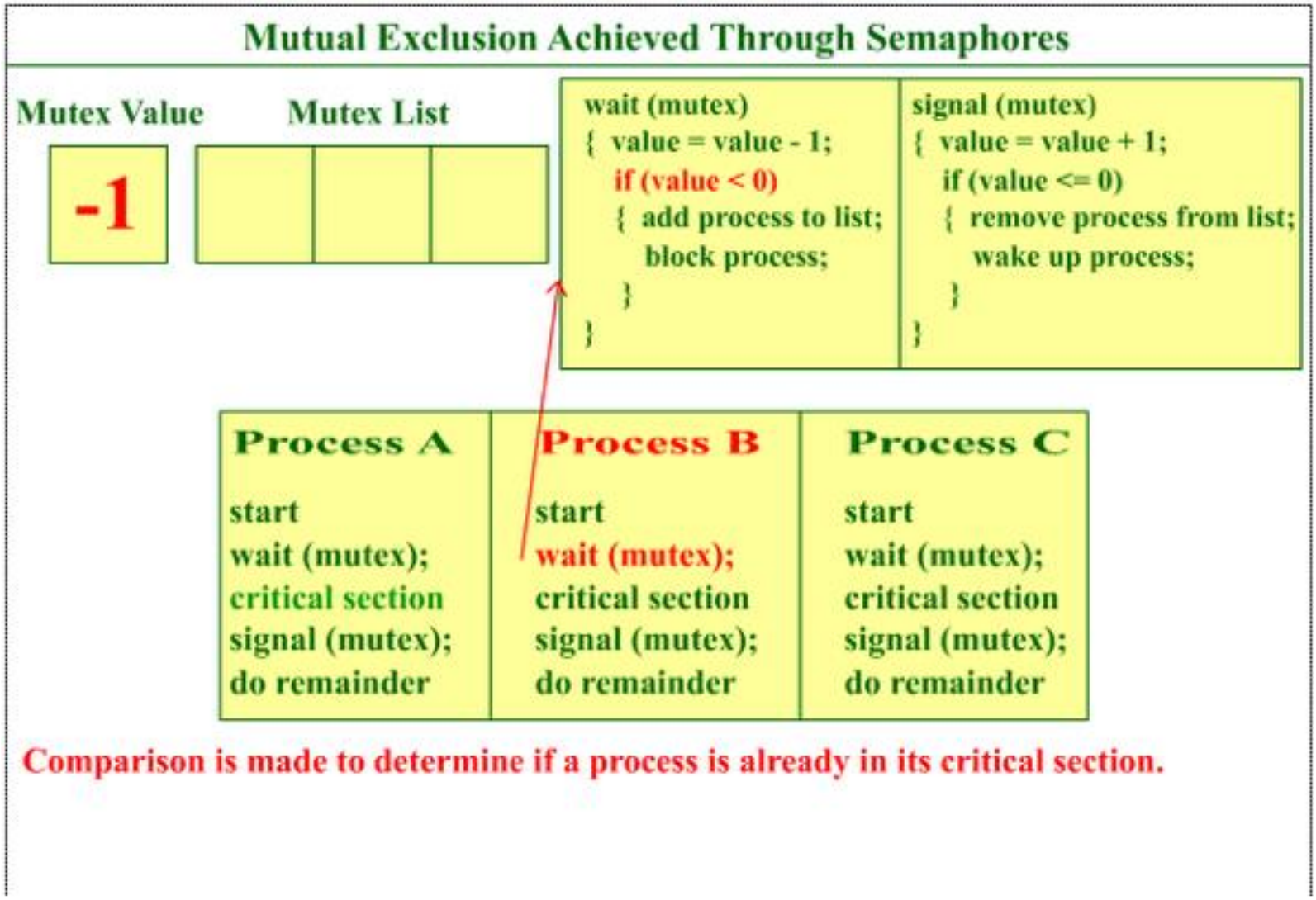
Mutual Exclusion with Semaphores (8/34)



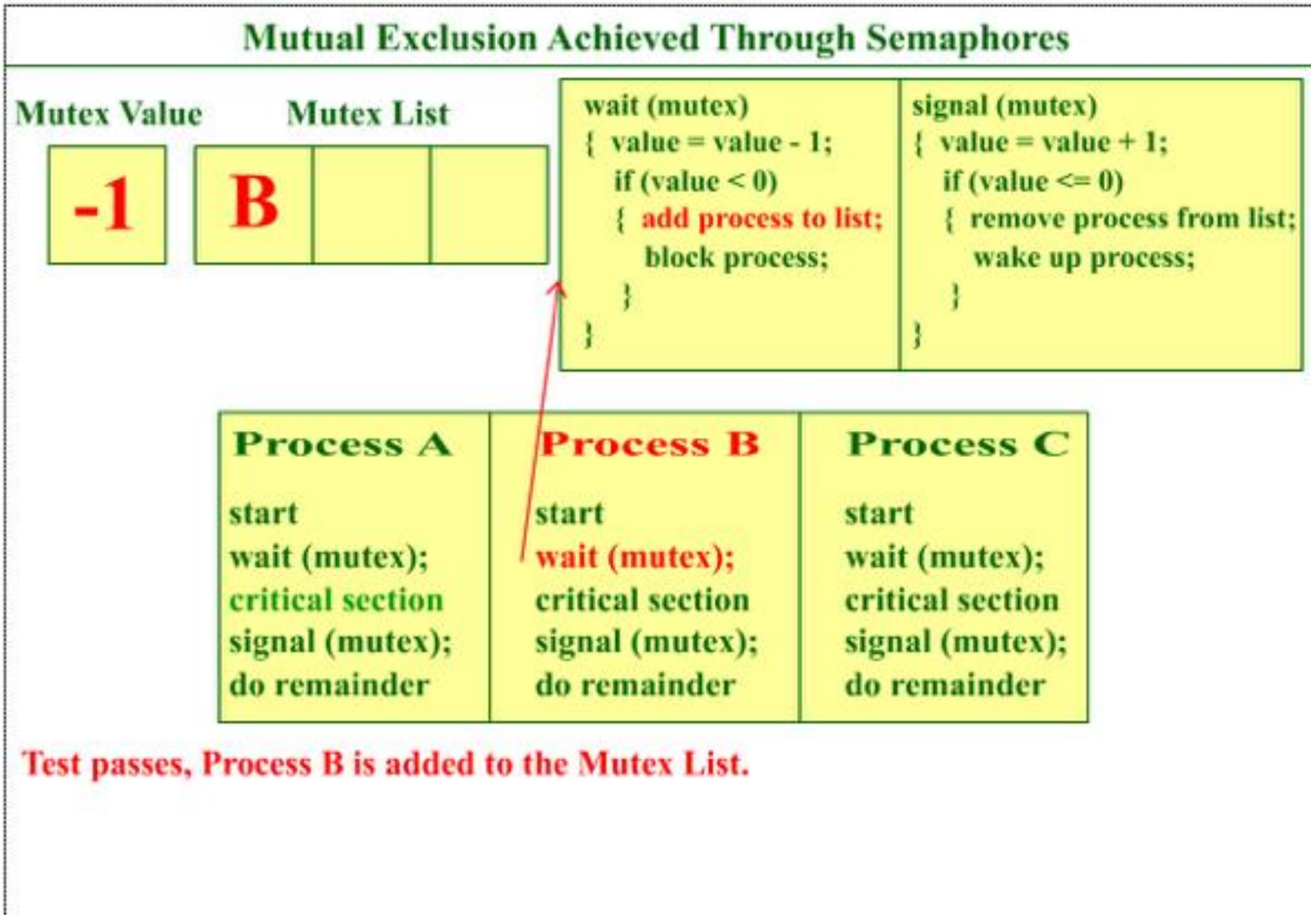
Mutual Exclusion with Semaphores (9/34)



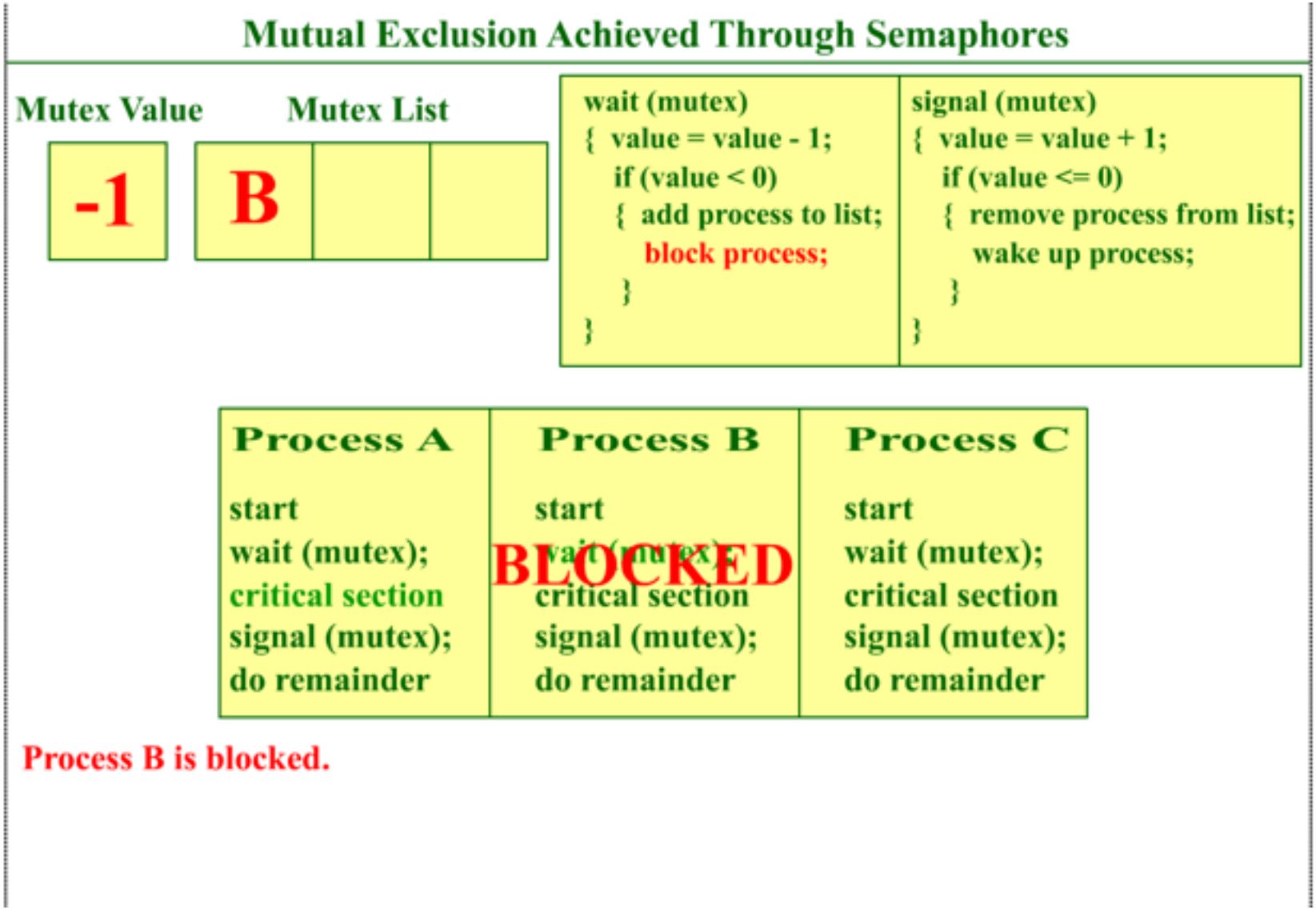
Mutual Exclusion with Semaphores (10/34)



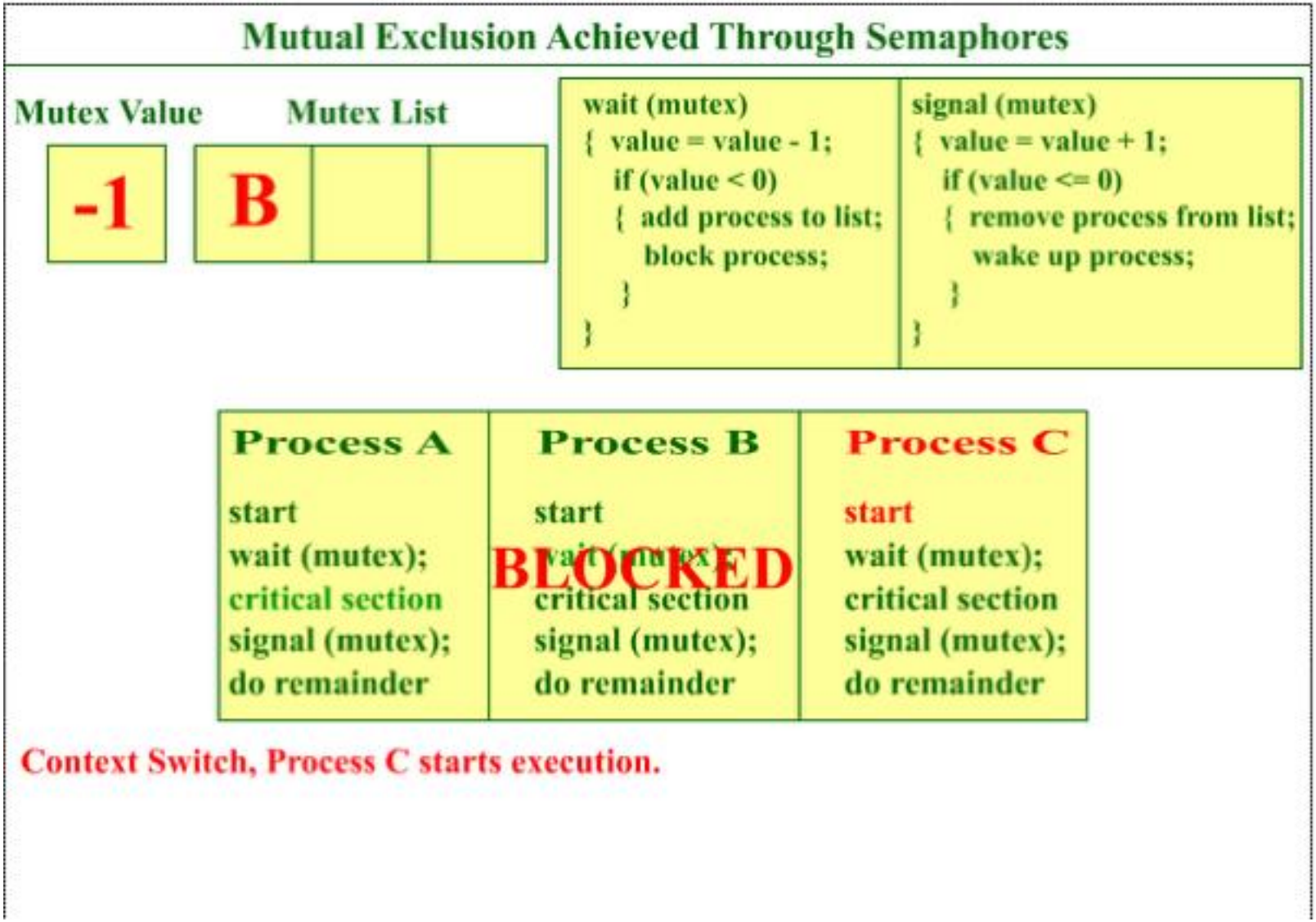
Mutual Exclusion with Semaphores (11/34)



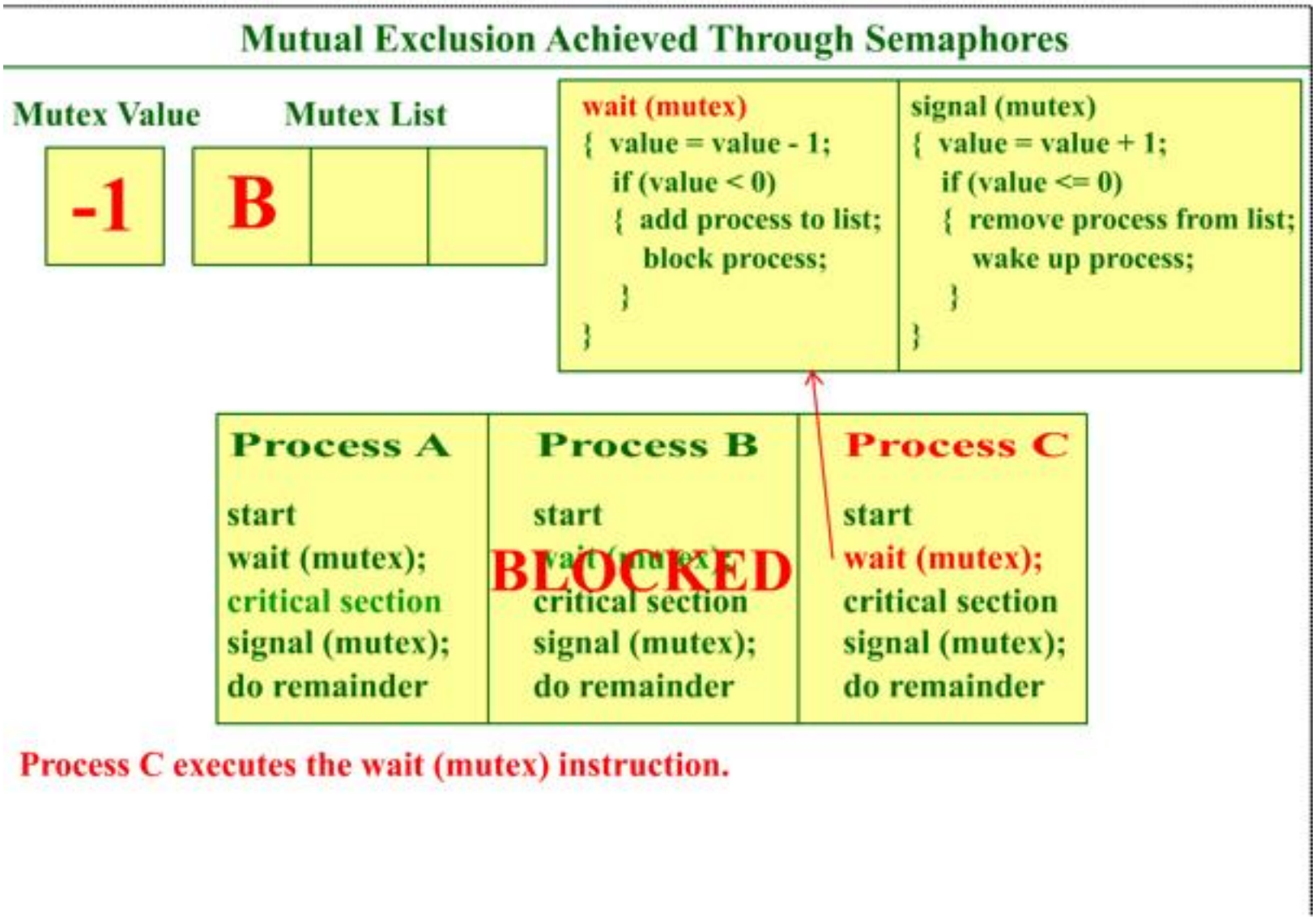
Mutual Exclusion with Semaphores (12/34)



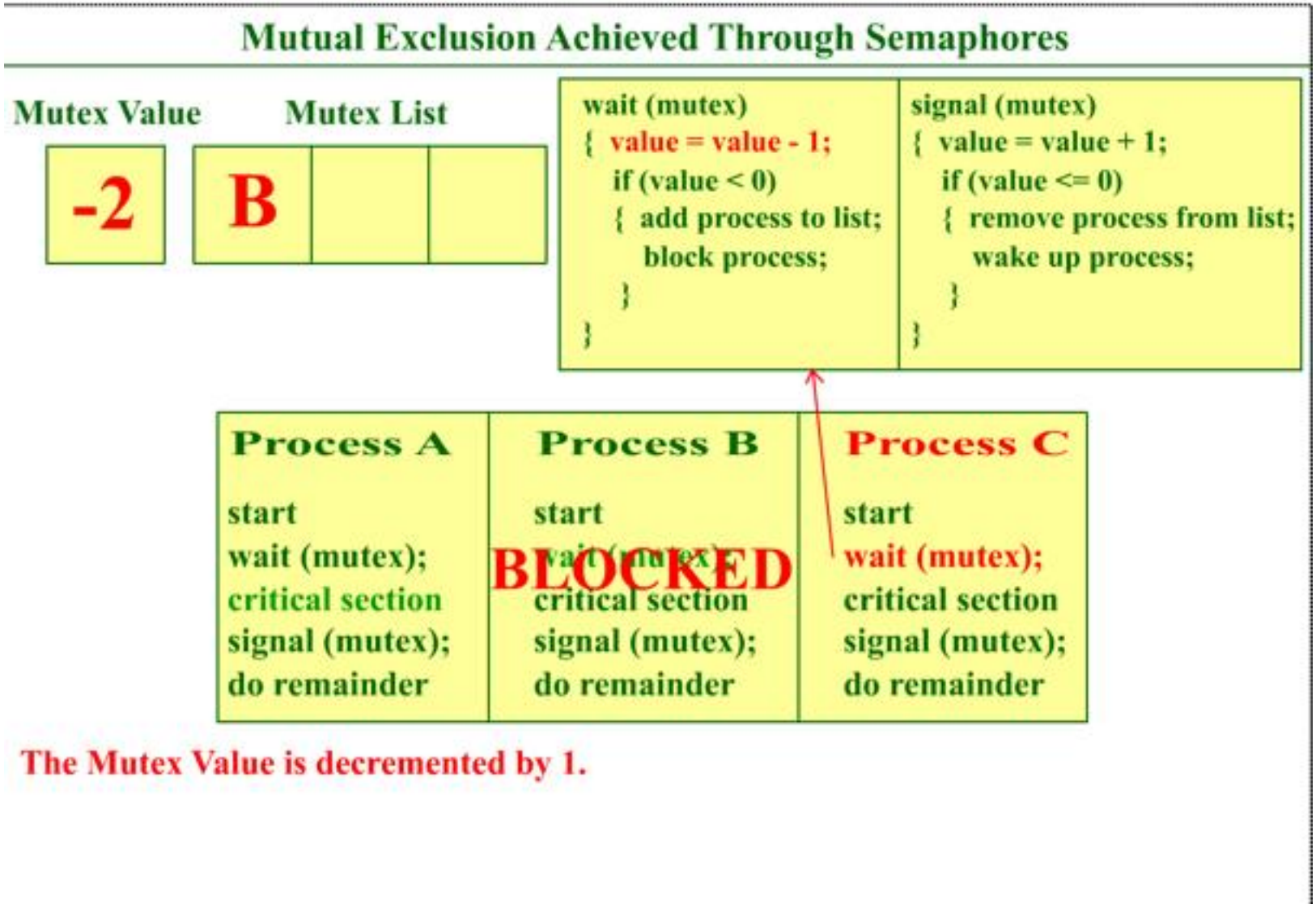
Mutual Exclusion with Semaphores (13/34)



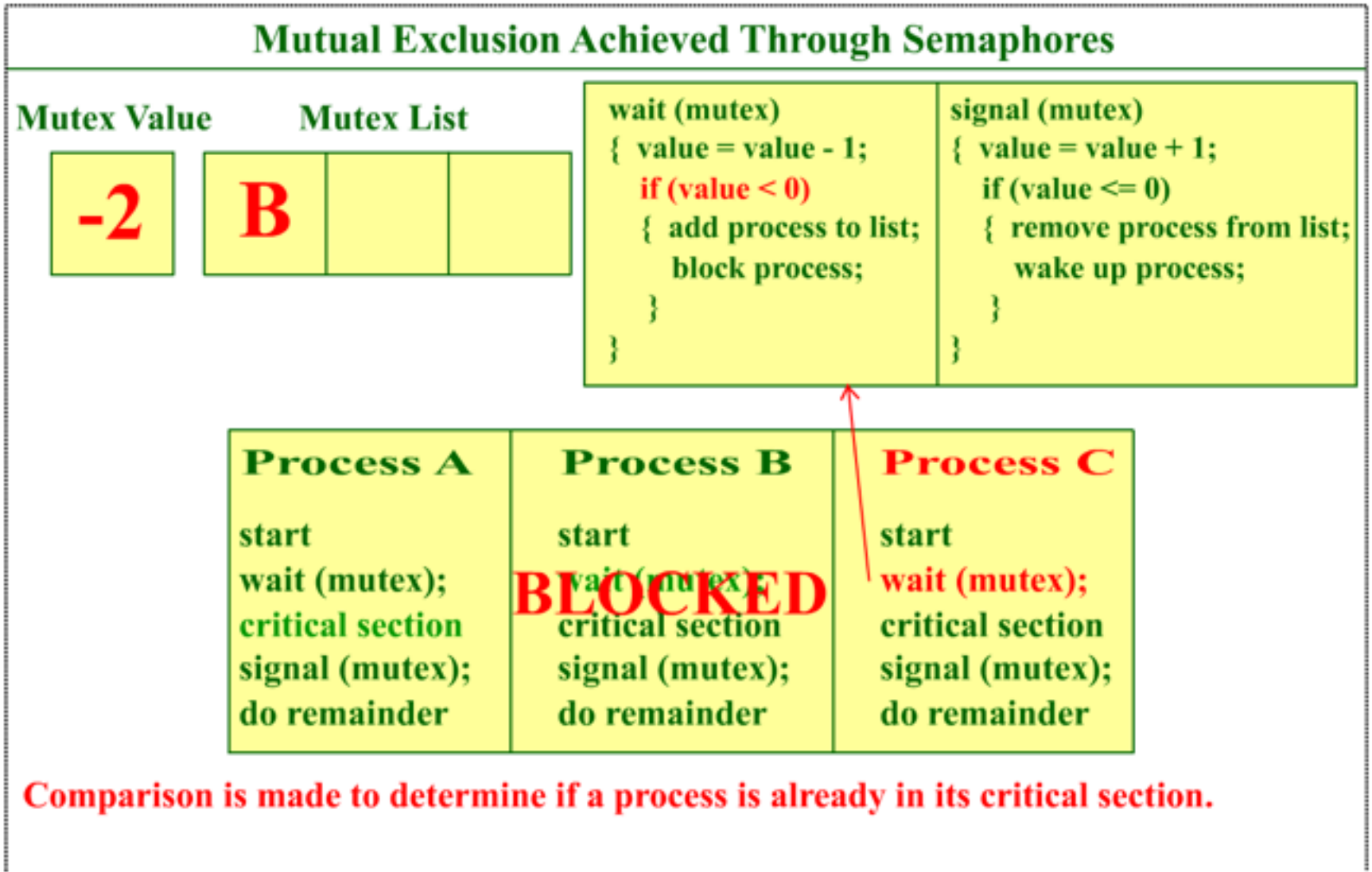
Mutual Exclusion with Semaphores (14/34)



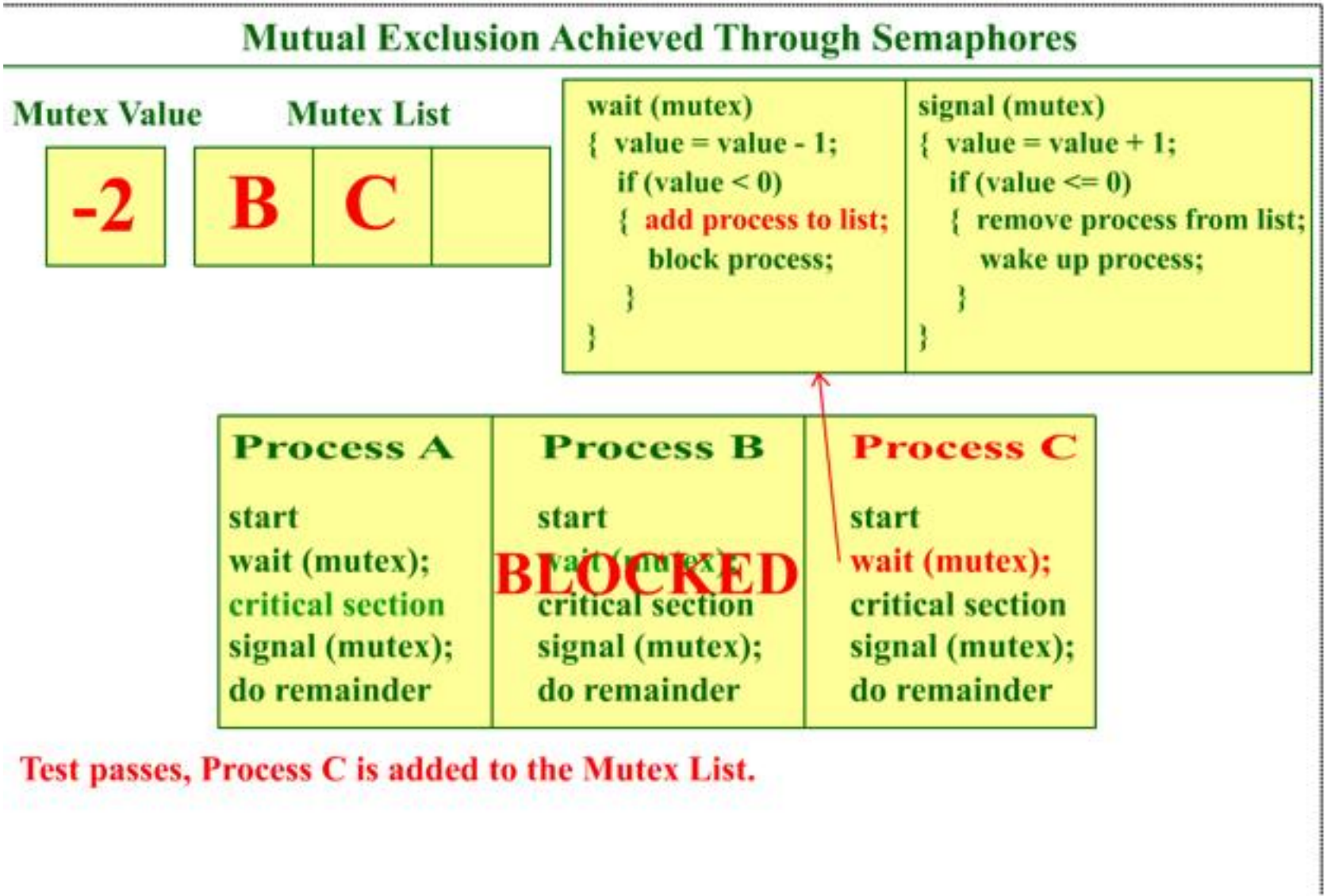
Mutual Exclusion with Semaphores (15/34)



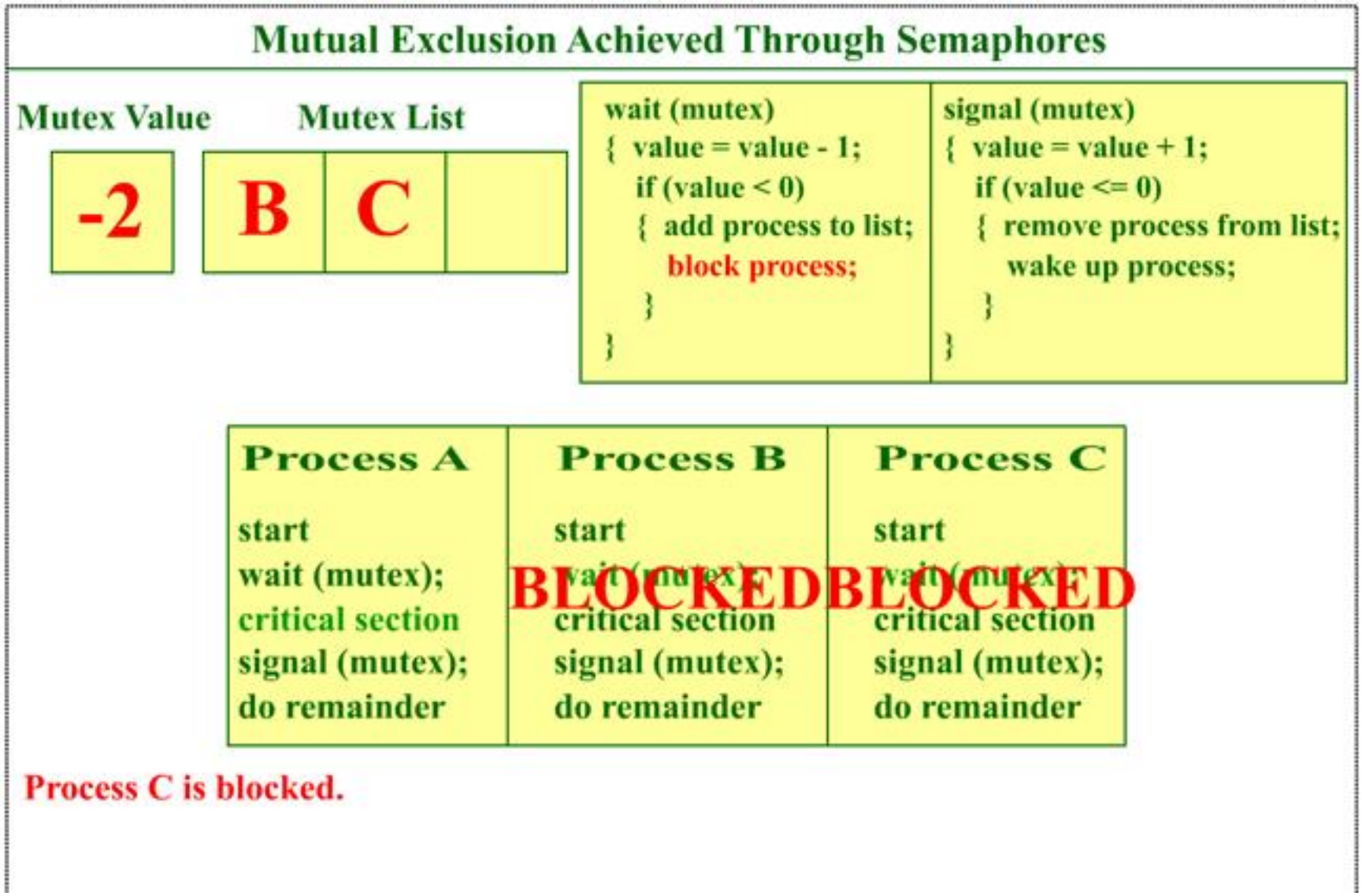
Mutual Exclusion with Semaphores (16/34)



Mutual Exclusion with Semaphores (17/34)

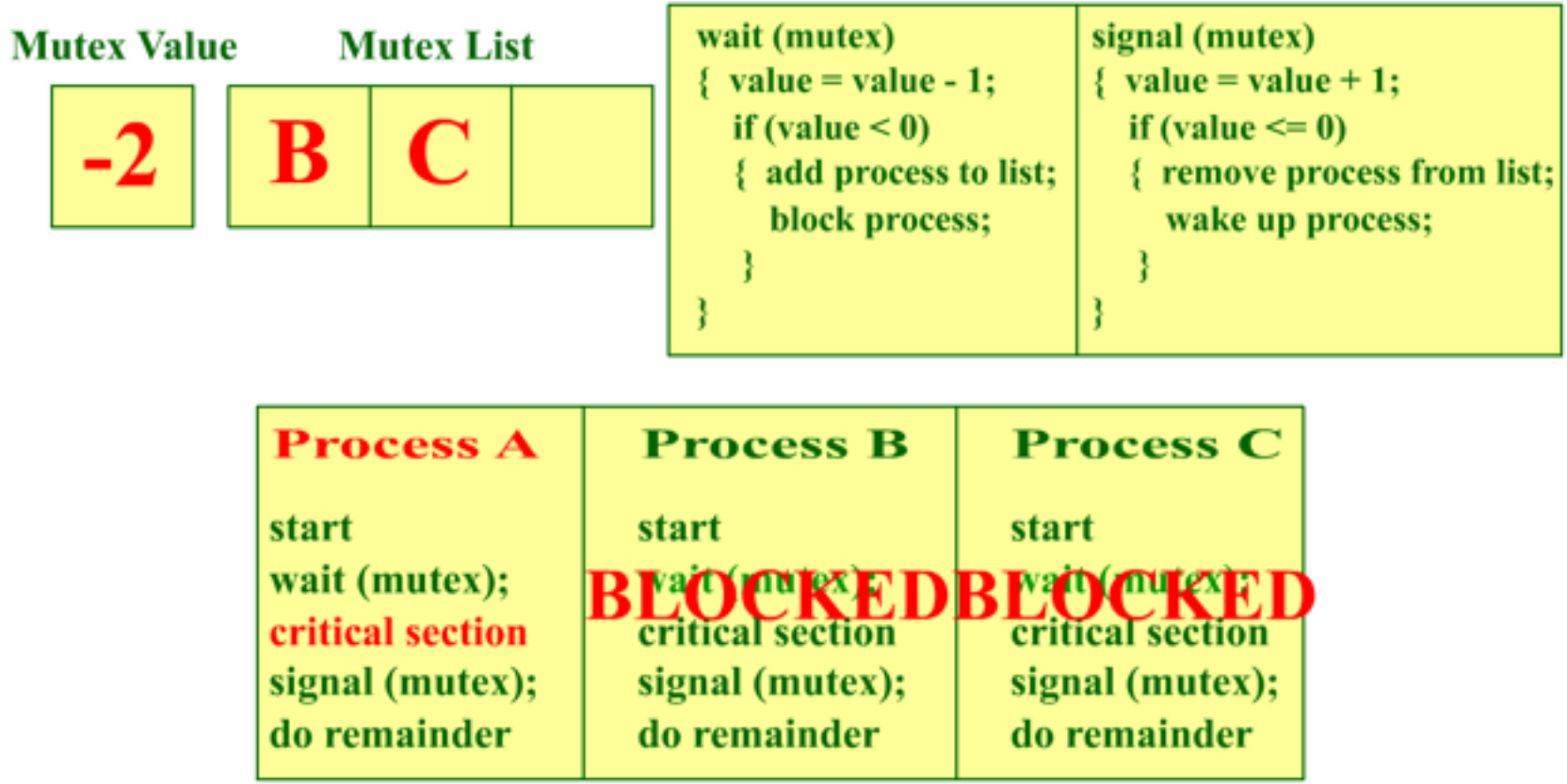


Mutual Exclusion with Semaphores (18/34)



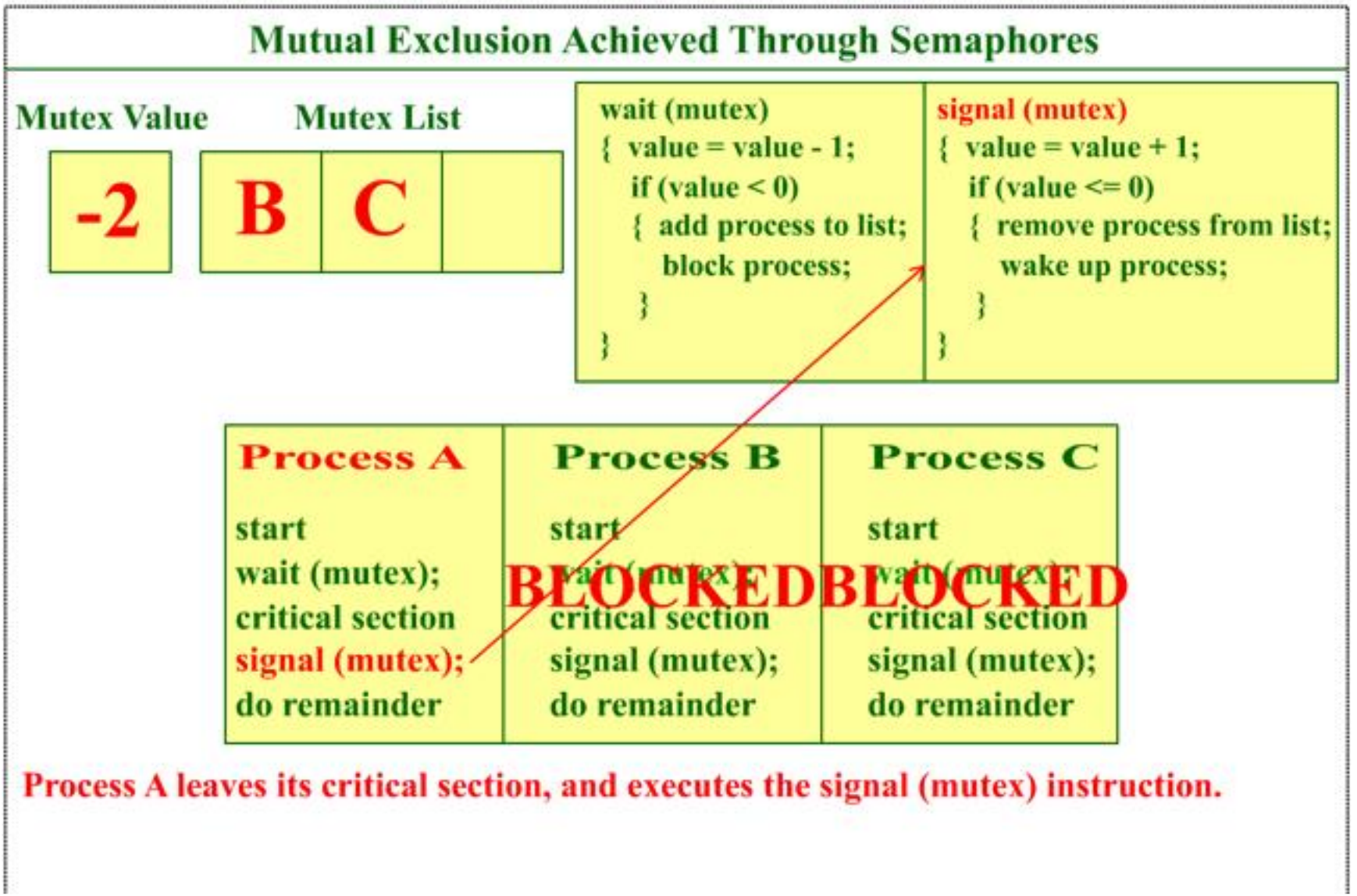
Mutual Exclusion with Semaphores (19/34)

Mutual Exclusion Achieved Through Semaphores

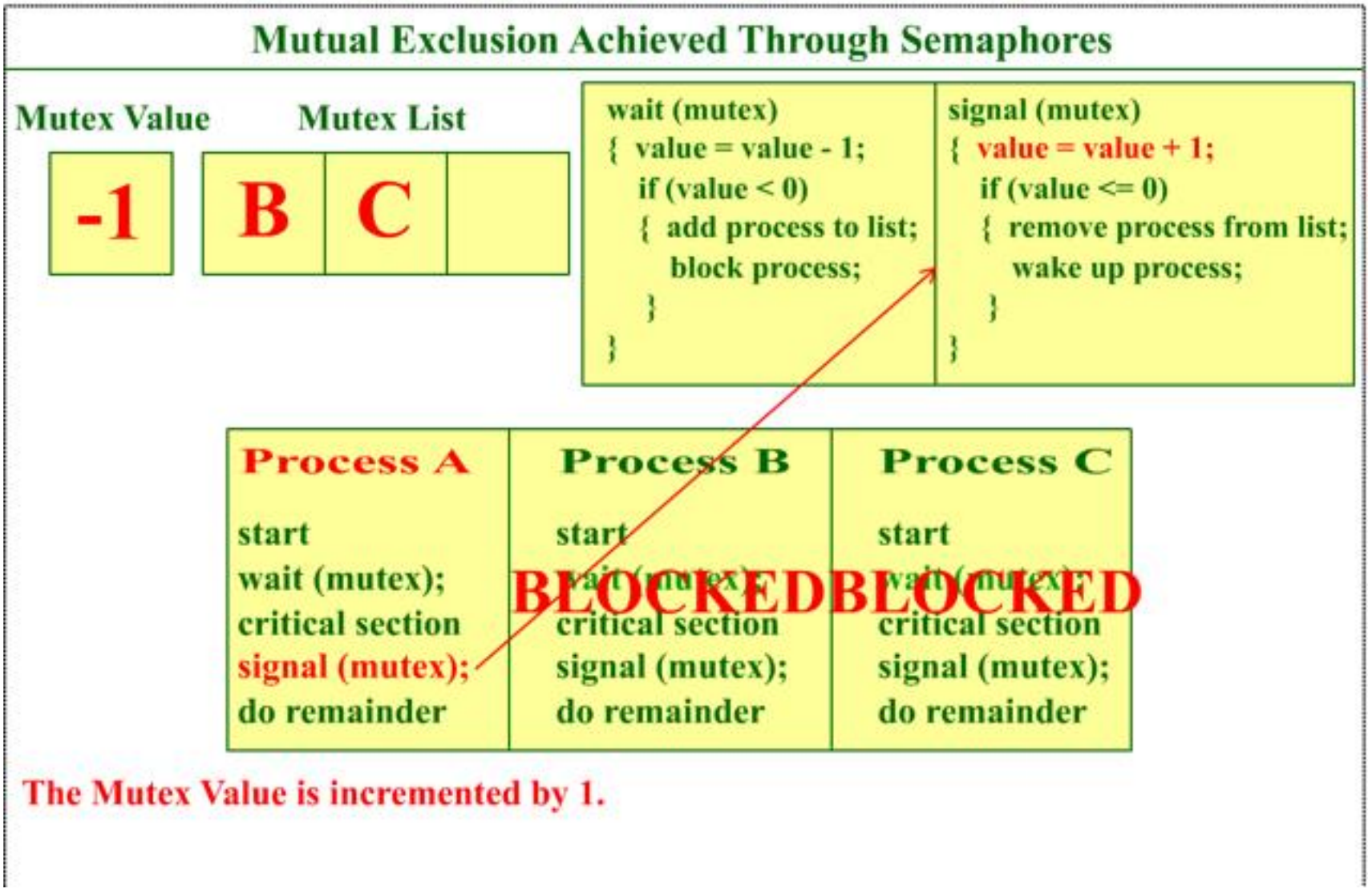


Context Switch, Process A resumes execution.

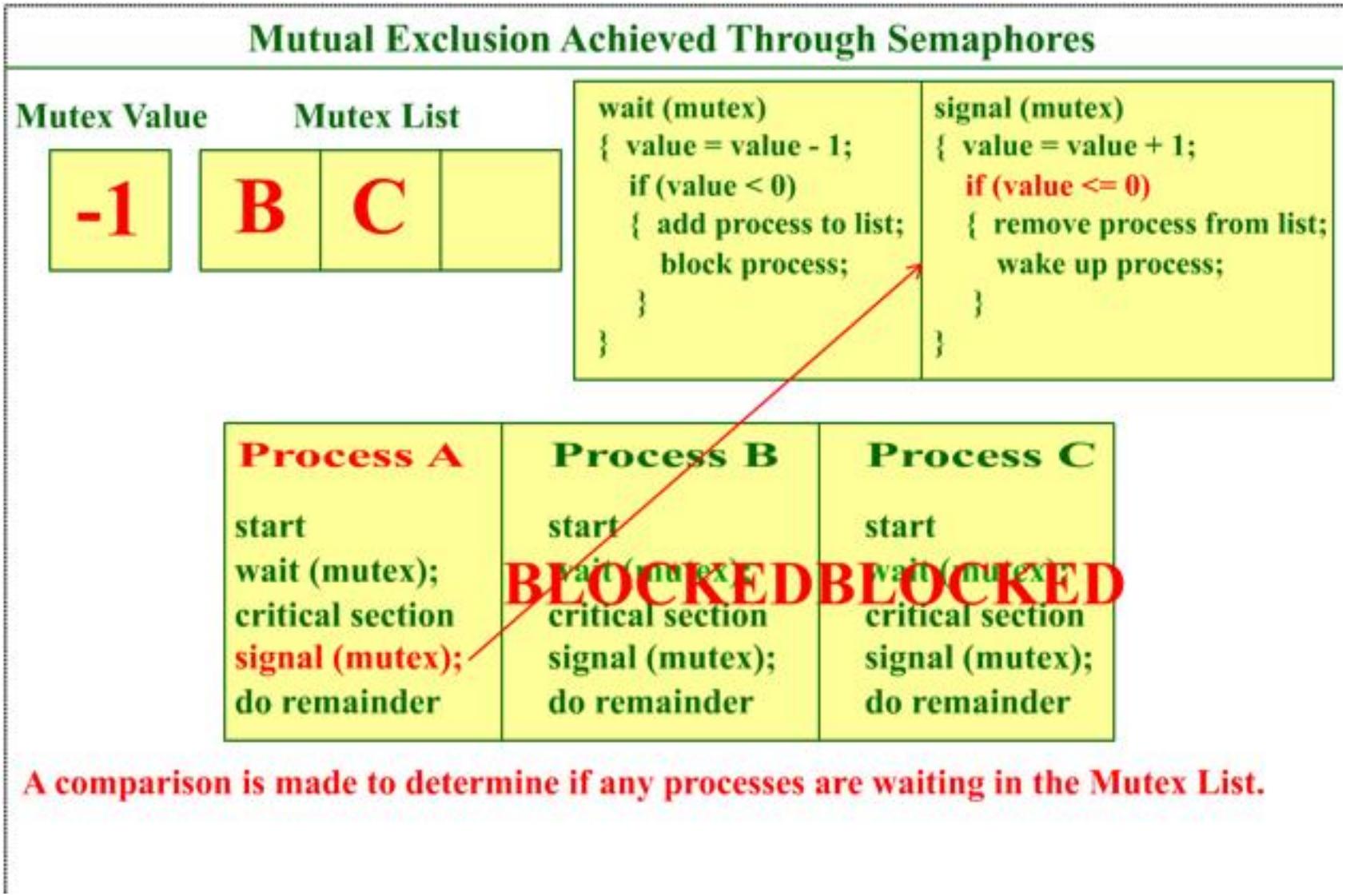
Mutual Exclusion with Semaphores (20/34)



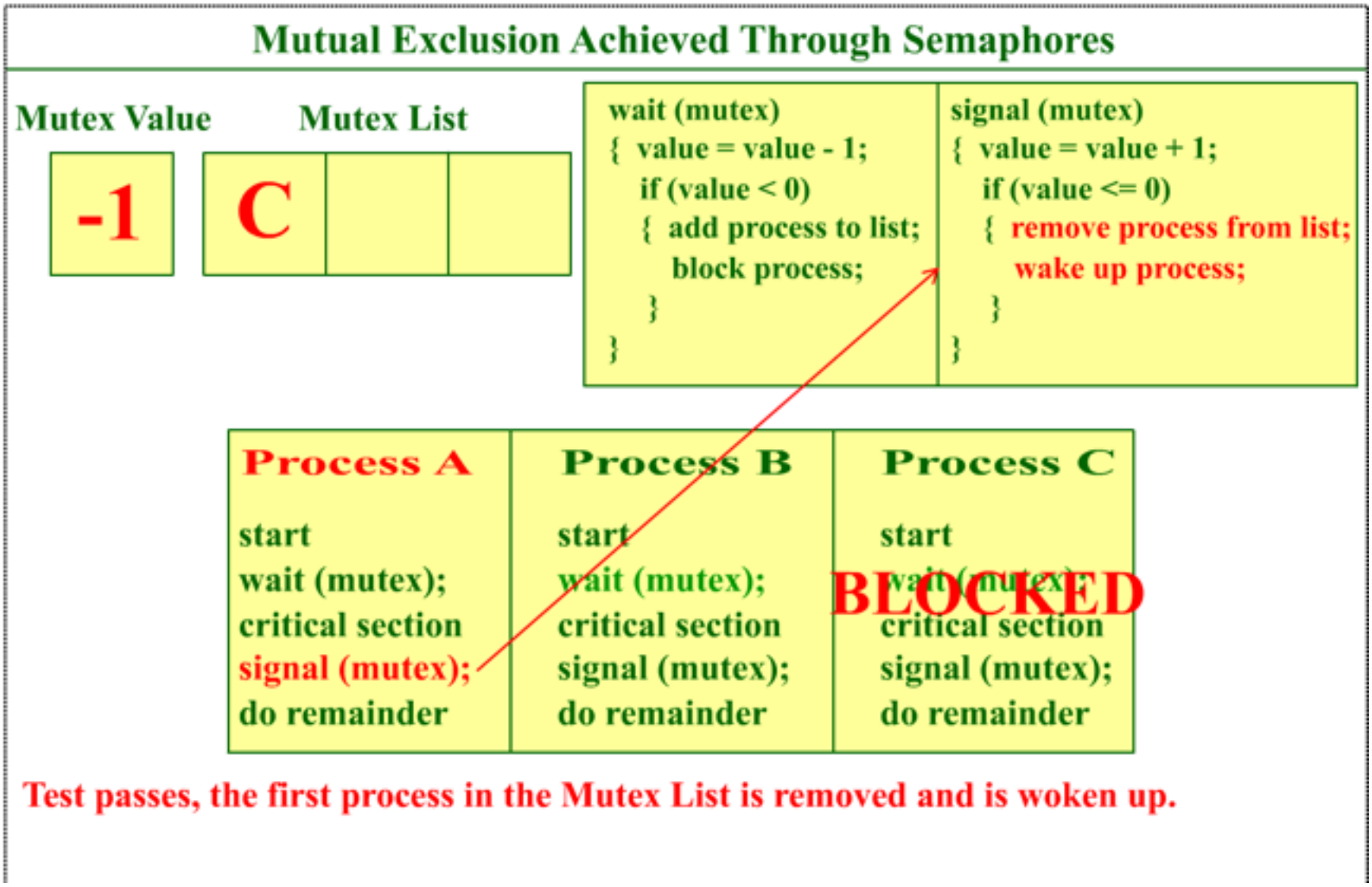
Mutual Exclusion with Semaphores (21/34)



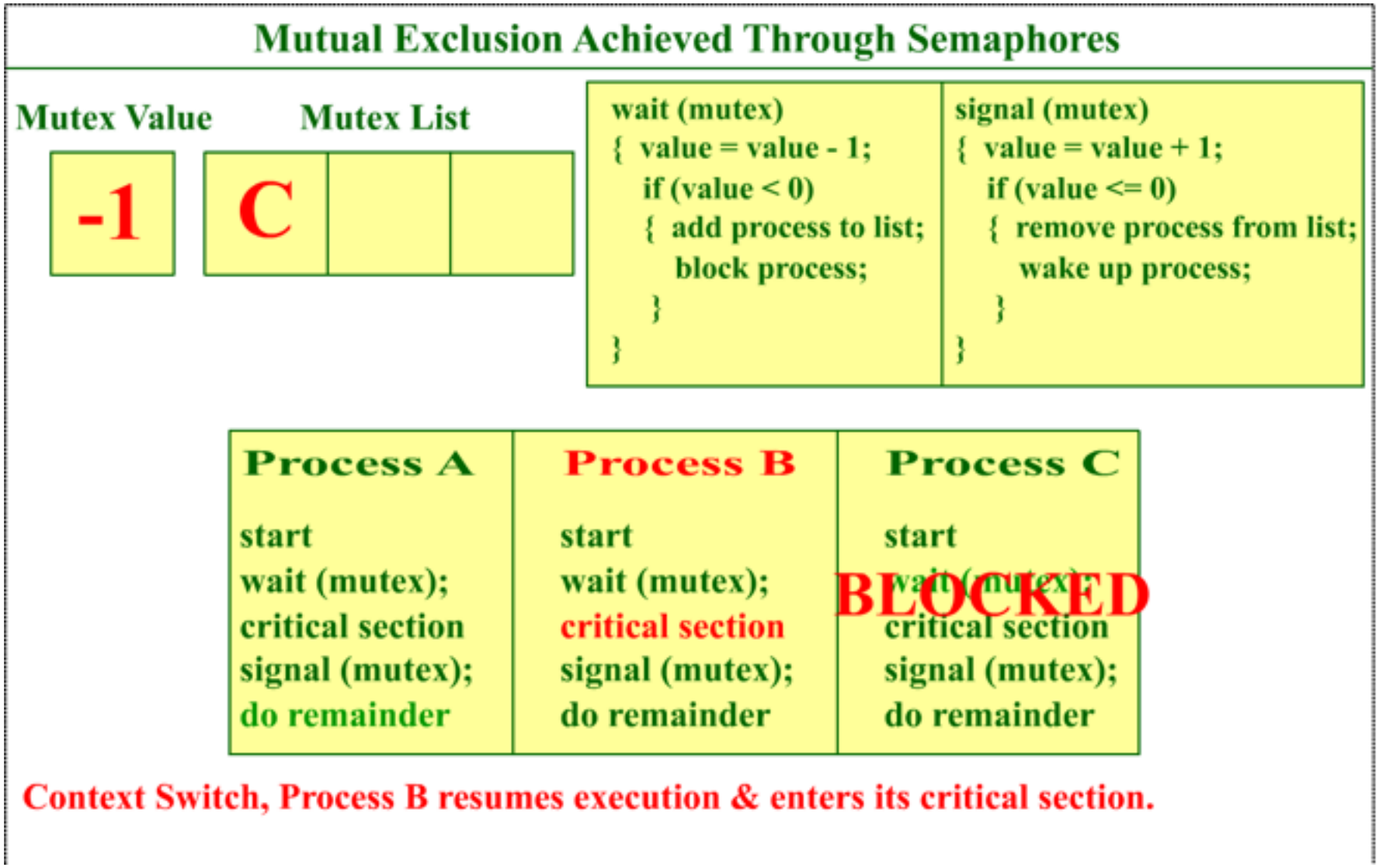
Mutual Exclusion with Semaphores (22/34)



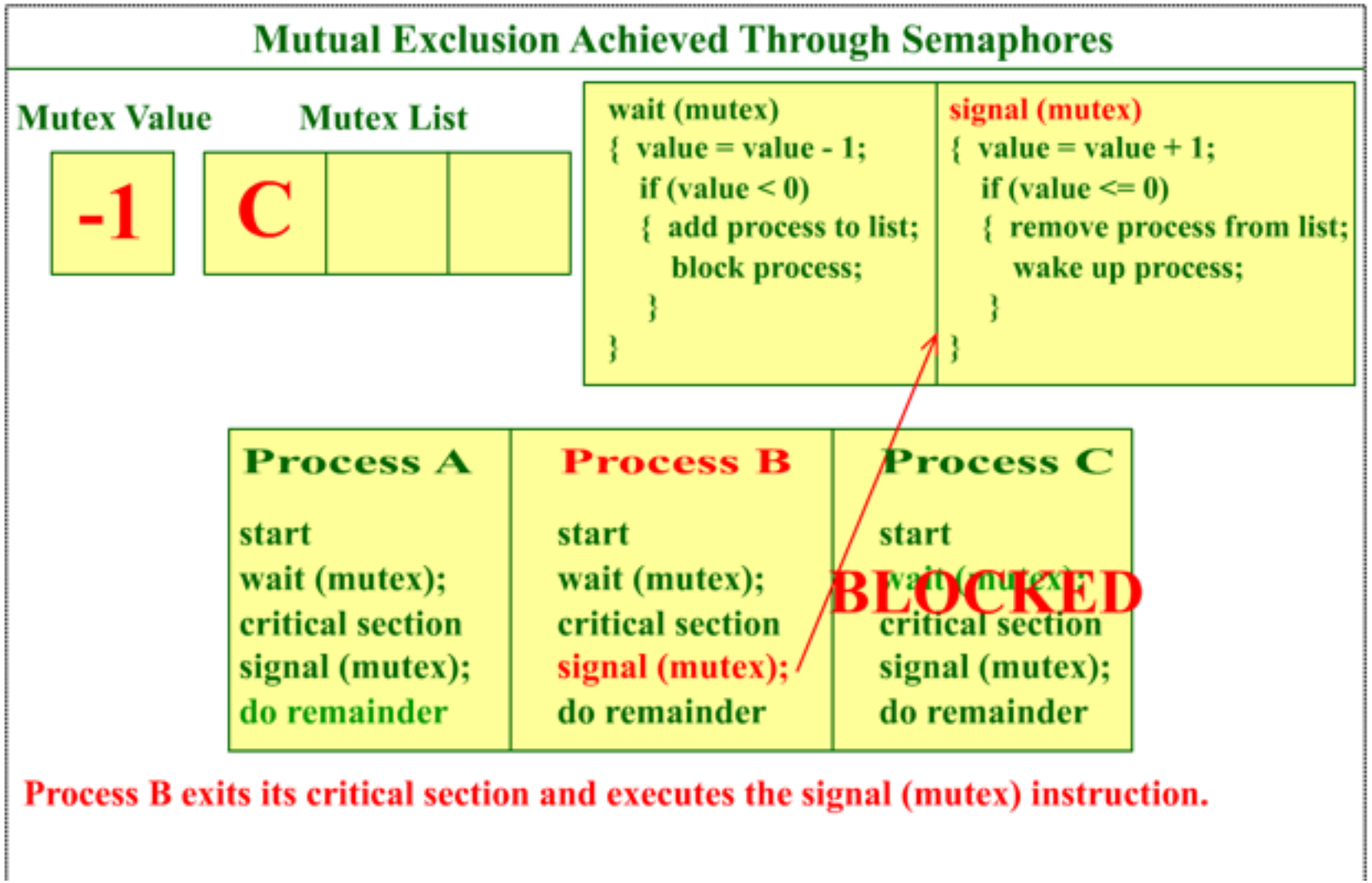
Mutual Exclusion with Semaphores (23/34)



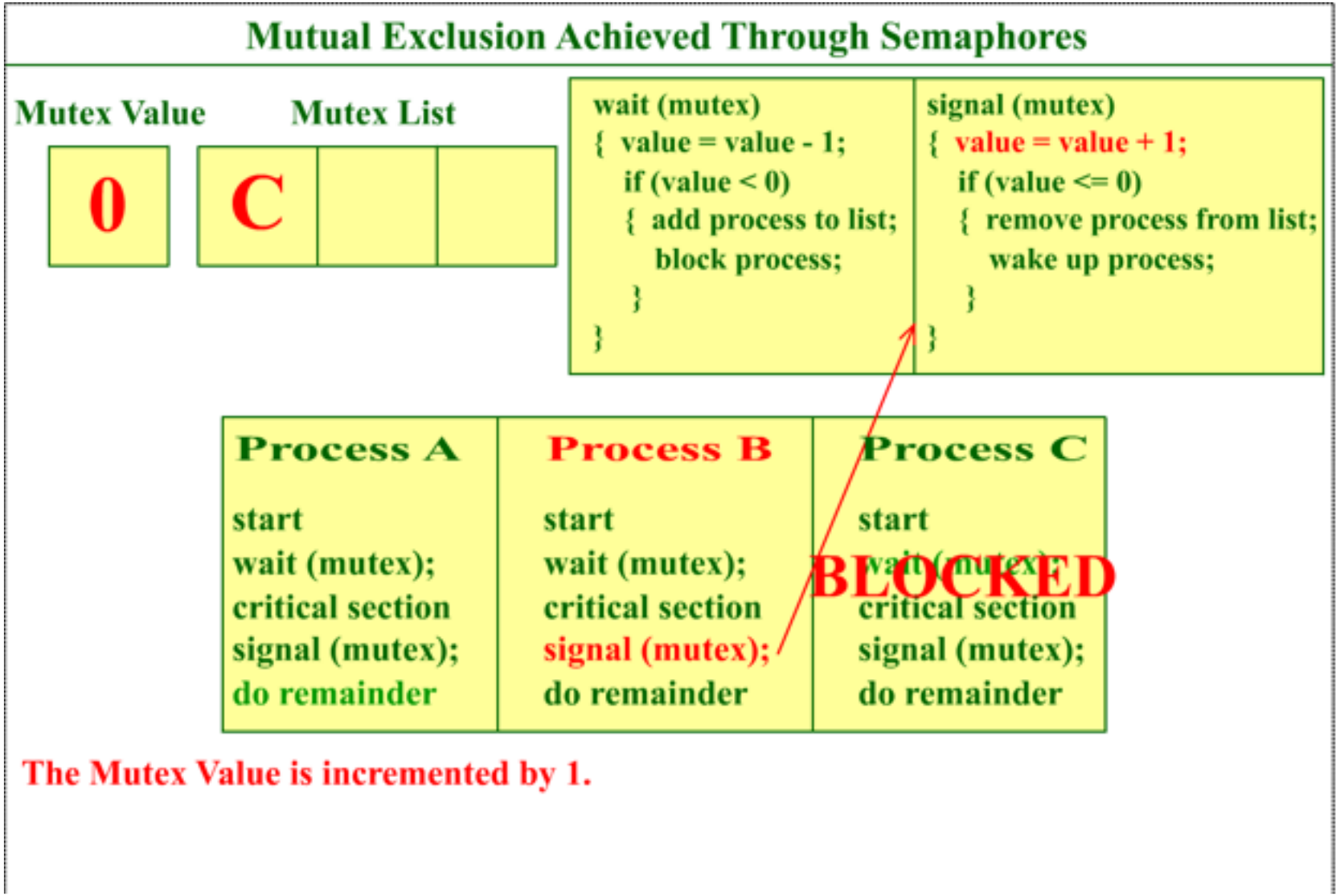
Mutual Exclusion through Semaphores (24/34)



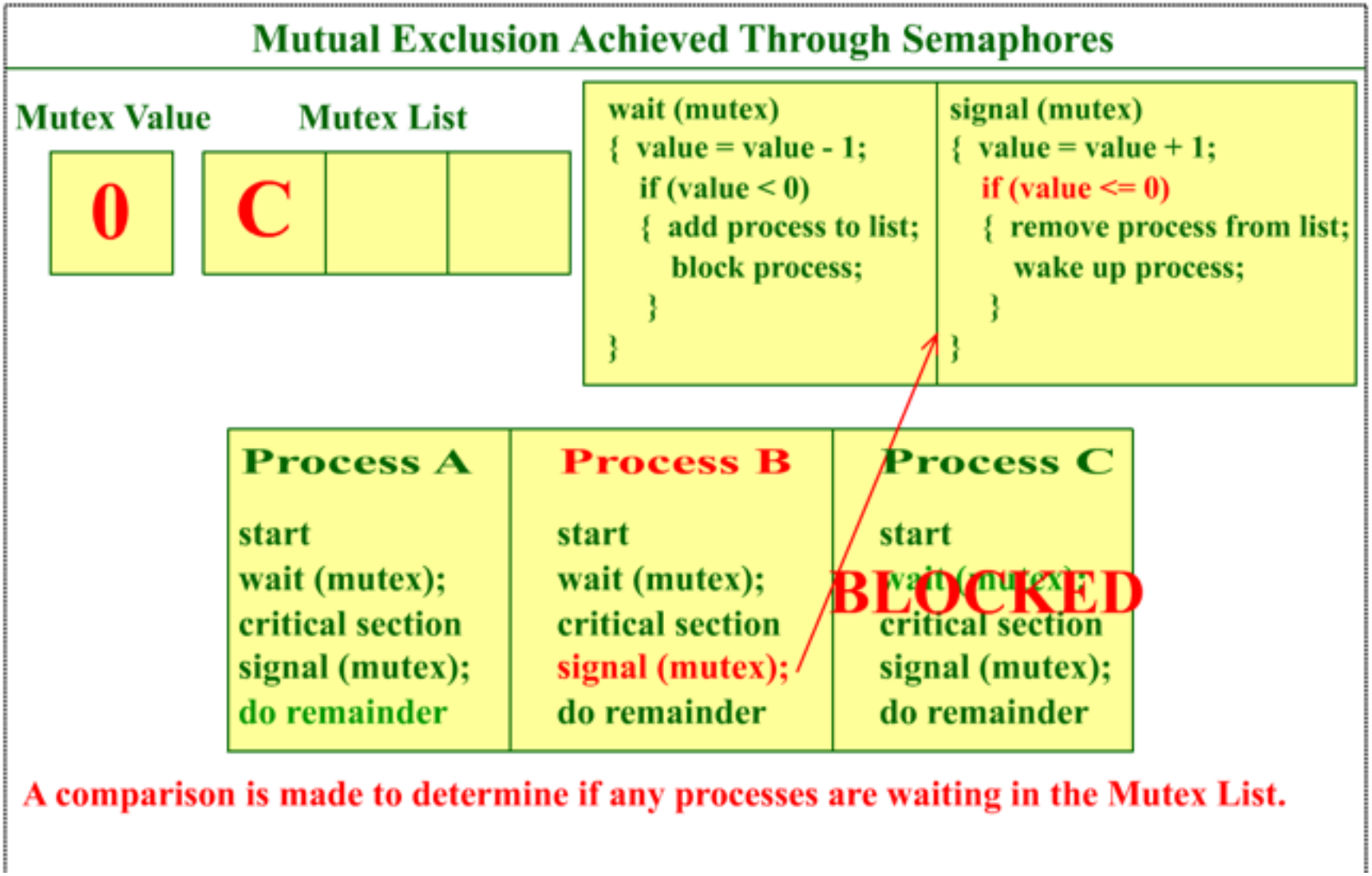
Mutual Exclusion through Semaphores(25/34)



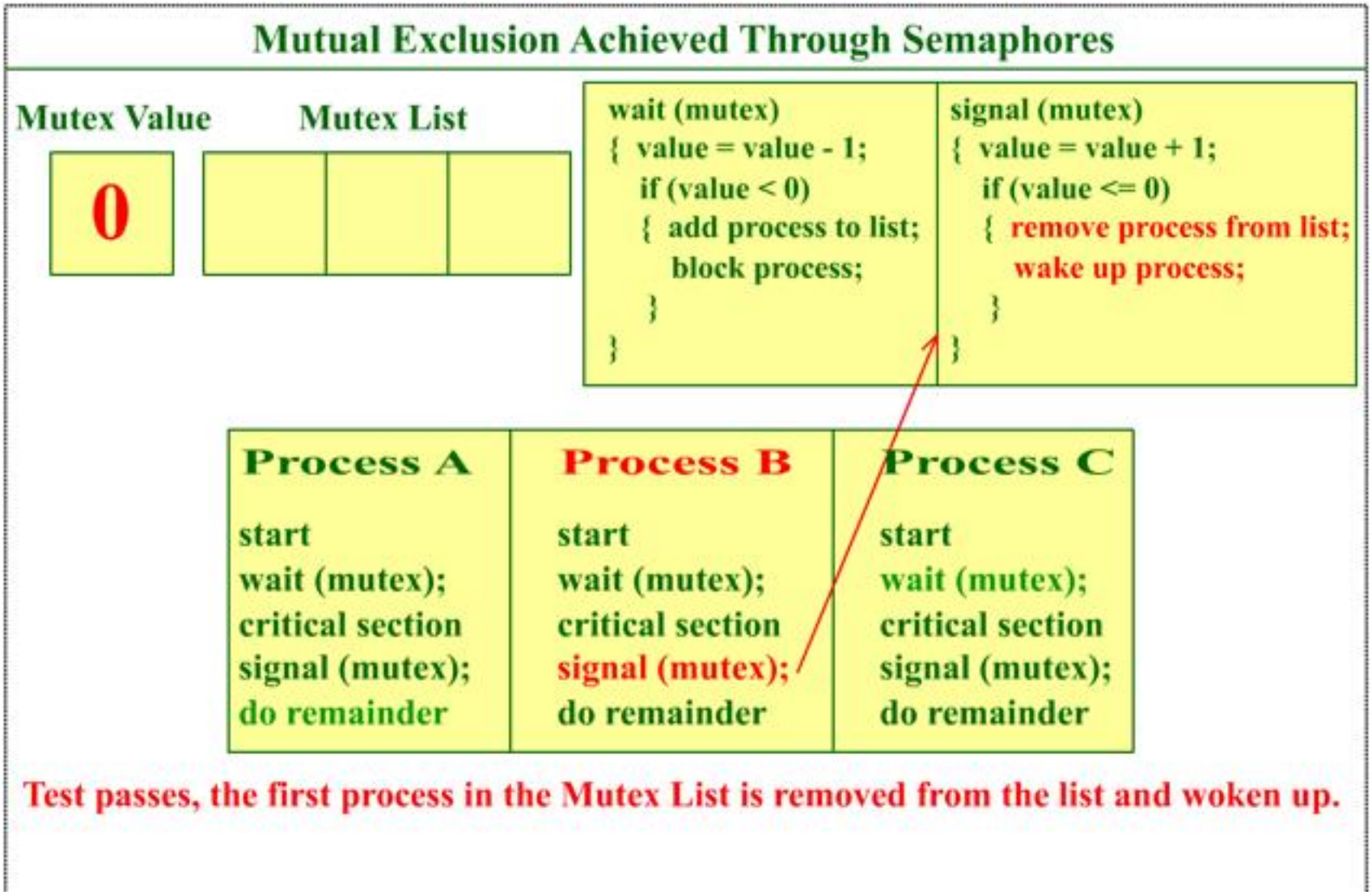
Mutual Exclusion through Semaphores(26/34)



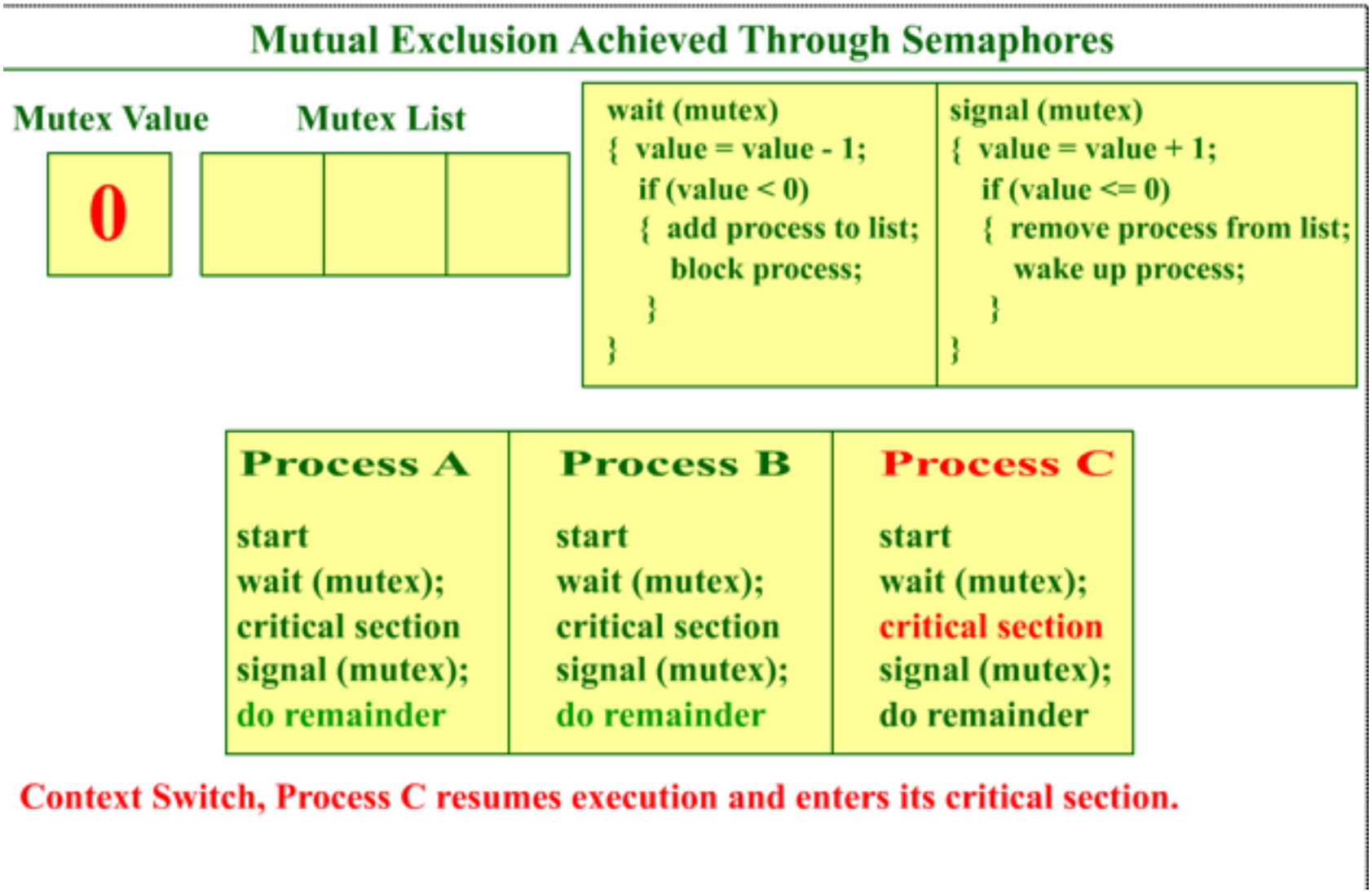
Mutual Exclusion through Semaphores(27/34)



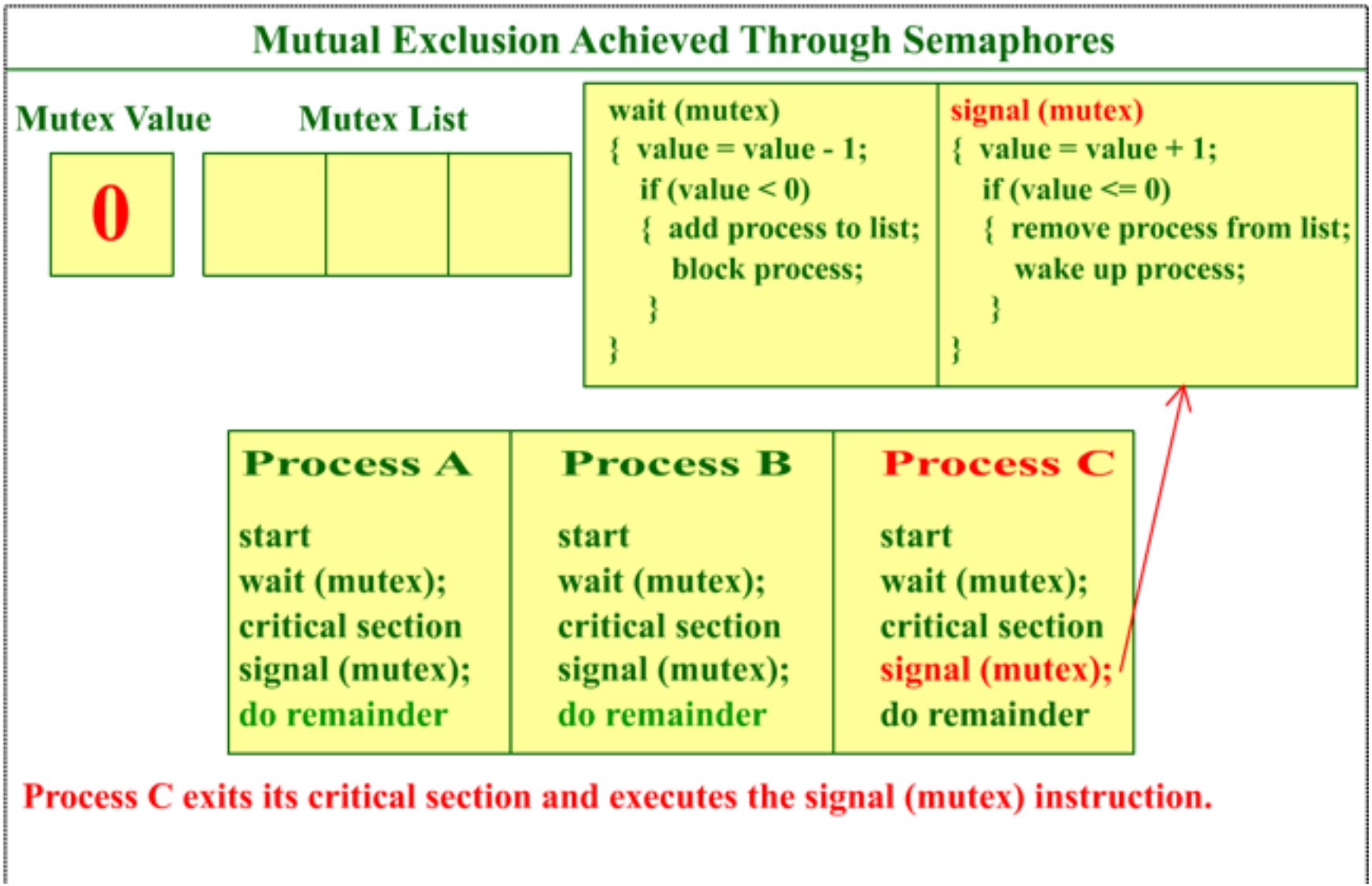
Mutual Exclusion through Semaphores(28/34)



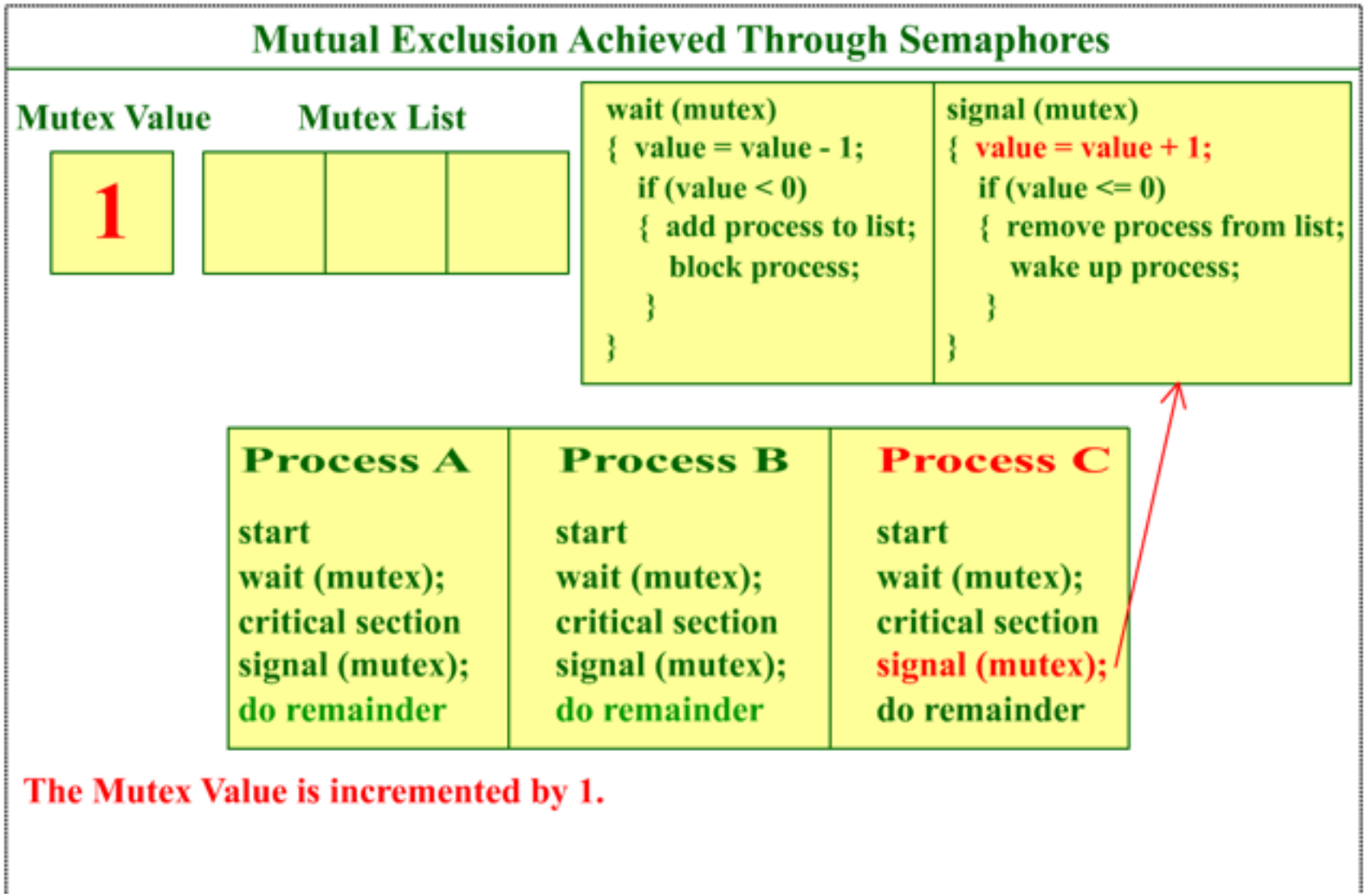
Mutual Exclusion through Semaphores(29/34)



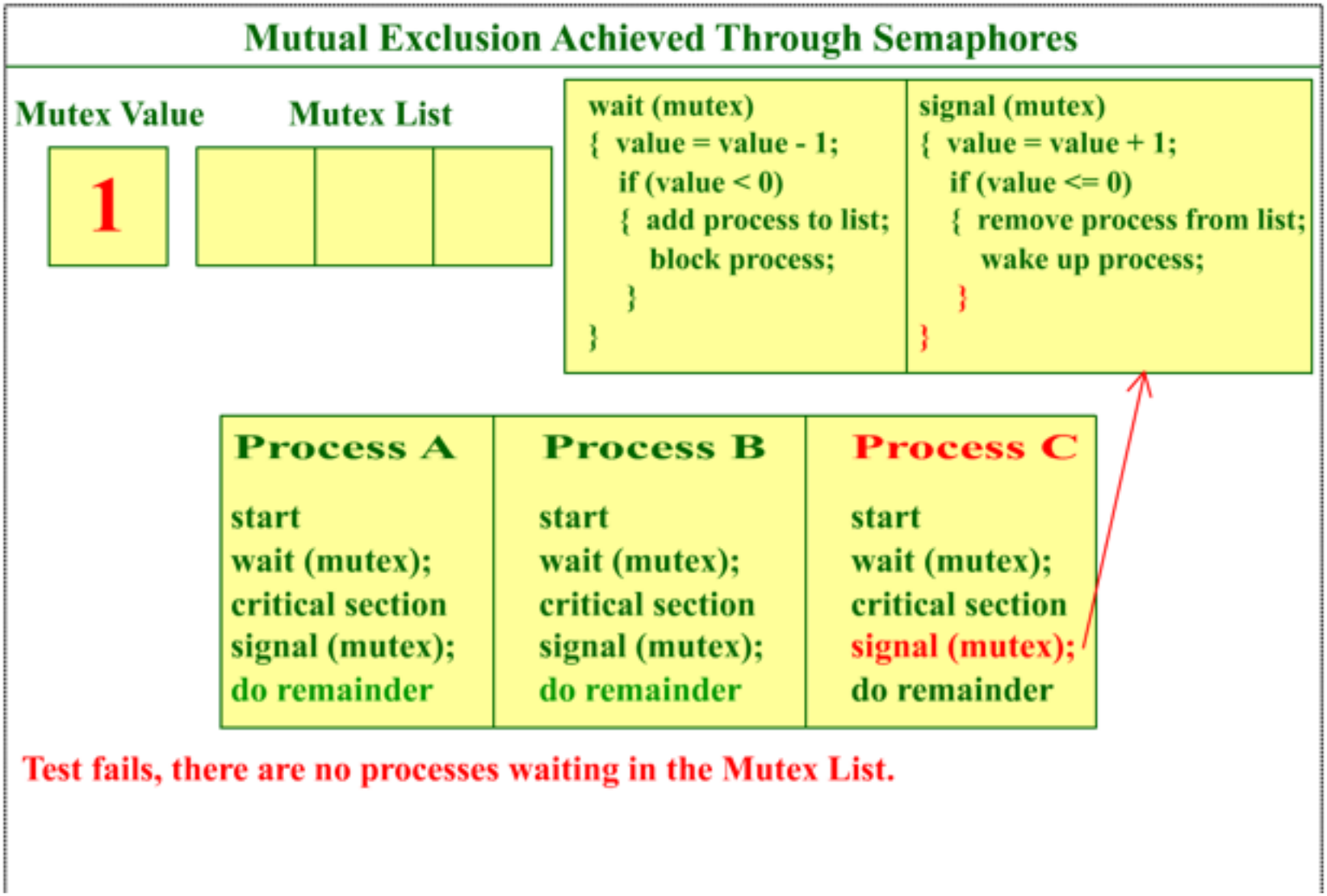
Mutual Exclusion through Semaphores(30/34)



Mutual Exclusion through Semaphores(31/34)



Mutual Exclusion through Semaphores(32/34)



Mutual Exclusion through Semaphores(33/34)

Mutual Exclusion Achieved Through Semaphores

Mutex Value

0

Mutex List

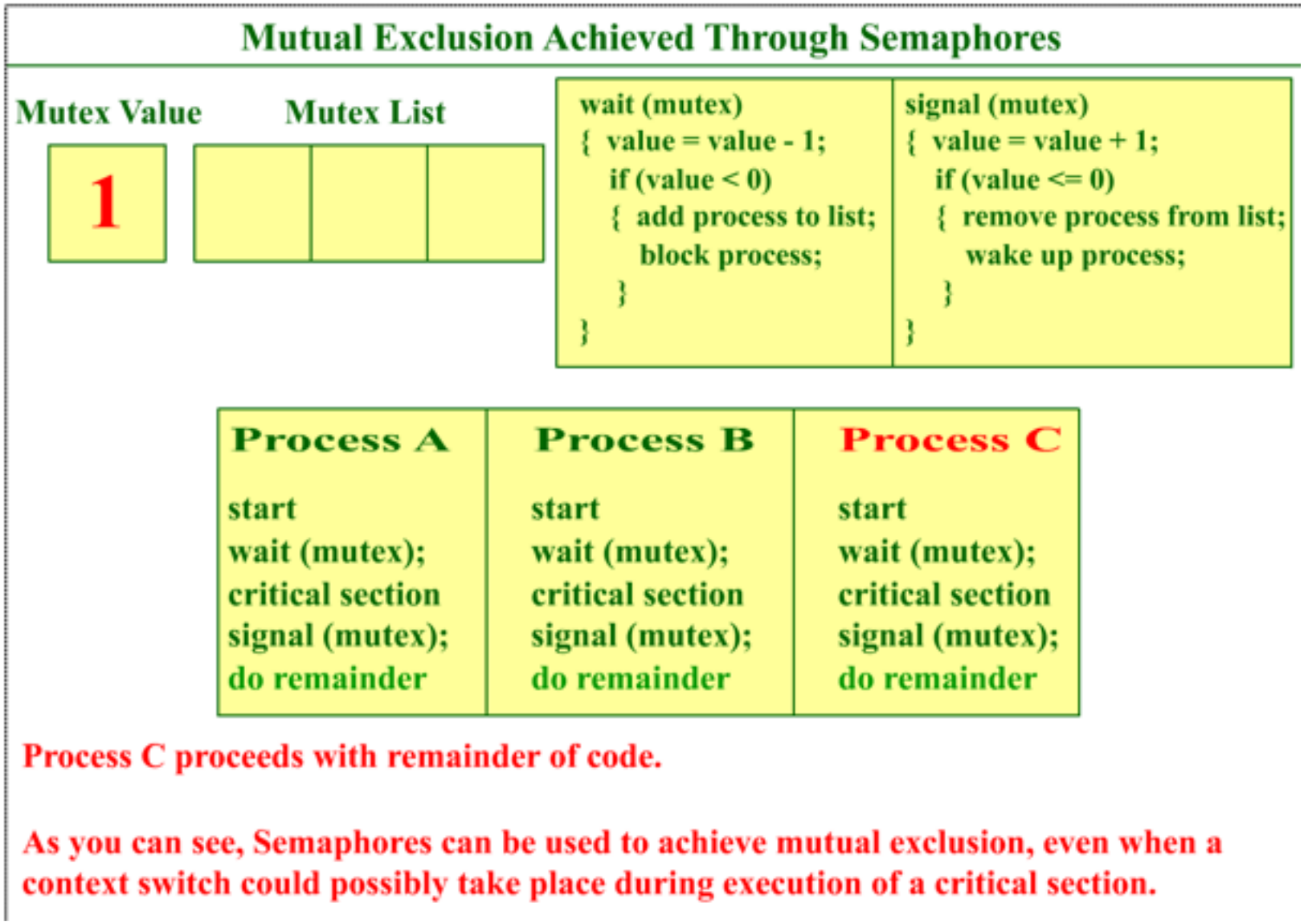
```
wait (mutex)
{  value = value - 1;
   if (value < 0)
   {  add process to list;
      block process;
   }
}
```

```
signal (mutex)
{  value = value + 1;
   if (value <= 0)
   {  remove process from list;
      wake up process;
   }
}
```

Process A	Process B	Process C
start	start	start
wait (mutex);	wait (mutex);	wait (mutex);
critical section	critical section	critical section
signal (mutex);	signal (mutex);	signal (mutex);
do remainder	do remainder	do remainder

Process A is allowed to enter its critical section.

Mutual Exclusion through Semaphores(34/34)



Process Synchronization with Semaphore(1/4)

- **Example 1:**
- Suppose We have 2 processes: ***P1*** and ***P2***
  - where ***P1*** contains code block **B1** and ***P2*** contains code block B2
  - We want to ensure code block B1 in ***P1*** executes before code block B2 in ***P2***?

- **Use semaphores to synchronize the order of execution of the two processes?**

Process Synchronization with Semaphore(2/4)

- **Example 1:**
- Suppose We have 2 processes: P1 and P2
  - where P1 contains code B1 and P2 contains code B2
  - We want to ensure code block B1 in P1 executes before code block B2 in P2?
- **Use semaphores to synchronize the order of execution of the two processes?**
- **Solution:**
  - Define a semaphore “synch”
  - Initialize synch to 0

- Semaphore synch=0; //initial value of semaphore
- **Process P1**
  - B1; //executes B1
  - Signal(synch); //increment Synch
- **Process P2**
  - Wait(synch); //decrement Sync
  - B2; //executes B2

Process Synchronization with Semaphore(3/4)

- **Example 2:**
- Suppose We have 3 processes: ***P1***, ***P2*** and ***P3***
- where ***P1*** prints **A** and ***P2*** prints **B** and ***P3*** prints **C**

- We want to ensure to have **ABCABCABC**…… in output?

Process Synchronization with Semaphore(4/4)

- **Example 2:**
- Suppose We have 3 processes: ***P1***, ***P2*** and ***P3***
- where ***P1*** prints **A** and ***P2*** prints **B** and ***P3*** prints **C**
- We want to ensure to have **ABCABCABC**…… in output?

- Semaphore x=0, y=0 and z=1; //initial value of semaphore
- **Process P1**
  - Wait (z)
  - Print "A"
  - Signal(x); //release(x)
- **Process P2**
  - Wait(x);
  - Print B;
  - Signal(y)
- **Process P3**
  - Wait(y);
  - Print "C"
  - Signal(z); //release(z)

Chapter 5: Process Synchronization

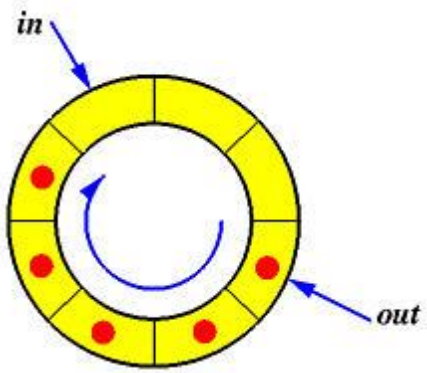
- Background
- The Critical-Section Problem
- Peterson's Solution
- Synchronization Hardware
- Mutex Locks
- Semaphores
- **Classic Problems of Synchronization**
- Monitors
- Condition Variables

Classical Problems of Synchronization

- Classical problems used to test newly-proposed synchronization schemes
  - Bounded-Buffer Problem (also called the producer consumer problem)
  - Readers and Writers Problem
  - Dining-Philosophers Problem

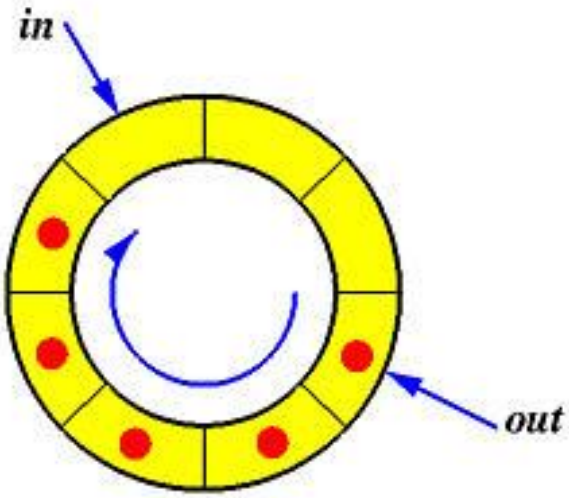
1. Bounded-Buffer Problem

- Also called producer-consumer problem
- This arises when two or more processes communicate with each other.
- Some produce data and others consume data.
- Bounded buffer: size ‘N’
  - Access entry 0… N-1, then “wrap around” to 0 again
- Producer process writes data to buffer
  - Must not write more than ‘N’ items more than consumer “consume”
- Consumer process reads data from buffer
  - Should not try to consume if there is no data
- All processes modify the same buffer
- Only one process modify buffer at the same time



1. Bounded-Buffer Problem

- Producer and Consumer share the following data:
  - ***n*** buffers, each can *hold* one item
  - Semaphore **`mutex`** initialized to the value 1; for permission access to buffer pool
  - Semaphore **`full`** initialized to the value 0; number of full buffers
  - Semaphore **`empty`** initialized to the value n; number of empty buffers



1. Bounded Buffer Problem (Cont.)

- The structure of the ***producer process***; solution using Semaphores
- Semaphore mutex=1, empty=n, full=0

```
do {
            ...
/* produce an item in
next_produced */
            ...
        wait(empty);
        wait(mutex);
            ...
//produced item
//add to the buffer
            ...
        signal(mutex);
        signal(full);
    } while (true);
```

- The structure of the ***consumer process***; solution using semaphores
- Semaphore mutex=1, empty=n, full=0

```
    do {
        wait(full);
        wait(mutex);
            ...
 /* remove an item from buffer
to next_consumed */
            ...
        signal(mutex);
        signal(empty);
            ...

            ...
    } while (true);
```

2. Readers-Writers Problem

- A dataset is shared among a number of concurrent processes
  - Readers – only read the dataset; they do ***not*** perform any updates
    - Never a problem if only readers access the data; no waiting necessary
  - Writers – can both read and write
    - Problem if some writer accesses data; thus, exclusive access only for writing
- **Problem** – allow multiple concurrent readers to read at the same time
  - Only one single writer can access the shared data at the same time
  - And if a writer is active, readers wait for it to finish

2. Readers-Writers Problem

- Several variations of how readers and writers are considered – all involve some form of priorities
- Solving Reader-Writer problem using ***Semaphore***
  - Semaphore **`rw_mutex`** initialized to 1 //used by reader and writer
  - Semaphore **`mutex`** initialized to 1 //Only used by reader to inc/dec read_count
  - Integer **`read_count`** initialized to 0 // number of readers

2. Readers-Writers Problem (Cont.)

- The structure of a ***writer process***

```
do {
    wait(rw_mutex);
//mutual exclusion for
writers

    ...
/* writing is performed
*/

    ...

signal(rw_mutex);
} while (true);
```

- The structure of a ***reader process***

```
do {
    wait(mutex);
read_count++; //critical section when
read_count being modified
if (read_count == 1)//if I'm first reader
wait(rw_mutex); //in case a writer is
writing
signal(mutex); //Lets another reader.

    ...
/* reading is performed */

    ...
    wait(mutex);
    read count--; //critical section
when read_count being modified
if (read_count == 0)
signal(rw_mutex); //only last reader release
lock
signal(mutex); //release lock


} while (true);
```

3. Reader-writer problem

- If there is a writer
  - Only first reader blocks on wr_mutex
  - Other readers block on mutex
- Once a reader is active, all readers get to go through
  - Which reader gets in first?
- The last reader to exit signals a writer
  - If no writer, then readers can continue
- If readers and writers are waiting on wr_mutex, and writer exits
  - Who gets to go in first?
- Why doesn't a writer need to use mutex?
- Is the previous solution fair?
- Readers can "starve" writers!
- Building a "fair" solution is tricky!

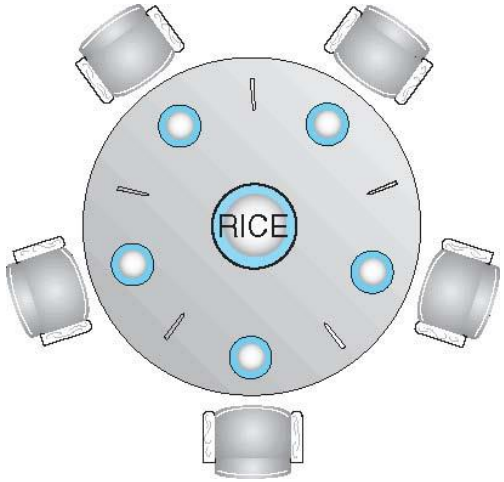
3. Dining-Philosophers Problem

- Philosophers spend their lives alternating ***thinking*** and ***eating***
- Don't interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
  - Need both chopsticks to eat, then release both when done
- In the case of ***5 philosophers***
  - Shared data
    - Bowl of rice (dataset)
    - Semaphore chopstick [5] initialized to **1**

3. Dining-Philosophers Problem Algorithm

- The structure of Philosopher ***i***:

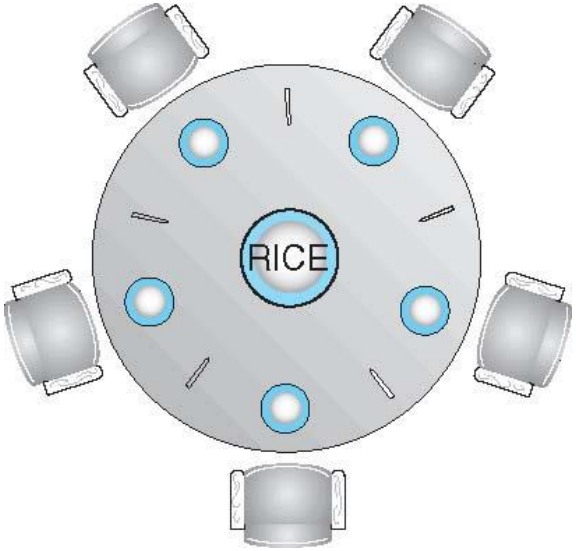
```
do {
   wait (chopstick[i] );
      wait (chopStick[ (i + 1) % 5] );

            //  eat

      signal (chopstick[i] );
      signal (chopstick[ (i + 1) % 5] );

           //  think

} while (TRUE);
```



- What is the problem with this algorithm?
  - Deadlock is possible if two neighbor philosophers grab only one chopstick and starts waiting for the other.
  - For example, i=2, then philosopher#2 grabs chopstick#2 first, and meanwhile philosopher#3 also becomes ready and grabs chopstick#3. Deadlock will occur

3. Dining-Philosophers Problem Algorithm (Cont.)

- Deadlock handling
  - Allow ***at most 4 philosophers*** to be sitting simultaneously at the table.
  - Allow a philosopher to pick up the ***forks only if both are available*** (picking must be done in a critical section).
  - Use an ***asymmetric solution*** -- an odd-numbered philosopher picks up first the left chopstick and then the right chopstick. Even-numbered philosopher picks up first the right chopstick and then the left chopstick.



Chapter 5: Process Synchronization

- Background
- The Critical-Section Problem
- Peterson's Solution
- Synchronization Hardware
- Mutex Locks
- Semaphores
- Classic Problems of Synchronization
- **Monitors**
- Condition variables

Problems with Semaphores

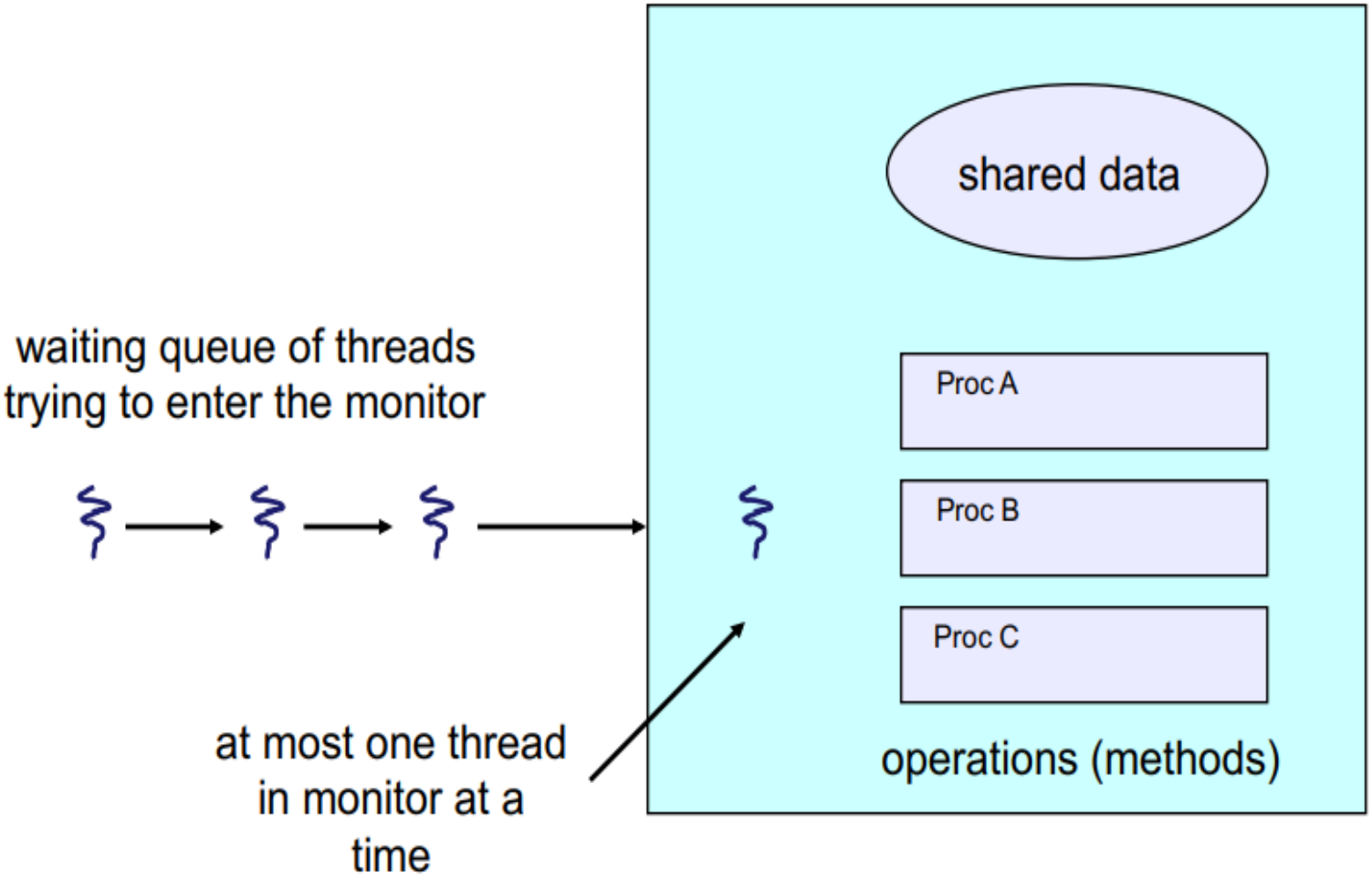
- Incorrect use of semaphore operations will result in timing errors.
- For example, If the **wait()** and **signal()** order is interchanged such as below, then there maybe several processes in the critical section
  - signal (mutex) …. wait (mutex)
- In the below cases a deadlock may occur.
  - wait (mutex) … wait (mutex)
  - Omitting of wait (mutex) or signal (mutex) (or both)
- Wait() and Signal() are scattered among several processes. Therefore, it is difficult to understand their effects.
- Usage must be correct in all the processes.
- One bad process can kill the whole system.

Monitors

- A **monitor** is a programming language construct that controls access to shared data
- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- A monitor is a module that encapsulates
  - Shared **data structures**
  - **Procedures** that operate on the shared data structures
  - **Synchronization** between concurrent threads that invoke the procedures
- Only one process may be active within the monitor at a time
  - The monitor ensures mutual exclusion
  - no need to program this constraint explicitly

```
monitor monitor-name   {
  // shared variable declarations
  procedure P1 (…) { …. }
  procedure Pn (…) {……}
     Initialization code (…) { … }
  }
}
```

Schematic view of a Monitor



Monitor in Java (Synchronized Java methods)

- The monitor enforces mutual exclusive access to synchronized methods invoked on the associated object.
- A java class can be turned into monitor by making its one or more classes **synchronized**

- For example, a bounded buffer class can be made monitor by:

```
public void synchronized put( Object item )

public char synchronized get( )
```

Monitor for Producer-Consumer in Java

```
class Buffer {
   private char [] buffer;
   private int count = 0, in = 0, out = 0;
   Buffer(int size)   {
      buffer = new char[size];   }
 public synchronized void Put(char c) {
     while(count == buffer.length) ;
     System.out.println("Producing " + c + " ...");
     buffer[in] = c;
     in = (in + 1) % buffer.length;
     count++;    }
 public synchronized char Get() {
     while (count == 0) ;
     char c = buffer[out];
     out = (out + 1) % buffer.length;
     count--;
     System.out.println("Consuming " + c + " ...");    return c;   }

}
```

```
public class PC
{
  public static void main(String [] args)
  {
     Buffer b = new Buffer(4);
     Producer p = new Producer(b);
     Consumer c = new Consumer(b);

     p.start(); //p puts items
     c.start(); //c gets items
  }
}
```

What are limitations of the above program?

- Execute the above program. And find out the problem? How can we handle the issue?

```
bilal@bilal-Latitude-E6540:~/Dropbox/os_prac$ java PC
Producing A ...
Consuming A ...
^Cbilal@bilal-Latitude-E6540:~/Dropbox/os_prac$ javac PCmonitor.j
```

Condition variables

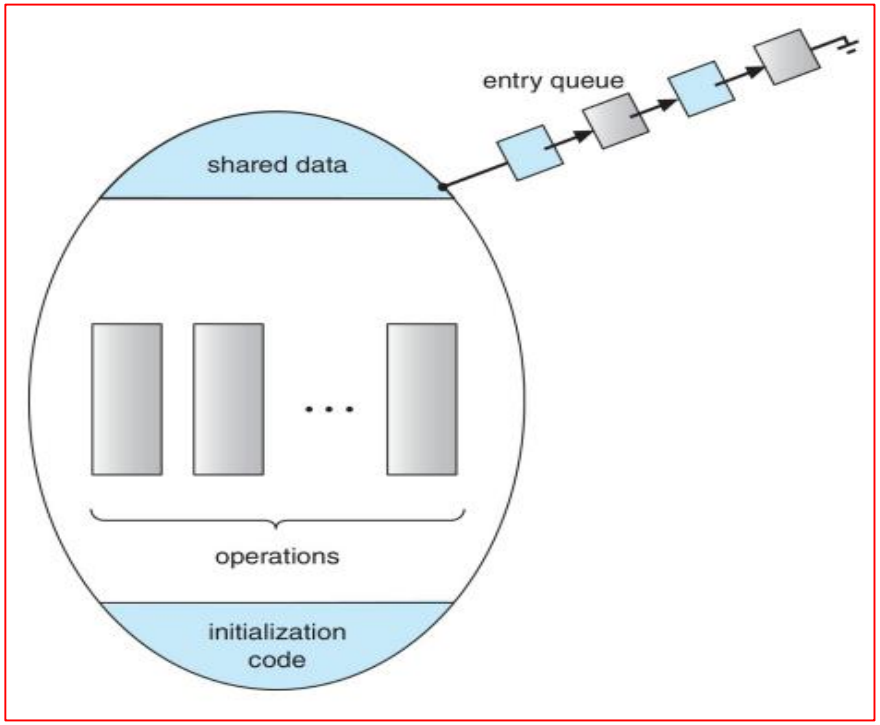
- Mutual exclusion is an easy task with monitors.
- But not powerful enough to model some synchronization schemes
- While a process is executing in a monitor, it may have to wait until an event occurs.
- Process synchronization is done using condition variables, which represent conditions (for example, **notEmpty**, **notFull**) a process may need to wait for before executing in the monitor
- A ***condition variable***, or a condition, has a private waiting list, and two public methods: ***signal*** and ***wait***.
-

Condition Variables

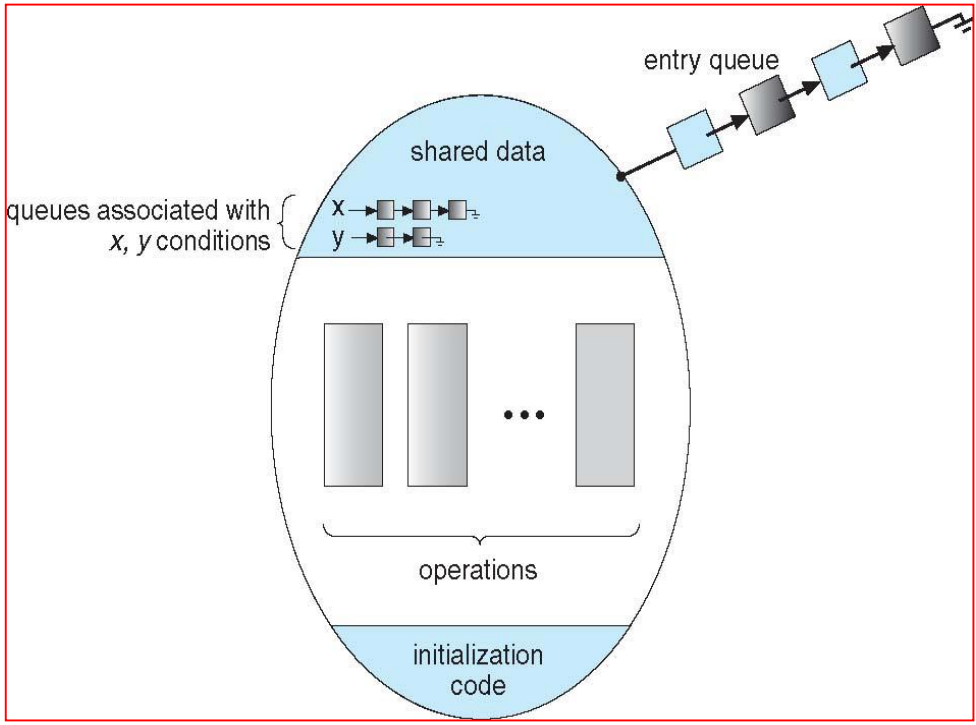
- Let ***x*** be a condition variable, defined as:
  - **`condition x;`**
- ***Two*** operations are allowed on a condition variable:
  - **`x.wait()`** – a process that invokes the operation is suspended until **`x.signal()`**
  - **`x.signal()`** – resumes one of processes (if any) that invoked **`x.wait()`**
    - If no one invoke **`x.wait()`** on the condition variable **x**, then **x.signal()** has no effect on the variable
- Condition ***wait*** and condition ***signal*** can only be used ***in a monitor***.

Monitor with Condition Variables

Monitor without condition variables



Monitor with condition variables



Bounded Buffer (Producer Consumer) with Monitors

```
monitor BoundedBuffer {

// Private variables …

        Object buf = new Object[n];

integer out = 0, // index of first full slot

        in = 0, // index of first empty slot

        count = 0; // number of full slots

// Condition variables ...

condvar not_full, // signalled when count < n. Consumer sends signal.

not_empty; // signalled when count > 0. Producer sends signal on this var.

//See next slide for the remaining part.
```

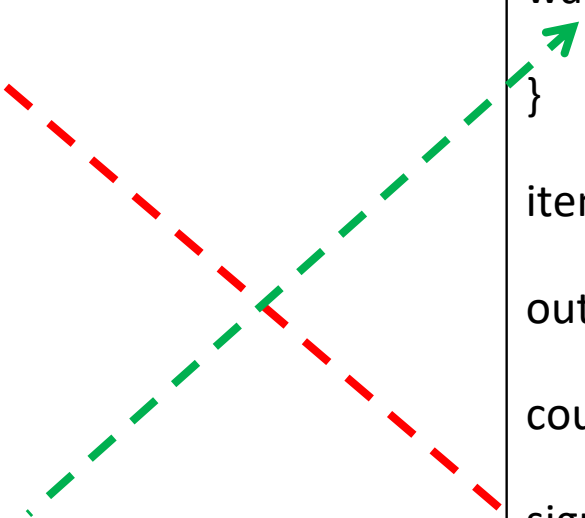
Bounded Buffer (Producer Consumer) with Monitors

// Monitor procedures ...

//(signal & continue signalling discipline)

```
procedure put(Object data) {
while(count == n) {
wait(not_full);
}
buf[in] = data;
in = (in + 1) % n;
count++;
signal(not_empty);
}
```

```
procedure get(Object &item) {
while(count == 0) {
wait(not_empty);
}
item = buf[out];
out = (out + 1) %n;
count--;
signal(not_full);
} }
```



Synchronization using wait() and notify()

- The wait() method suspends the calling thread. It wakes up only when another thread calls notify().

Producer Thread

```
for(int i = 0; i < 10; i++) {
    buffer.Put((char)('A'+ i%26 ));
}
```

Consumer Thread

```
for(int i = 0; i < 10; i++) {
        buffer.Get();
}
```

```
public synchronized void Put(char c) {
        if (count == buffer.length)  wait();
        System.out.println("Producing " + c + " ...");
        buffer[in] = c;
        in = (in + 1) % buffer.length;
        count++;
        notify();      }
```

```
public synchronized char Get() {
        if (count == 0)    wait();
        char c = buffer[out];
        out = (out + 1) % buffer.length;
        count--;
        System.out.println("Consuming " + c + " ...");
        notify();   }
```


bilal@bilal-Latitude-E6540:~/Dropbox/os_prac$ java PCmonitor
Producing A ...
Producing B ...
Producing C ...
Producing D ...
Consuming A ...
Consuming B ...
Consuming C ...
Consuming D ...
Producing E ...
Producing F ...
Producing G ...
Producing H ...
Consuming E ...
Consuming F ...
Consuming G ...
Consuming H ...
Producing I ...
Producing J ...
Consuming I ...
Consuming J ...
bilal@bilal-Latitude-E6540:~/Dropbox/os_prac$

Condition Variables Choices

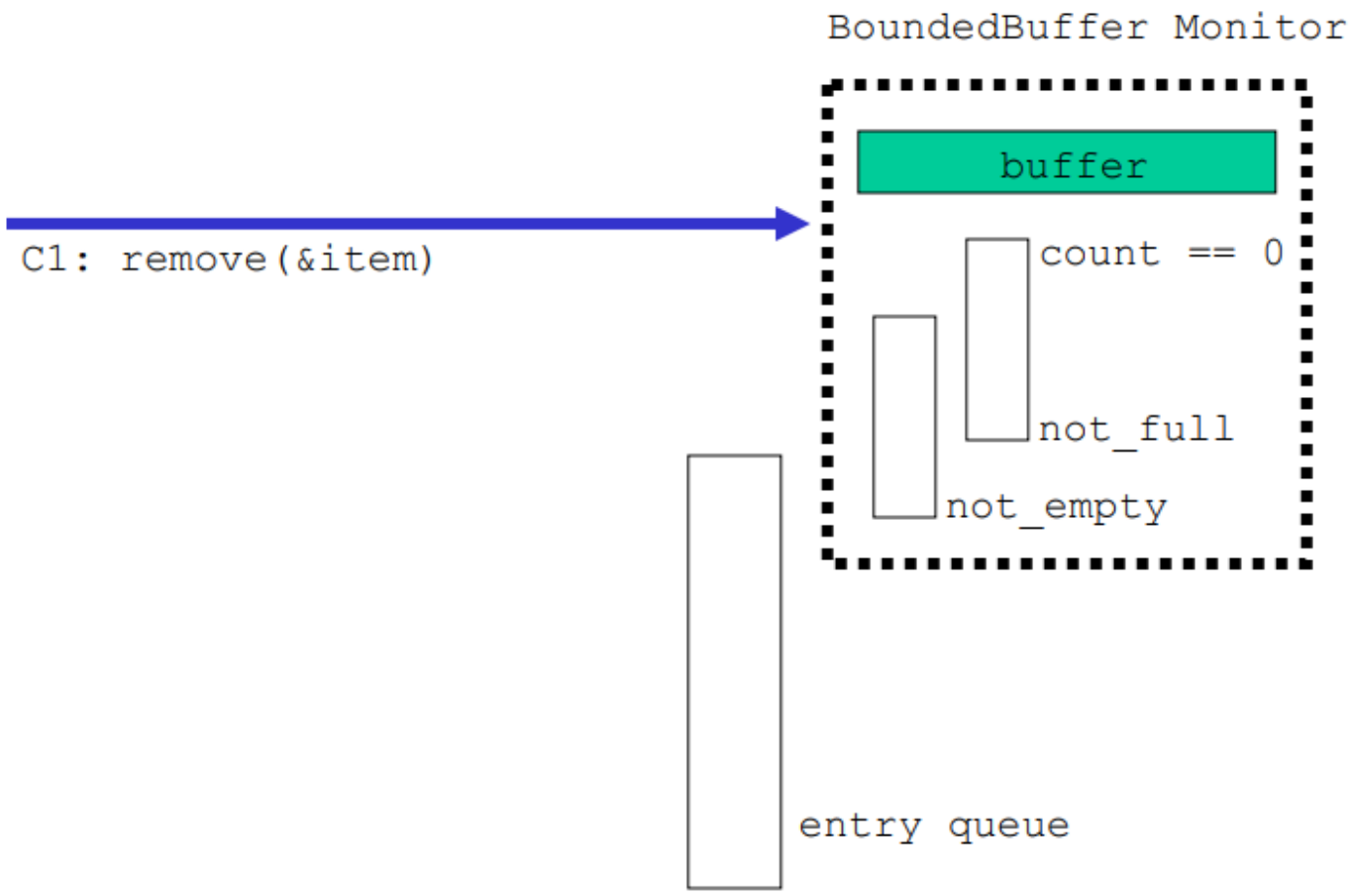
- When a monitor procedure calls **signal** on a condition variable, it wakes up the first blocked process in the *delay queue* waiting on the condition.

  - ***Signal and Wait:*** the signaler waits until some later time and the signaled process executes now. In other words, this semantic lets the newly awakened process run, suspending the other one.
  - ***Signal and Continue:*** the signaler continues and the signaled process executes at some later time. In other words, the newly awakened process is moved from the signal queue back to the entry queue.

  - The examples in this lecture use the *signal and continue* signaling discipline.

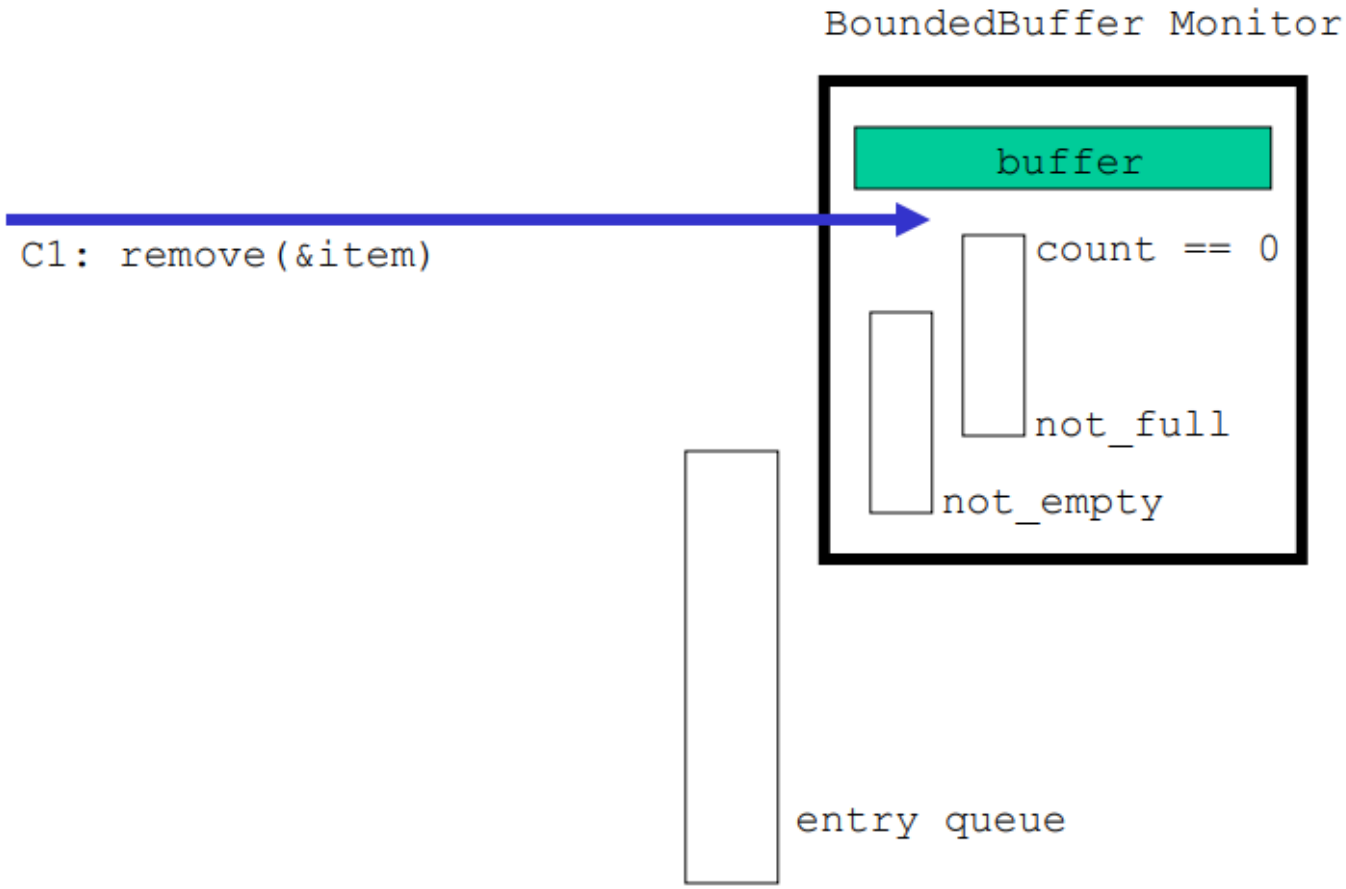
Example Trace 1

- A consumer C1 arrives and wants to get() an item from the buffer.
- There is neither any consumer nor any producer currently in the monitor.
- So C1 will be allowed to enter monitor.



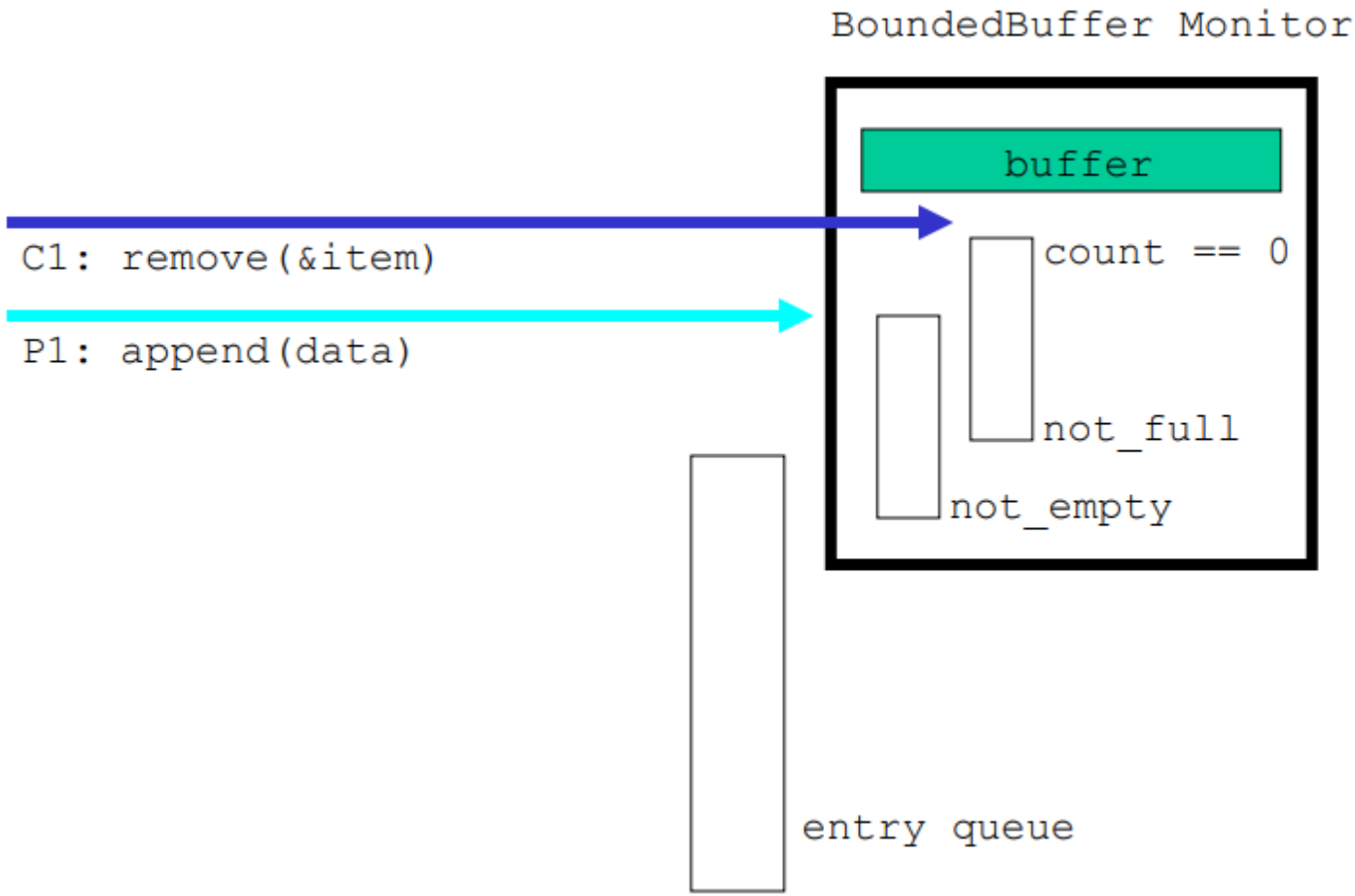
Example Trace 2

- C1 is allowed to enter monitor.
- However, there is no item in the buffer. So C1 will wait on the condition variable **Not_empty**.

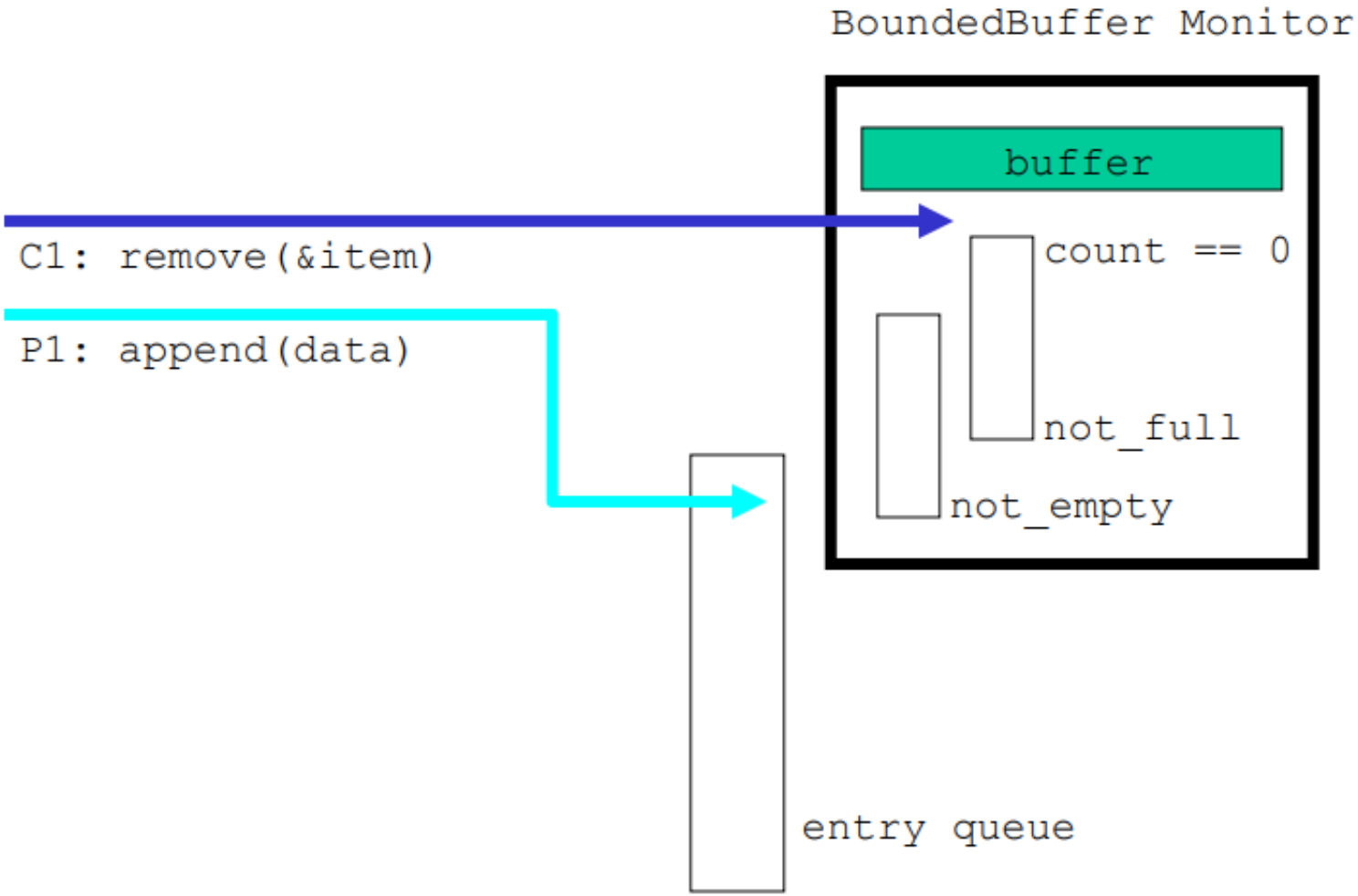


Example Trace 3

- Next producer **P1** arrives.

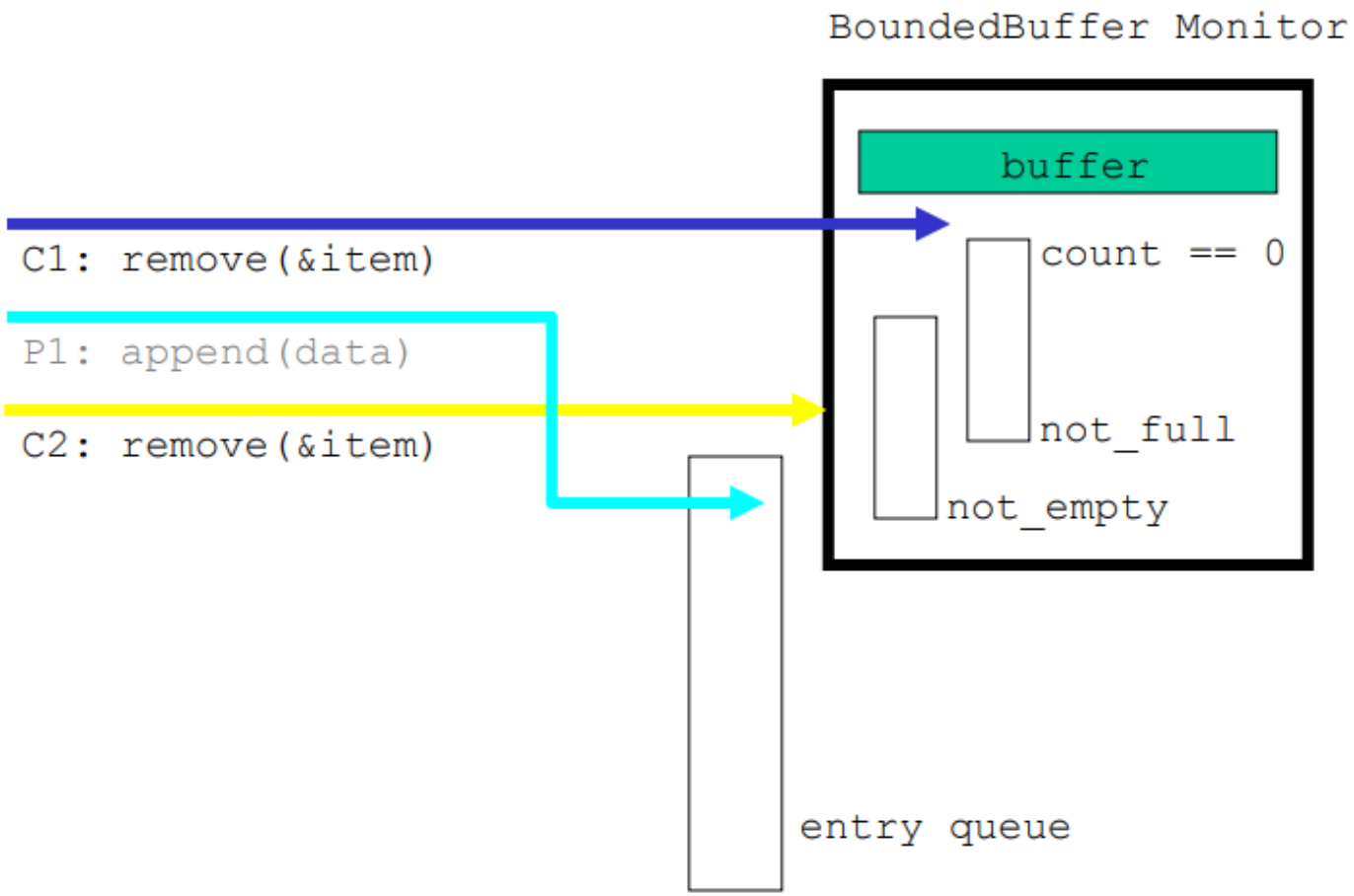


Example Trace 4



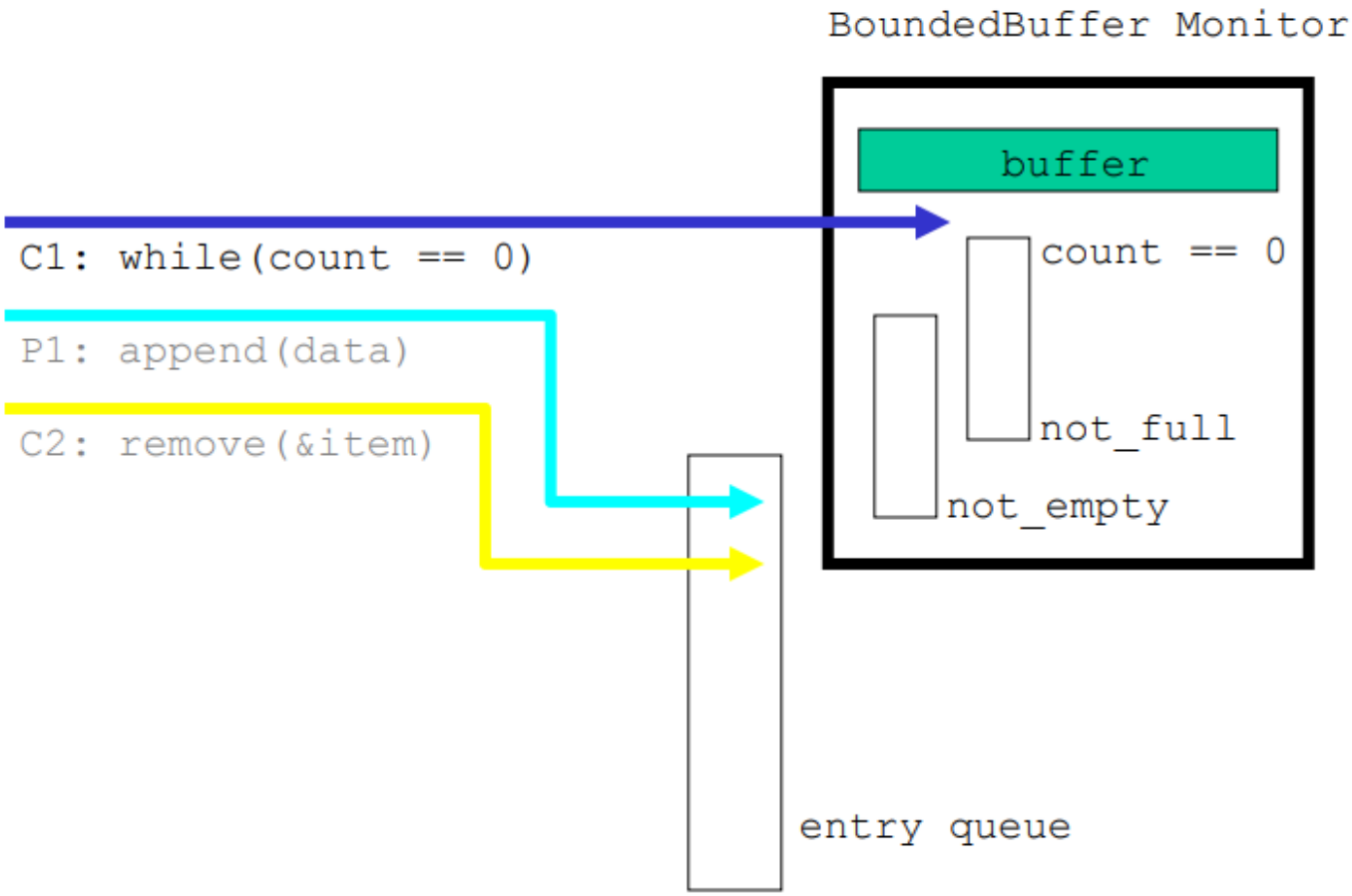
Example Trace 5

- Meanwhile consumer **C2** arrives.

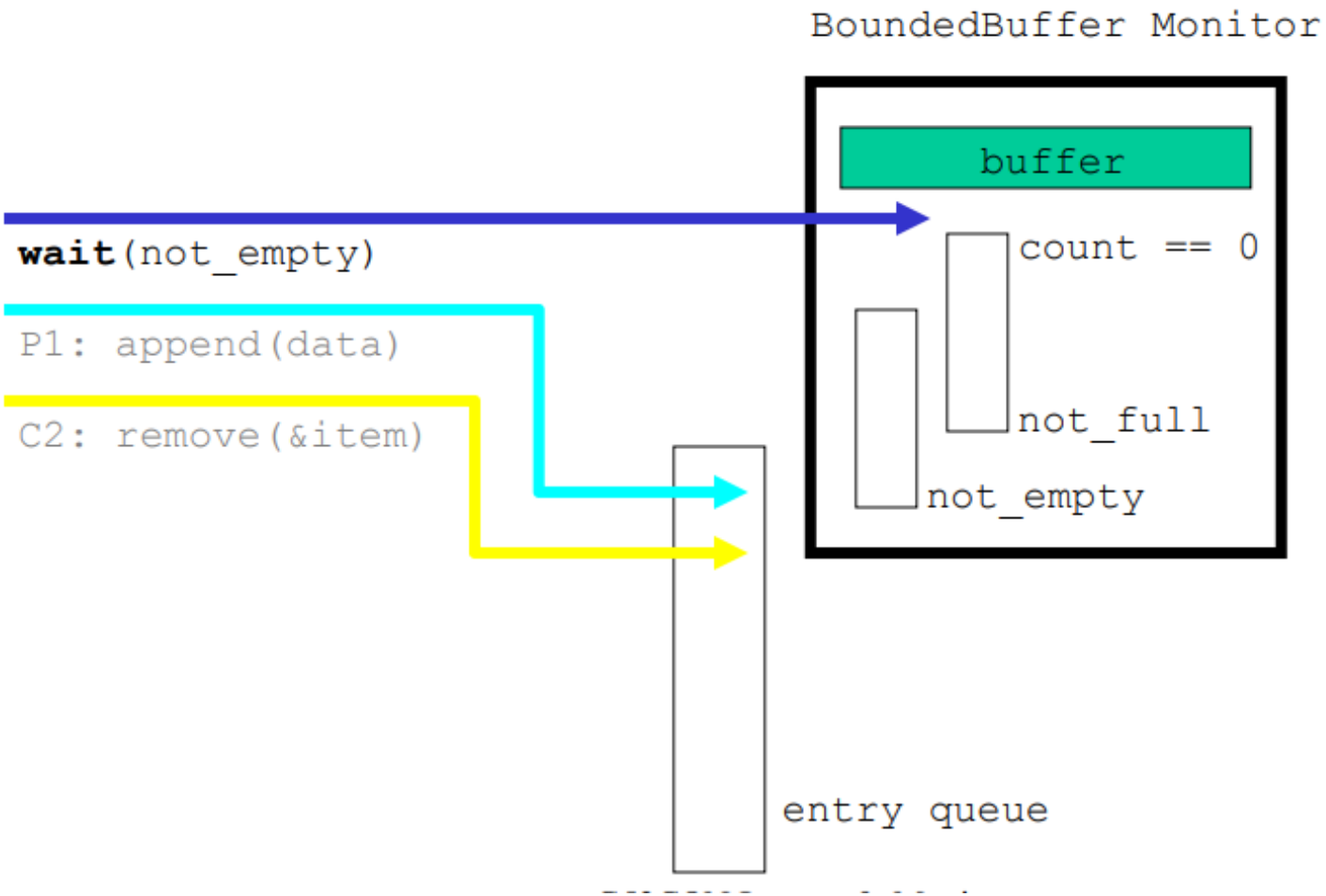


Example Trace 6

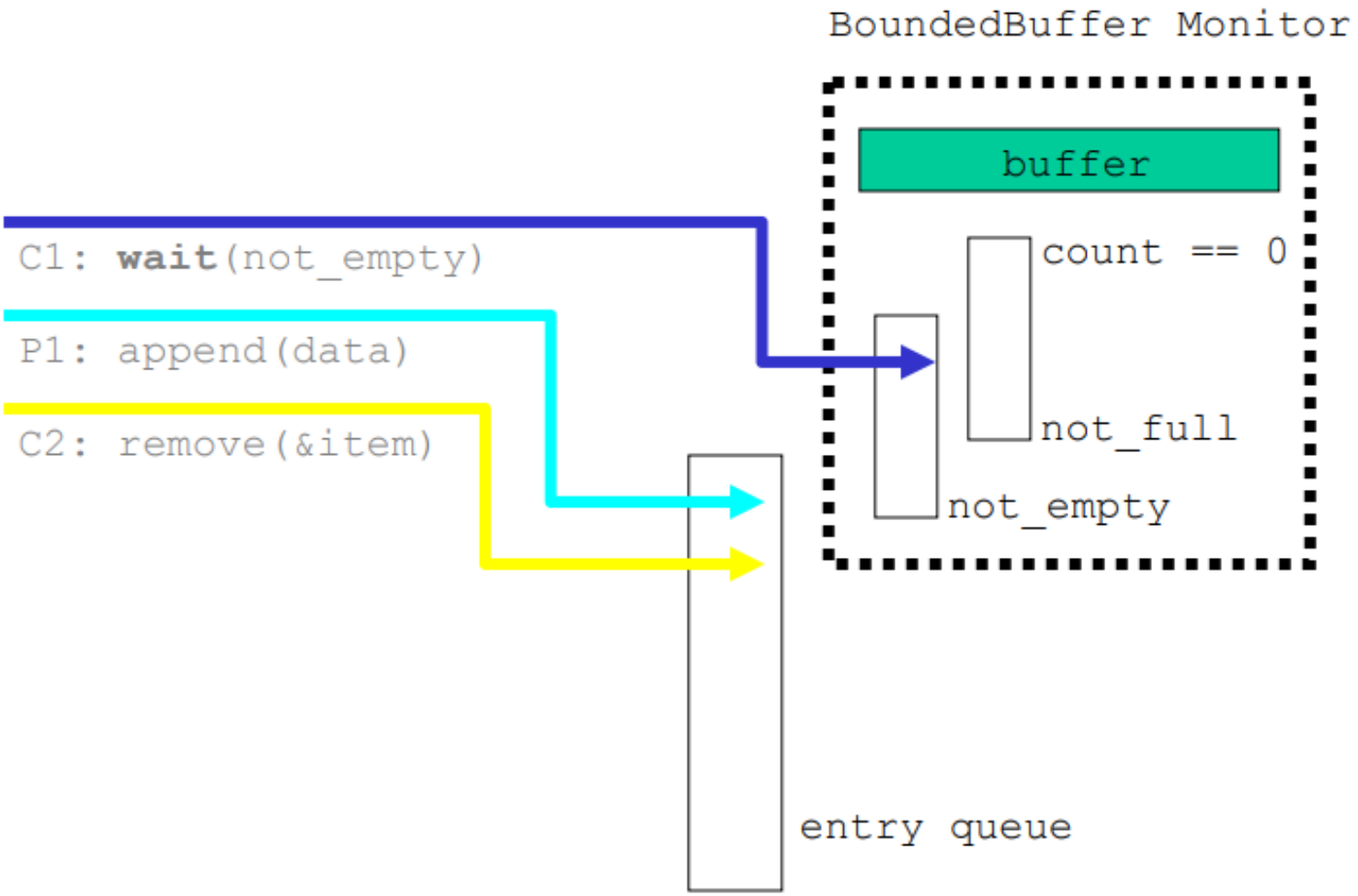
- Both P1 and C2 will wait outside the monitor to ensure Mutual exclusion.



Example Trace 7

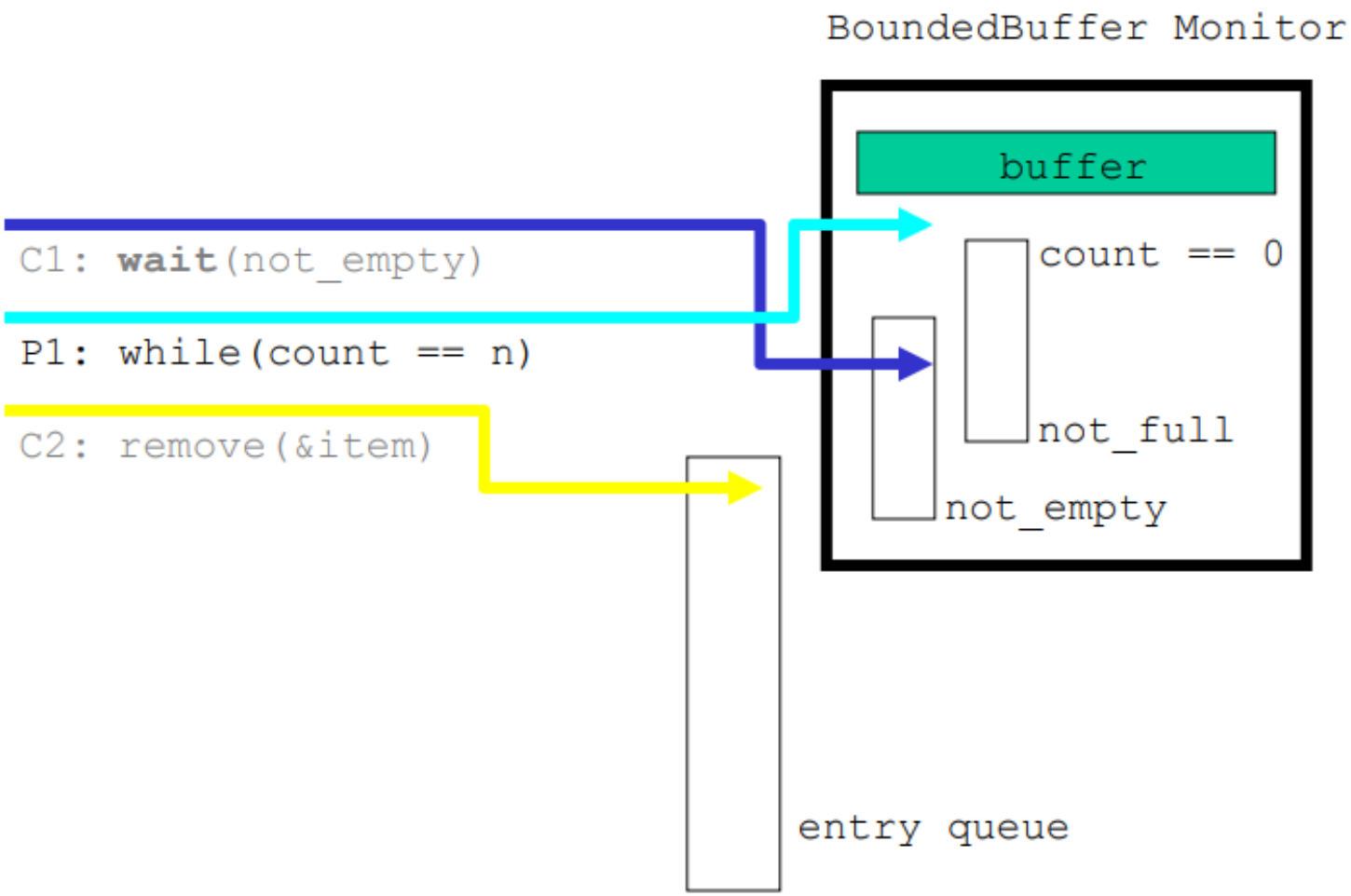


Example Trace 8



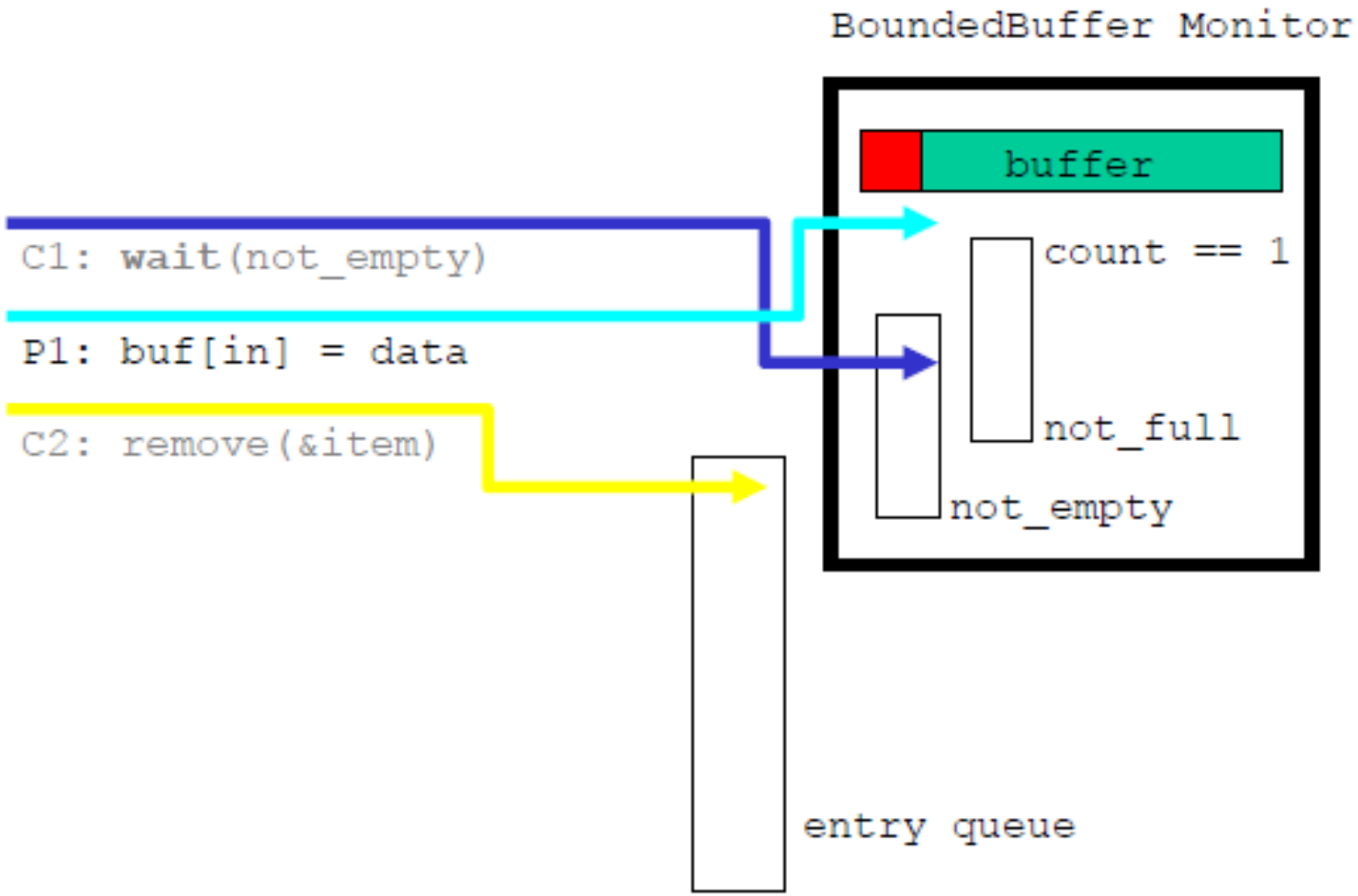
Example Trace 9

- P1 can now enter the monitor as **C1** is waiting on the condition variable.

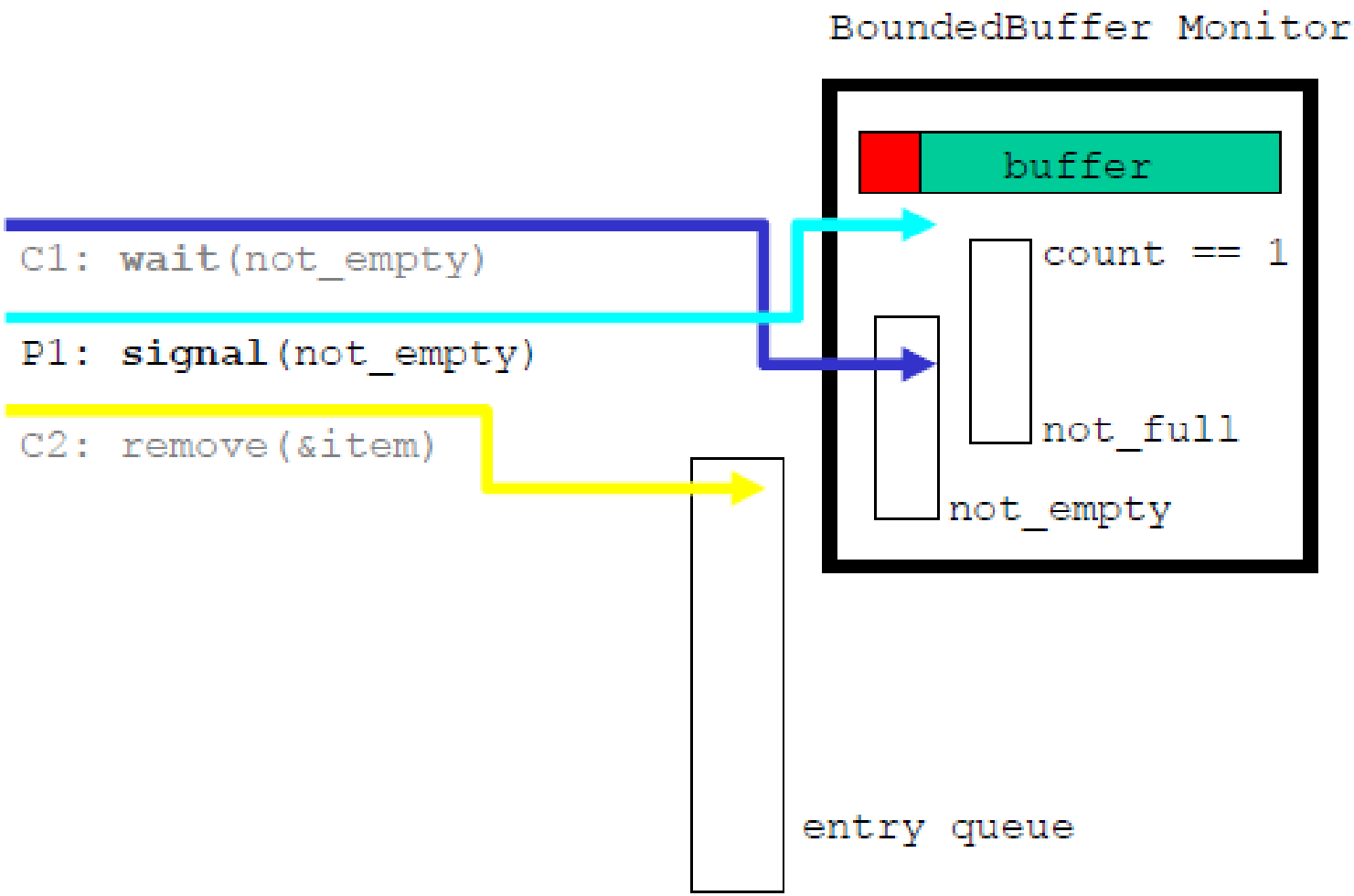


Example Trace 10

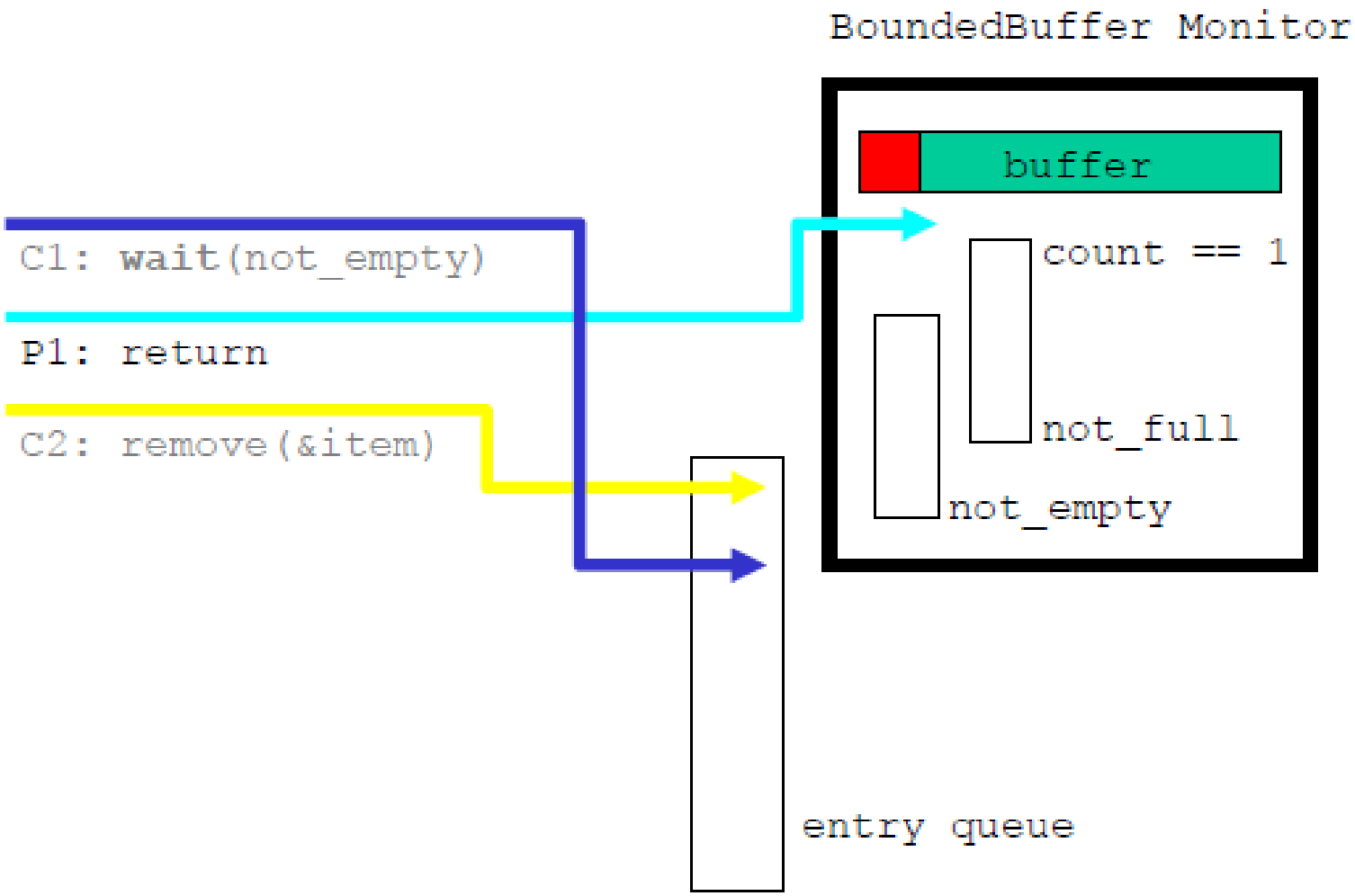
- P1 produces an item in the buffer.



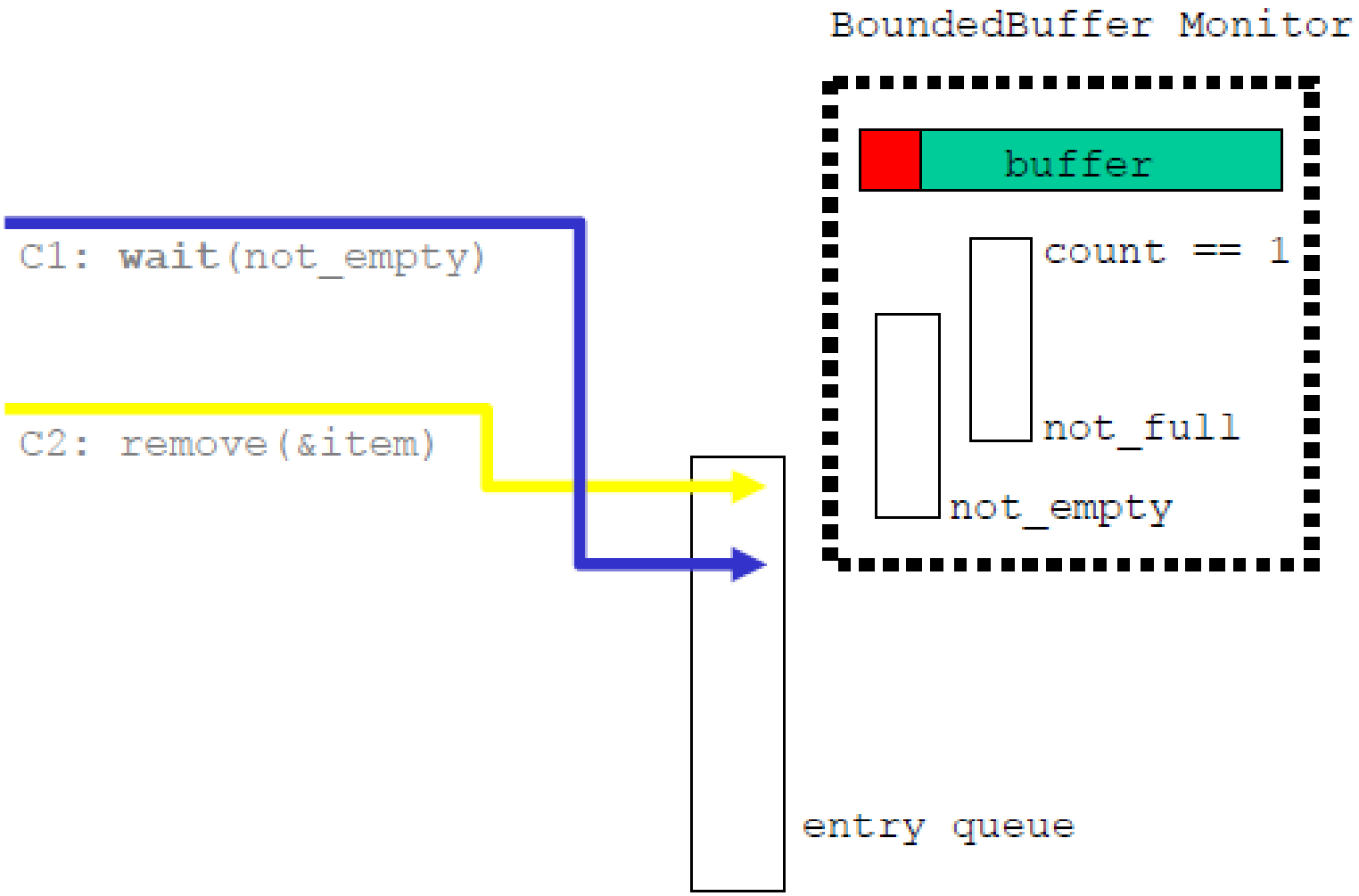
Example Trace 11



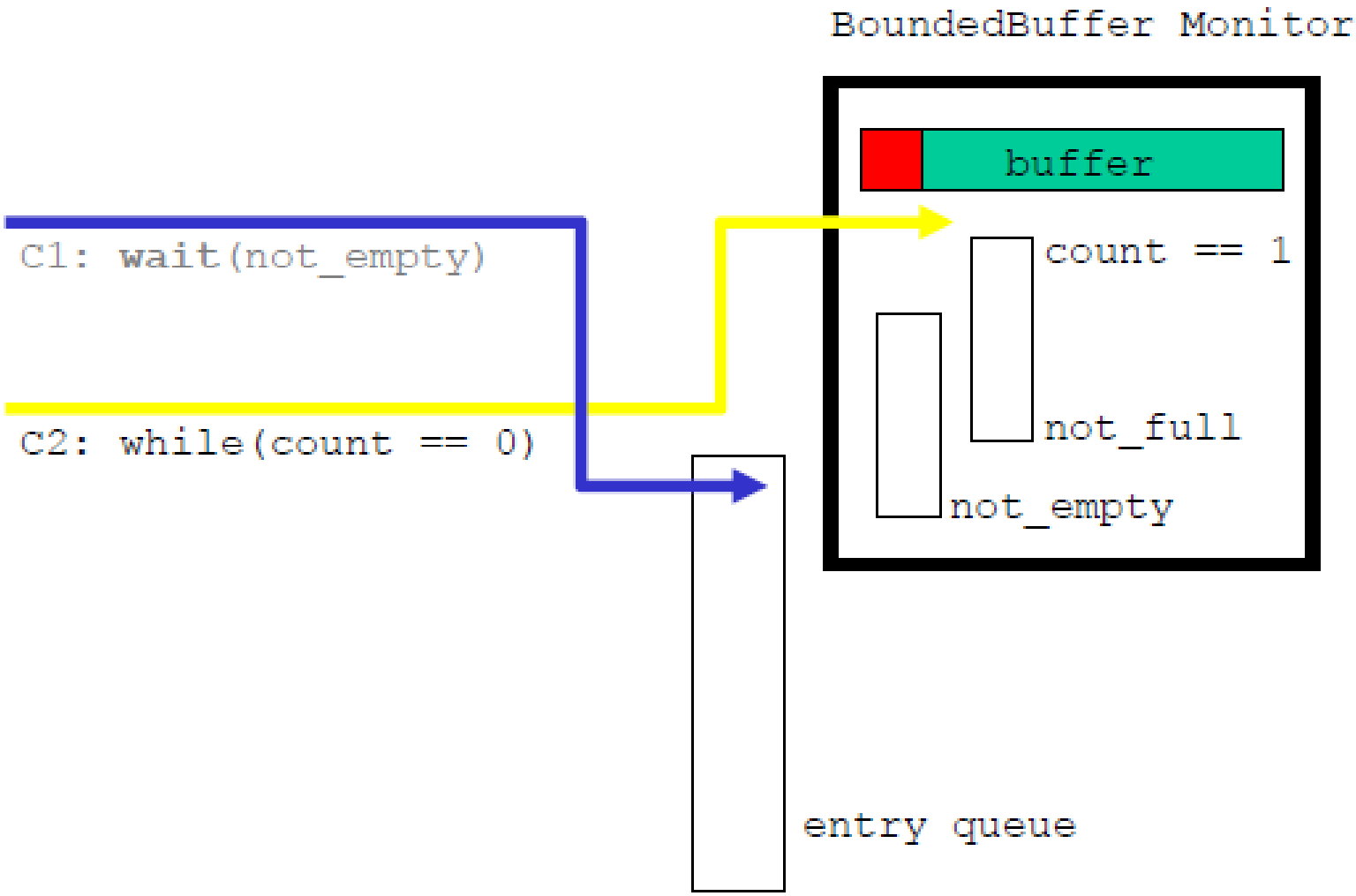
Example Trace 12



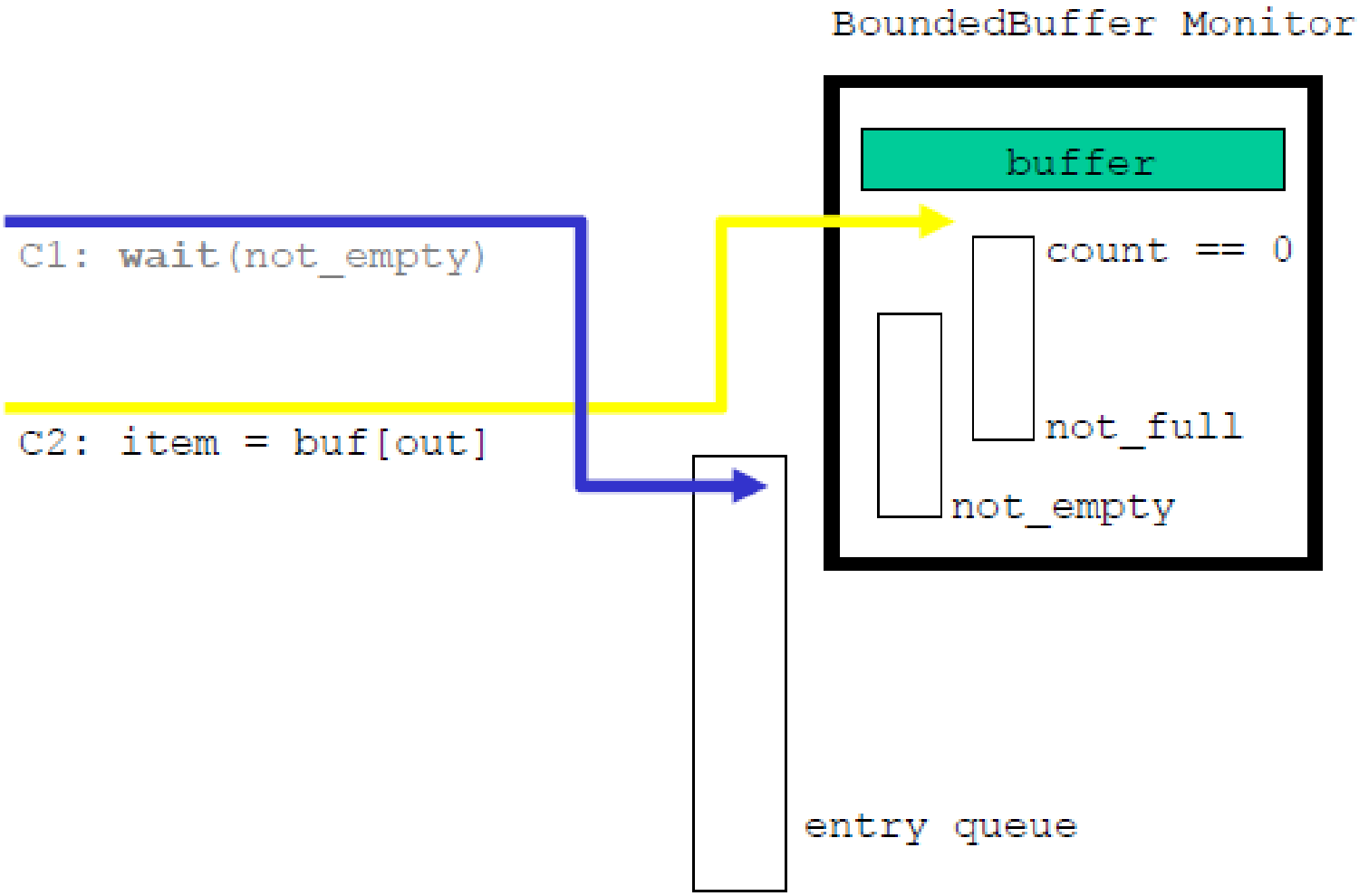
Example Trace 13



Example Trace 14

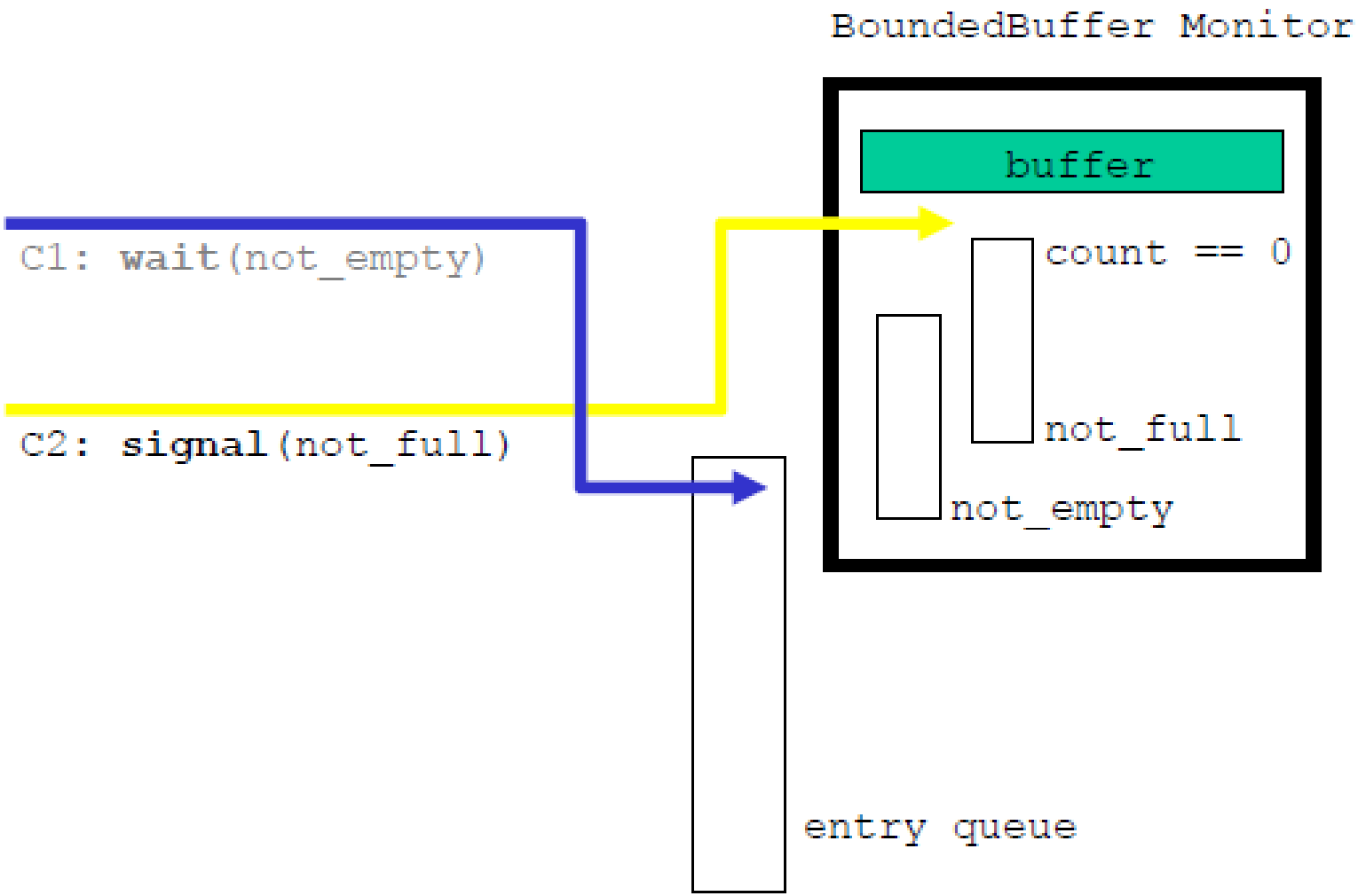


Example Trace 15

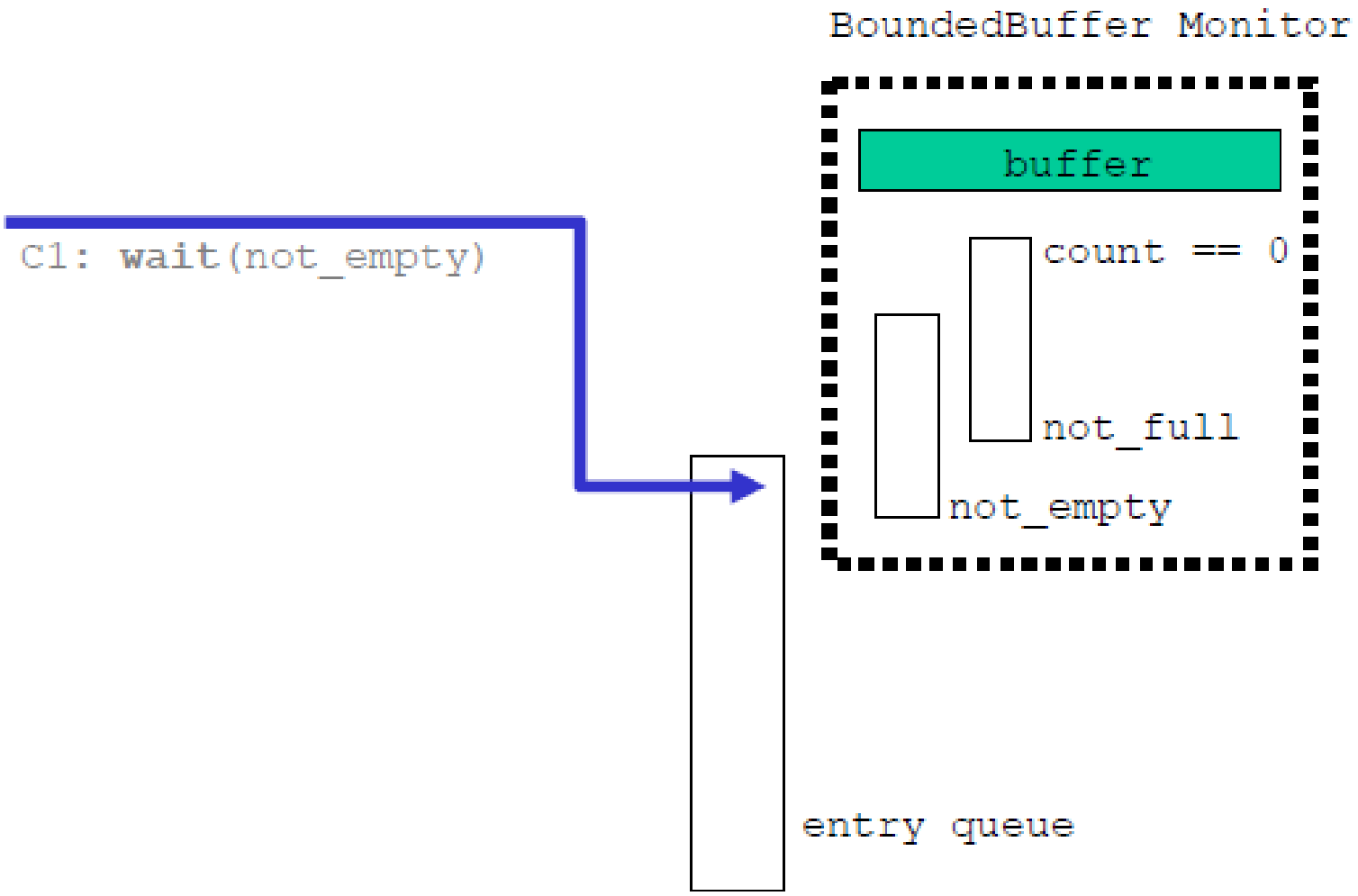


Example Trace 16

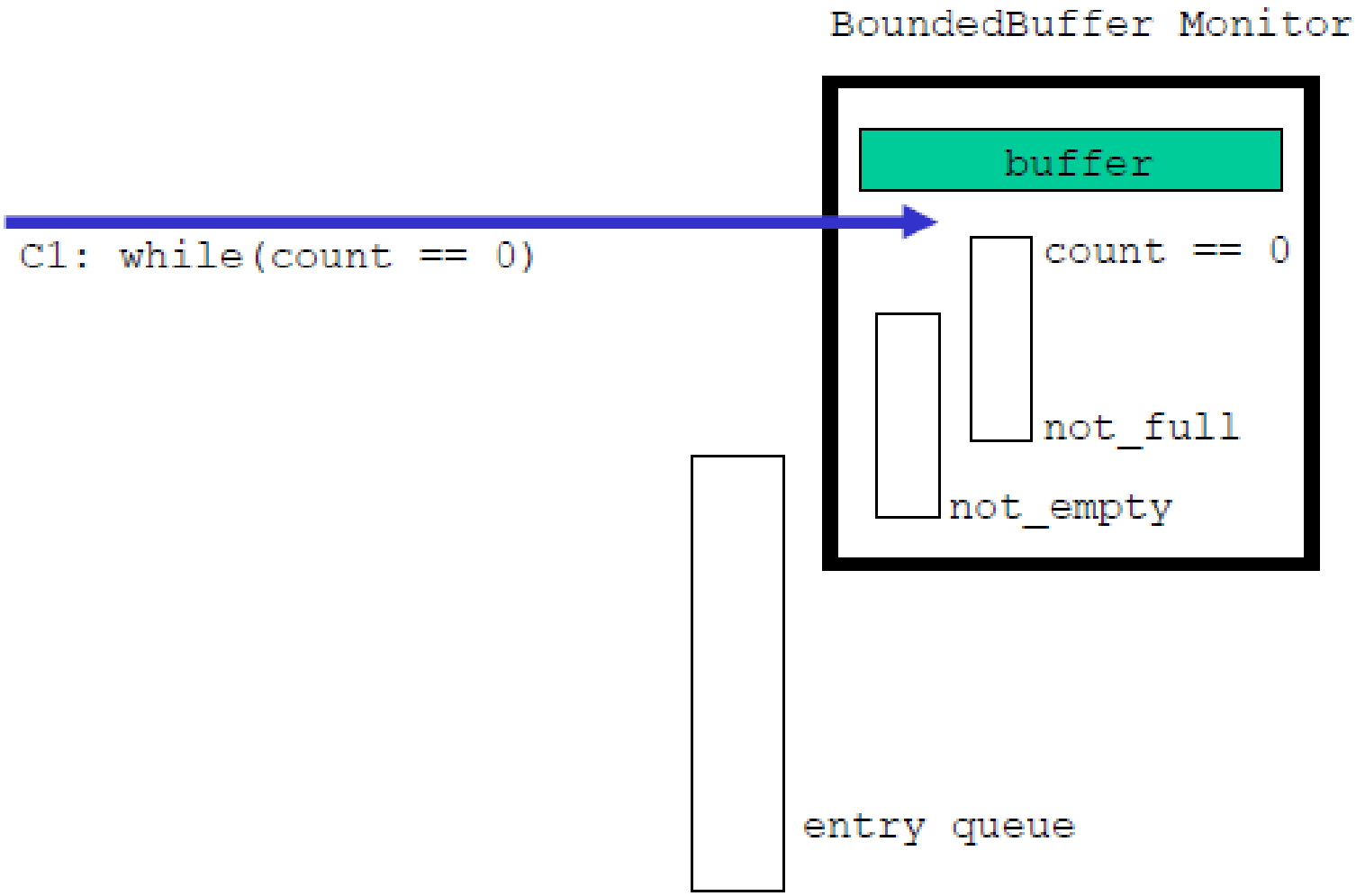
- **signal(not_full)** does not have any effect if there is no producer waiting on the **not_full** condition variable.



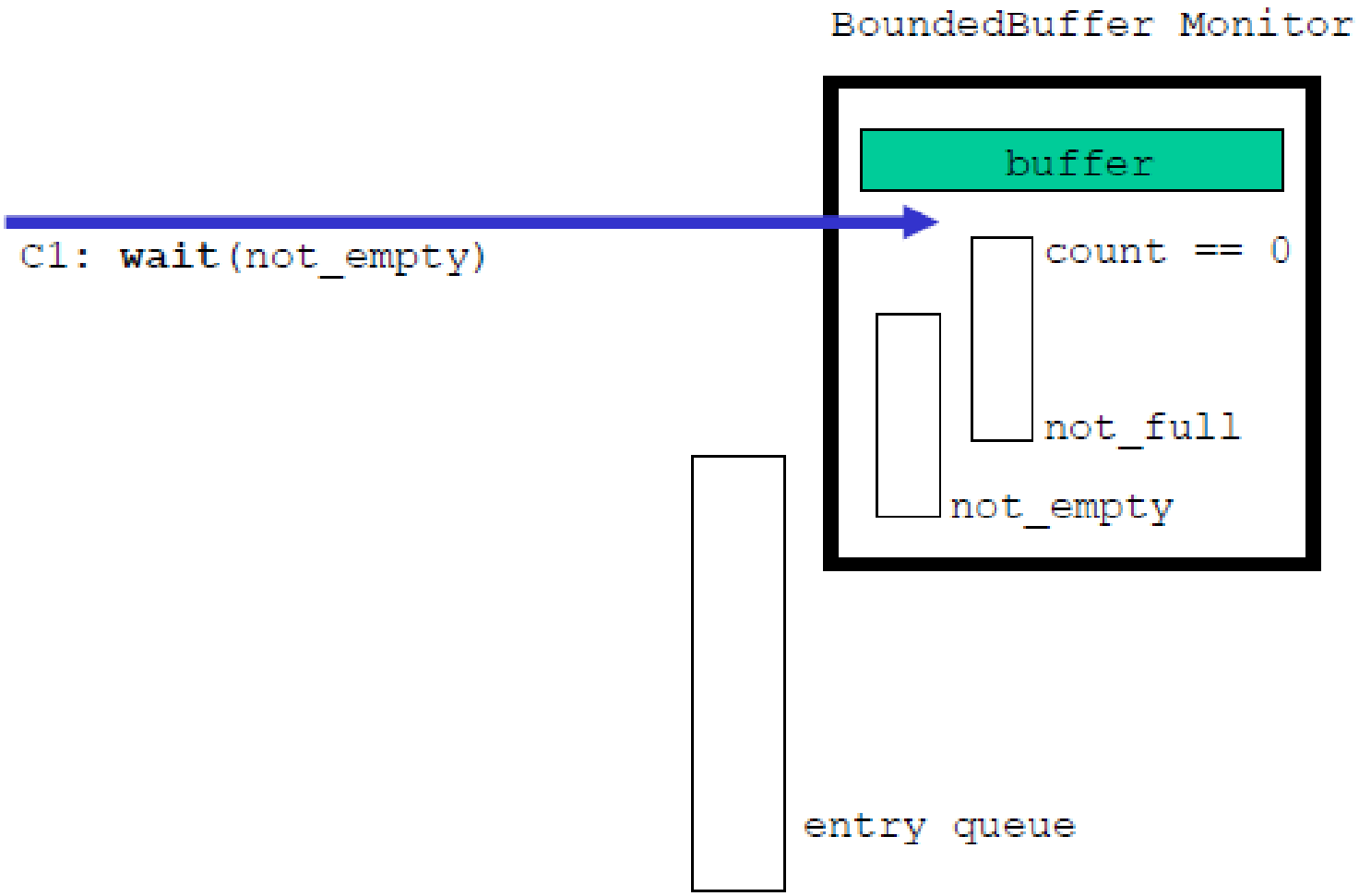
Example Trace 17



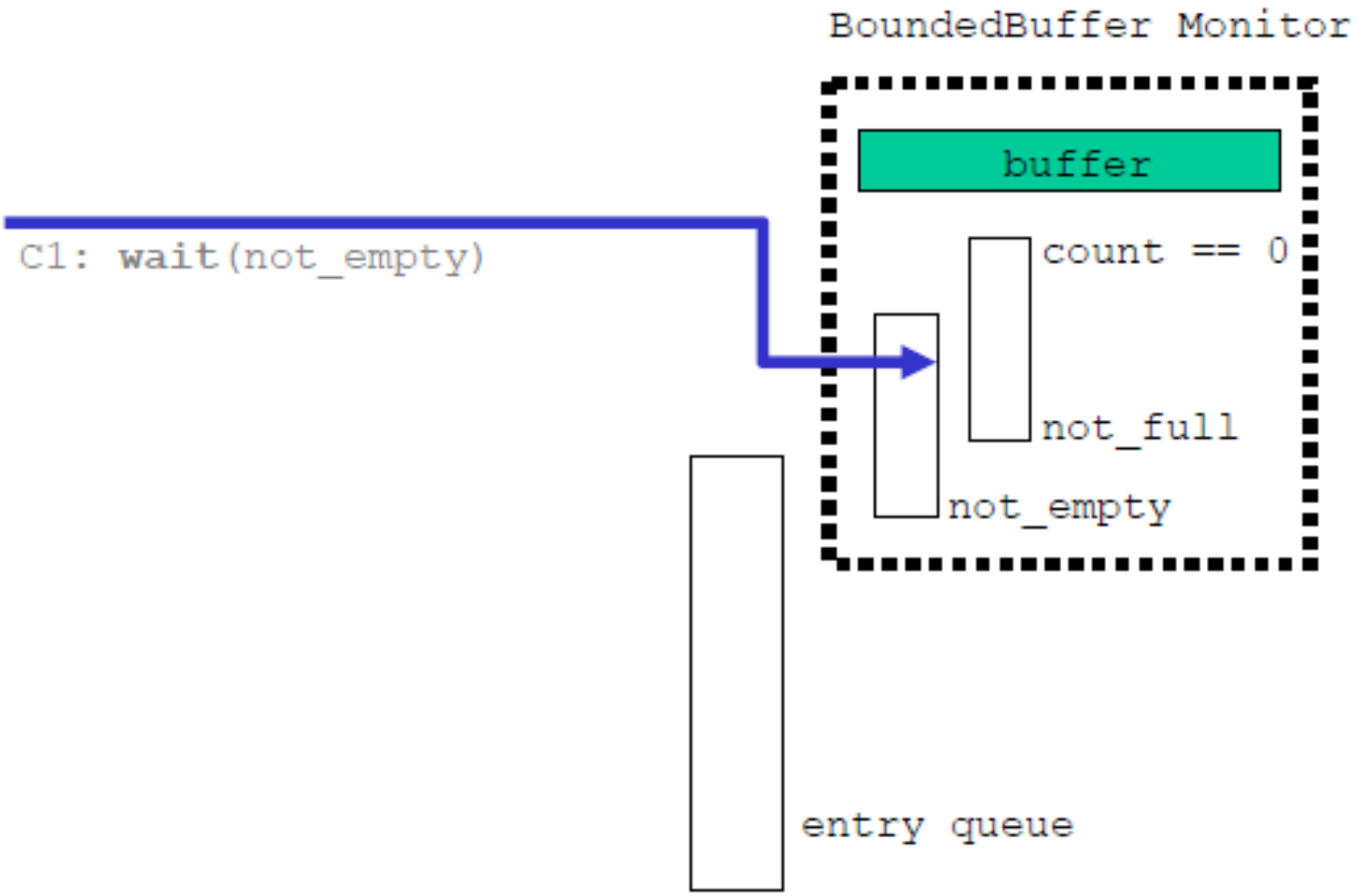
Example Trace 18



Example Trace 19



Example Trace 20



End of Chapter 5