OPERATING SYSTEMS

Subject code : CSC 404

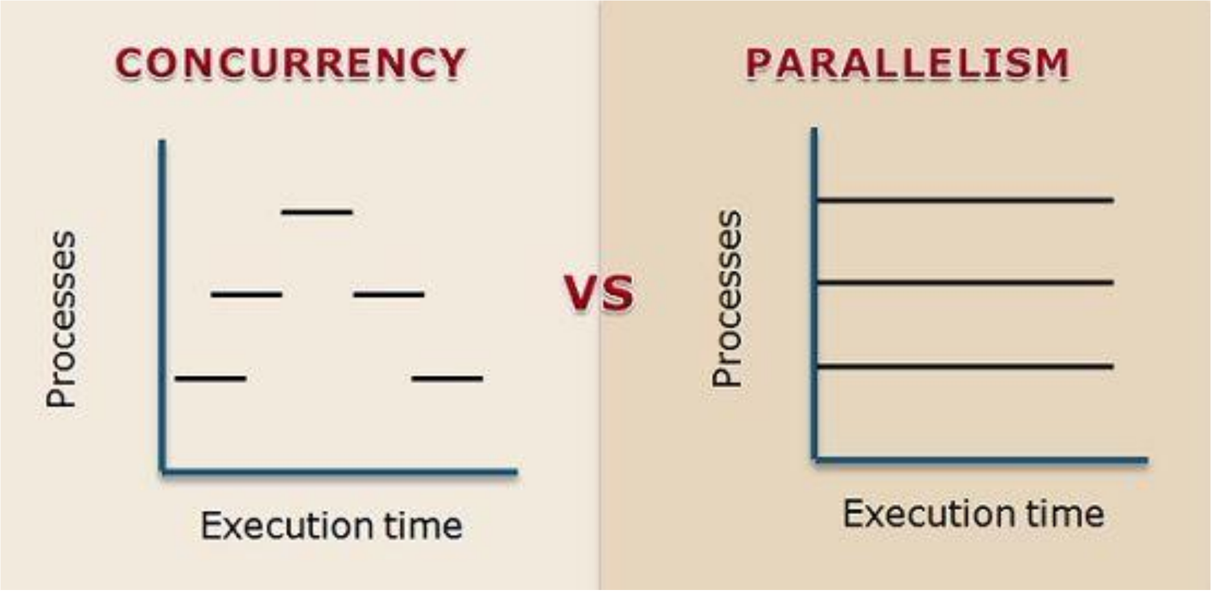
Subject In-charge

Nidhi Gaur
Assistant Professor
email: nidhigaur@sfit.ac.in

Module 3:Process Synchronization

Concurrency vs Parallelism

- If multiple processes are running concurrently, they each take turns working towards accomplishing their goals.
- If multiple processes are running in parallel, they are all accomplishing their tasks simultaneously and independently of each other.



Processes

Processes executing concurrently in OS can be:

- Independent processes: cannot affect or be affected by other processes executing in the system
- Cooperating processes: can affect or be affected by other processes in the system.
- Any process that shares data with other processes is a cooperating process.

Interprocess communication

- Inherently OS don't allow processes to interfere with each other's environment.
- Reasons for providing an environment that allows process cooperation.
  - Information sharing
  - Computation speed-up
  - Modularity
  - Convenience

<u>Cooperating processes require IPC</u>

Concurrent processes

- Independent processes are easier to manage.
- Problem arises when processes are interacting.
- They share some data structure or need to communicate.
- There should be mechanism to synchronize the processes.

Producer Consumer problem

- One of the cooperating processes example is Producer Consumer processes.
- A producer process produces information consumed by consumer process.
- For example, Compiler produces assembler code, which is consumed by assembler. The assembler in turn produces object modules, which are consumed by loader.
- One solution to Producer consumer problem is shared memory.
- To allow producer and consumer processes to run concurrently, we must have available buffer of items that can be filled by producer and consumed by consumer.

Background

- 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 producer-consumer problem that fills the buffer. We can do so by having an integer count that keeps track of the buffer size. Initially, count is set to 0. It is incremented by the producer after it produces a new item and is decremented by the consumer after it consumes an item.

Producer/consumer

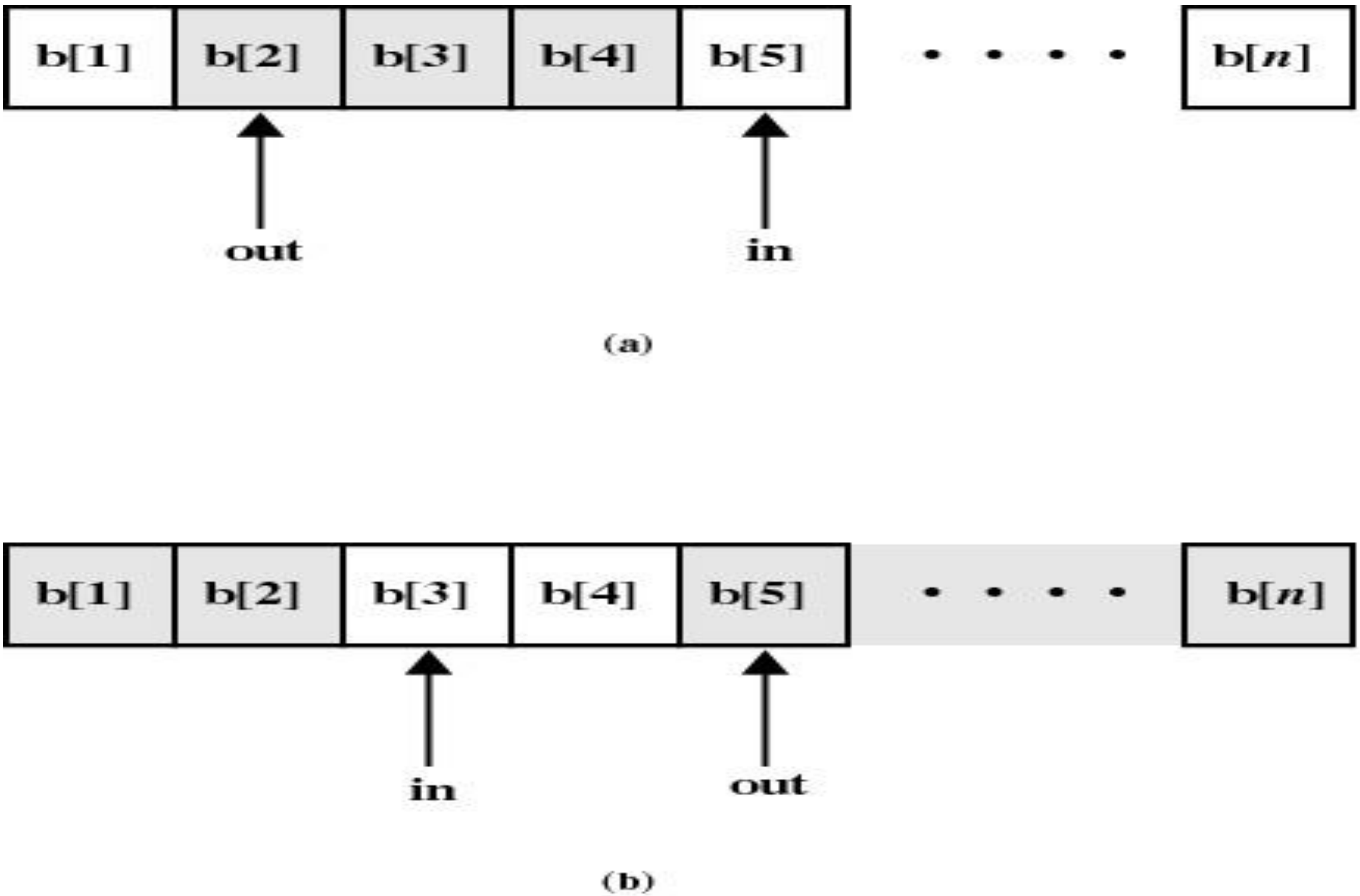


Figure 5.15 Finite Circular Buffer for the Producer/Consumer Problem

Race Condition

A situation where several processes access and manipulate the same data concurrently and outcome of the execution depends on the particular order in which the access takes place.

- count++ could be implemented as

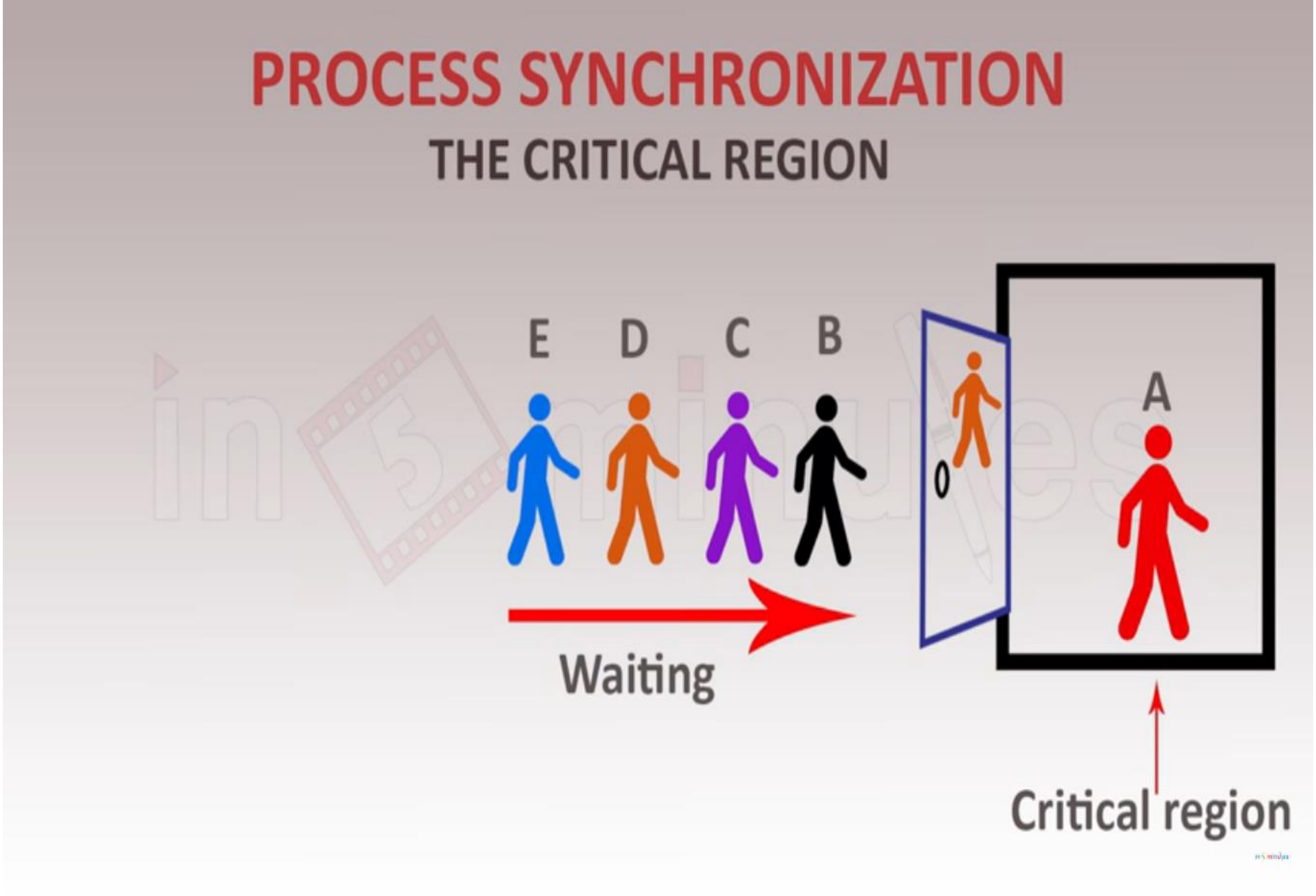
  register1 = count
  register1 = register1 + 1
  count = register1
- count-- could be implemented as

  register2 = count
  register2 = register2 - 1
  count = register2
- Consider this execution interleaving with "count = 5" initially:

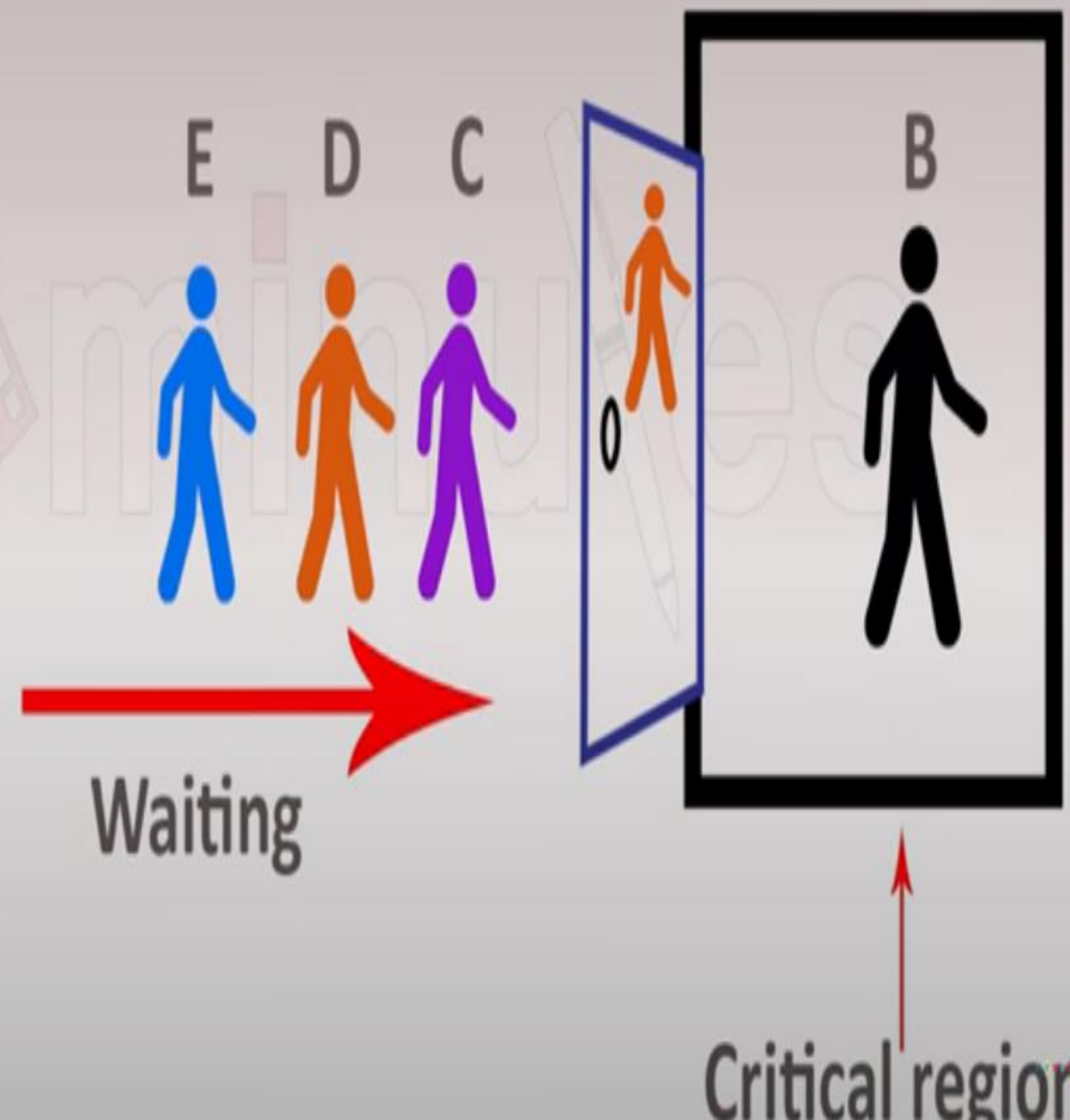
  S0: producer execute register1 = count {register1 = 5}
  S1: producer execute register1 = register1 + 1 {register1 = 6}
  S2: consumer execute register2 = count {register2 = 5}
  S3: consumer execute register2 = register2 - 1 {register2 = 4}
  S4: producer execute count = register1 {count = 6 }
  S5: consumer execute count = register2 {count = 4}

Mutual Exclusion

- An OS must be able to ensure cooperation among processes.
- If more than one process are sharing data then it must be protected from multiple accesses.
- When a process is given access to non-sharable resource such as printer, another process cannot be given access at the same time, otherwise output will be mixed.
- Mutual exclusion ensures that process do not access or update a resource concurrently.


PROCESS SYNCHRONIZATION
THE CRITICAL REGION
E
D
C
B
A
Waiting
Critical region

THE CRITICAL REGION



PROCESS SYNCHRONIZATION

THE CRITICAL REGION

Program → Some part → Access in Mutual Exclusive manner

PA

File access → P_x, P_y, P_z

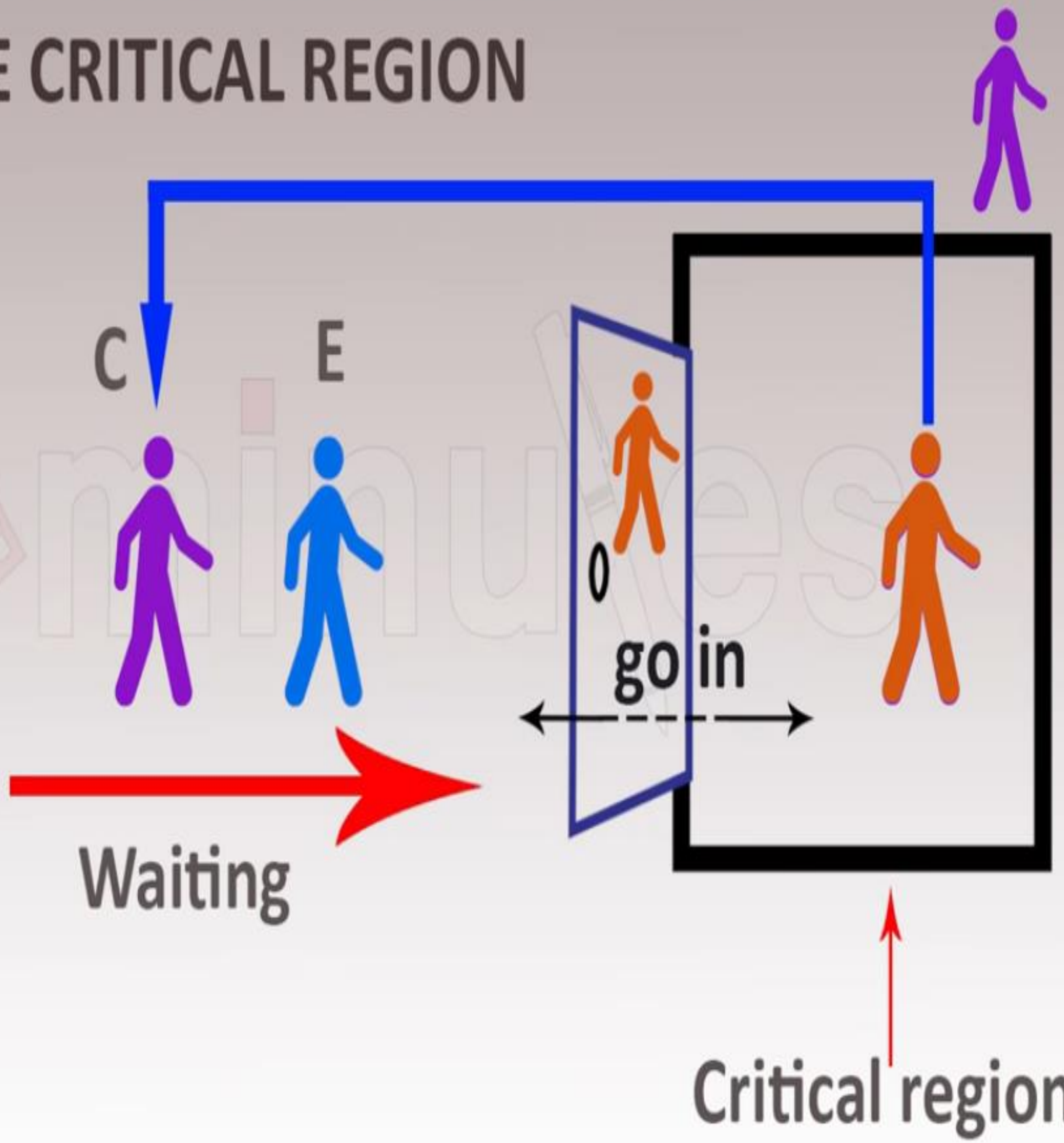
DataBase access

Variable access

PROCESS SYNCHRONIZATION

THE CRITICAL REGION

```
do
 {
   Enter CR;
  do something ;
   Exit CR;
   Reminder ;
 } while(True);
```



need

Critical section

Process p ()

Do {

Entry section

Critical section

Exit section

Remainder section

} while(true);

- Each process has segment of code known as critical section in which process updates table, write file.
- A section of code that must be protected.
- Process uses critical section to access shared memory

Critical section problem

To design a protocol that the processes can use to cooperate.

Requirements for solution to Critical-Section Problem

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 exist 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

Requirements for solution to Critical-Section Problem

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.

Process p ()
Do {
Entry section
Critical section
Exit section
Remainder section
} while(true);

Mutual exclusion

- Software approaches

First Attempt

Busy Waiting

- Process is always checking to see if it can enter the critical section
- Process can do nothing productive until it gets permission to enter its critical section

First Attempt

- Shared global variable **int turn** = 0;
- If value of *turn* is equal to no. of process, then process may proceed to its critical section, otherwise wait.
- Drawback: pace of execution is dictated by slower of the two processes.

Second Attempt

- Each process can examine the other's status but cannot alter it.
- When a process wants to enter the critical section it checks the other processes first.
- If no other process is in the critical section, it sets its status for the critical section
- This method does not guarantee mutual exclusion
- Each process can check the flags and then proceed to enter the critical section at the same time

Second Attempt

A boolean vector flag is defined.

P0→flag[0] ; P1→flag[1]

Consider sequence:

P0 executes while & finds flag[1] set to false.

P1 executes while & finds flag[0] set to false.

P0 sets flag[0] true & enters critical section.

P1 sets flag[1] to true & enters critical section.

Both process in critical section, program incorrect.

Solution is not independent of execution speed.

Second Attempt

Bcoz a process can change its state after the other process has checked it, second attempt failed.

Third Attempt

- Set flag to enter critical section before checking other processes
- If another process is in the critical section when the flag is set, the process is blocked until the other process releases the critical section
- Deadlock is possible when two process set their flags to enter the critical section. Now each process must wait for the other process to release the critical section

Fourth Attempt

- A process sets its flag to indicate its desire to enter its critical section but is prepared to reset the flag

Other processes are checked. **If they are in the critical region, the flag is reset and later set to indicate desire to enter the critical region.** This is repeated until the process can enter the critical region.

Fourth Attempt

It is possible for each process to set their flag, check other processes, and reset their flags.

Fourth Attempt

Consider the sequence:

P0 sets flag[0] to true.

P1 sets flag[1] to true.

P0 checks flag[1].

P1 checks flag[0].

P0 sets flag[0] to false.

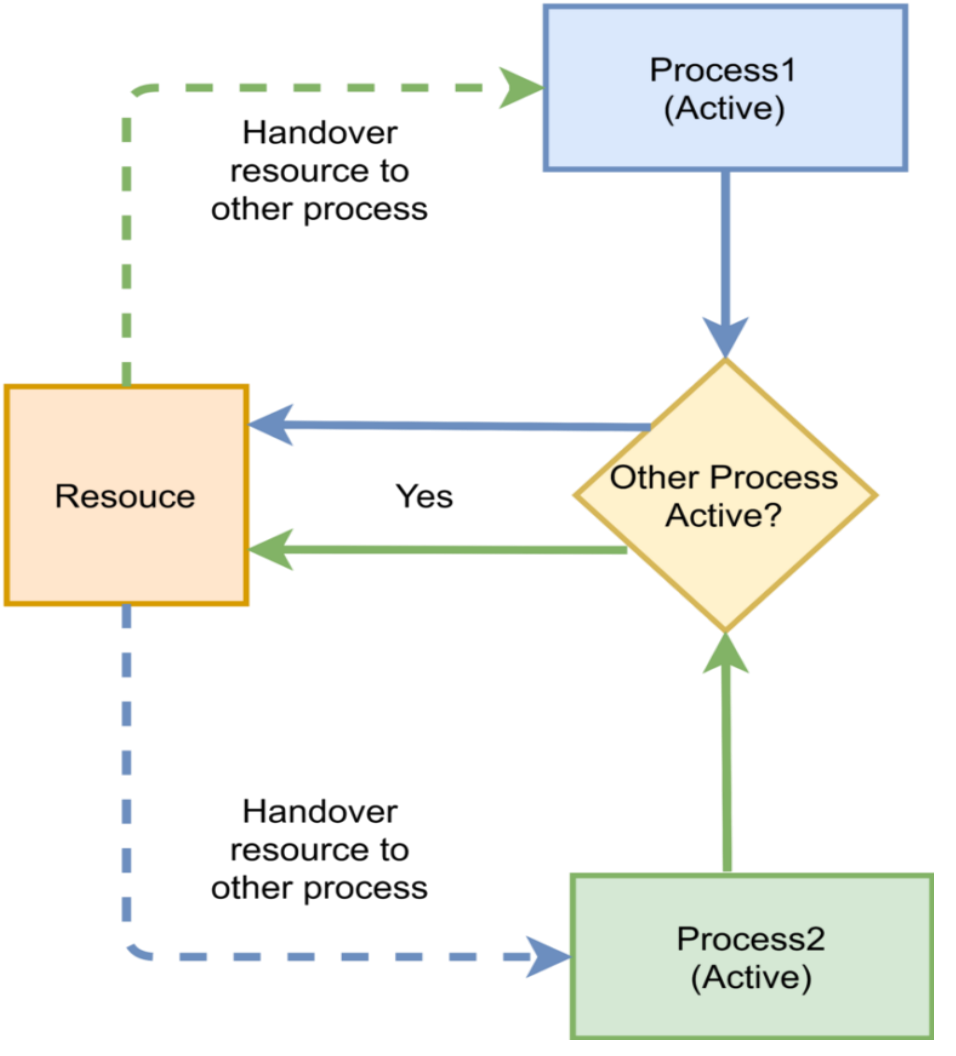
P1 sets flag[1] to false.

This condition is live lock.

This is not deadlock as any alteration in speed will break the sequence.

Live lock

- A common instance of Live lock is that when two persons meet face to face in a corridor, and both move aside to let the other pass. They end up moving in the same direction simultaneously. So, they are failed to cross each other.



Live lock

Live lock

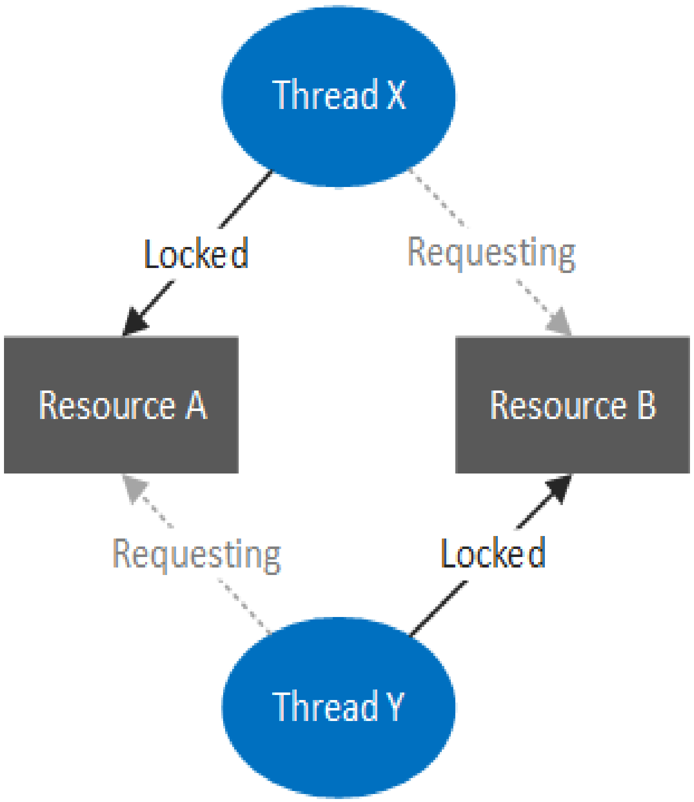
- a scenario in which thread *A* performs an action that causes thread *B* to perform an action that in turn causes thread *A* to perform its original action.
- These processes are not in the waiting state, and they are running concurrently. This is different from a deadlock because in a deadlock all processes are in the waiting state.
- Live lock is a special case of resource **starvation**

Deadlock

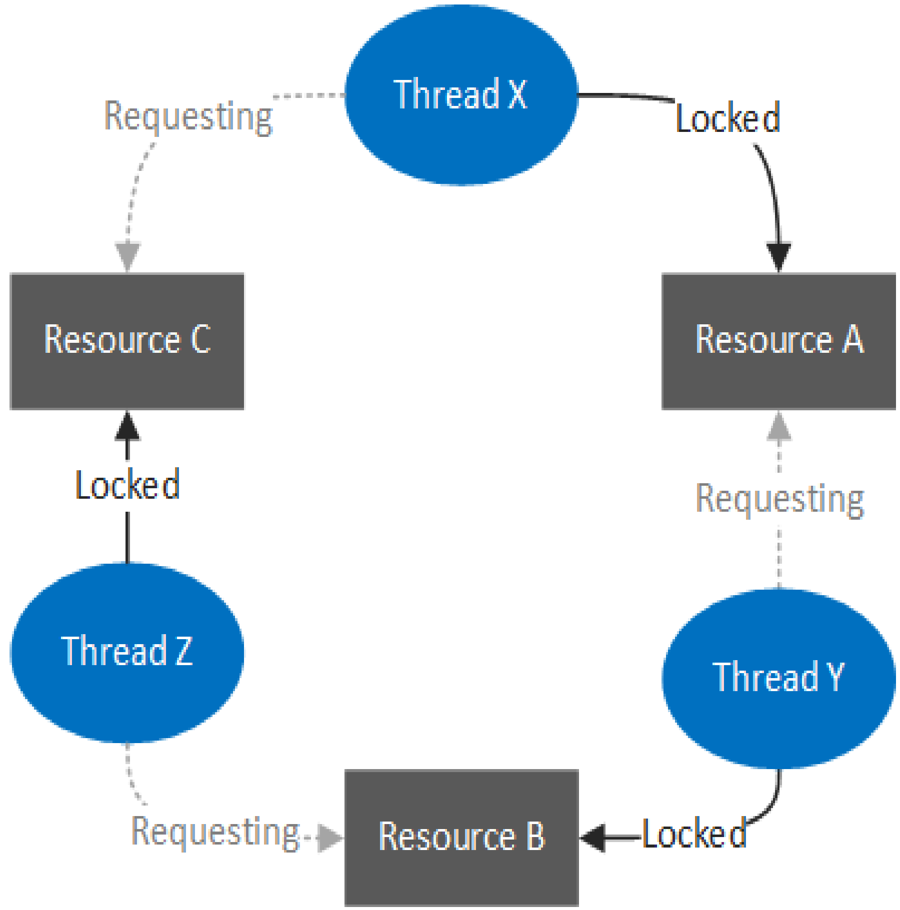
Deadlock :

- a scenario in which one thread/process holds some resource, *A*, and is blocked, waiting for some resource, *B*, to become available, while another thread/process holds resource *B* and is blocked, waiting for resource *A* to become available.
- When a deadlock occurs, no progress is made within a program.

Deadlock



SIMPLE DEADLOCK



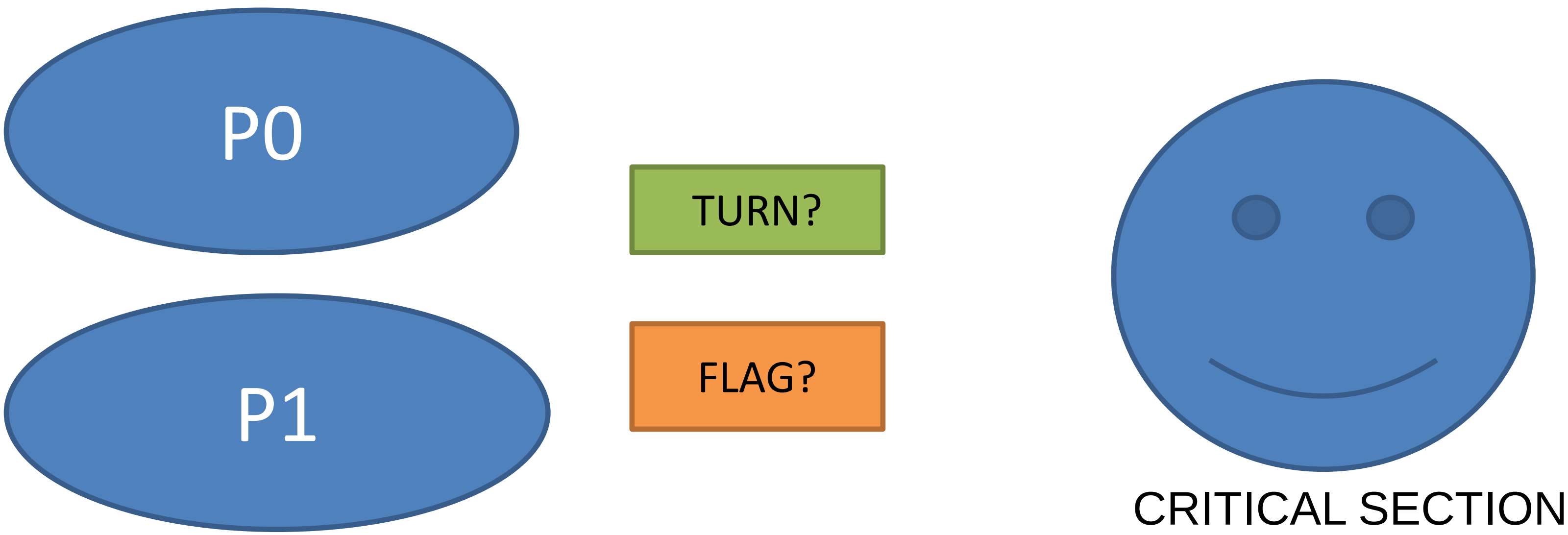
MORE THAN TWO PROCESSES/THREADS IN DEADLOCK

Deadlock vs Livelock

- Livelock is similar to deadlock in that no progress is made but differs in that neither of the process is blocked or waiting.

Correct Solution

- Each process gets a turn at the critical section
- If a process wants the critical section, it sets its flag and may have to wait for its turn



Peterson's Solution

- **Two process solution**
- Assume that the LOAD and STORE instructions are atomic; that is, cannot be interrupted.
- 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.

Algorithm for Process P_i

```
while (true) {
        flag[i] = TRUE;     // Pi is interested in CS
        turn = j;           //  TURN is of Pj
        while ( flag[j] && turn == j); // check status of Pj
           /*do nothing*/
/* CRITICAL SECTION*/

        flag[i] = FALSE;

/* REMAINDER SECTION*/

   }
```

Peterson's Solution

Mutual blocking is prevented.

Global variable flag indicates the position of each process w.r.t mutual exclusion.

Global variable turn resolves simultaniety conflicts.

Semaphores

- The semaphore acts as a guard or lock on the resource.
- The problem of race condition is avoided by not updating a global variable by more than one process at a time.
- Special variable called a semaphore is used for signaling.

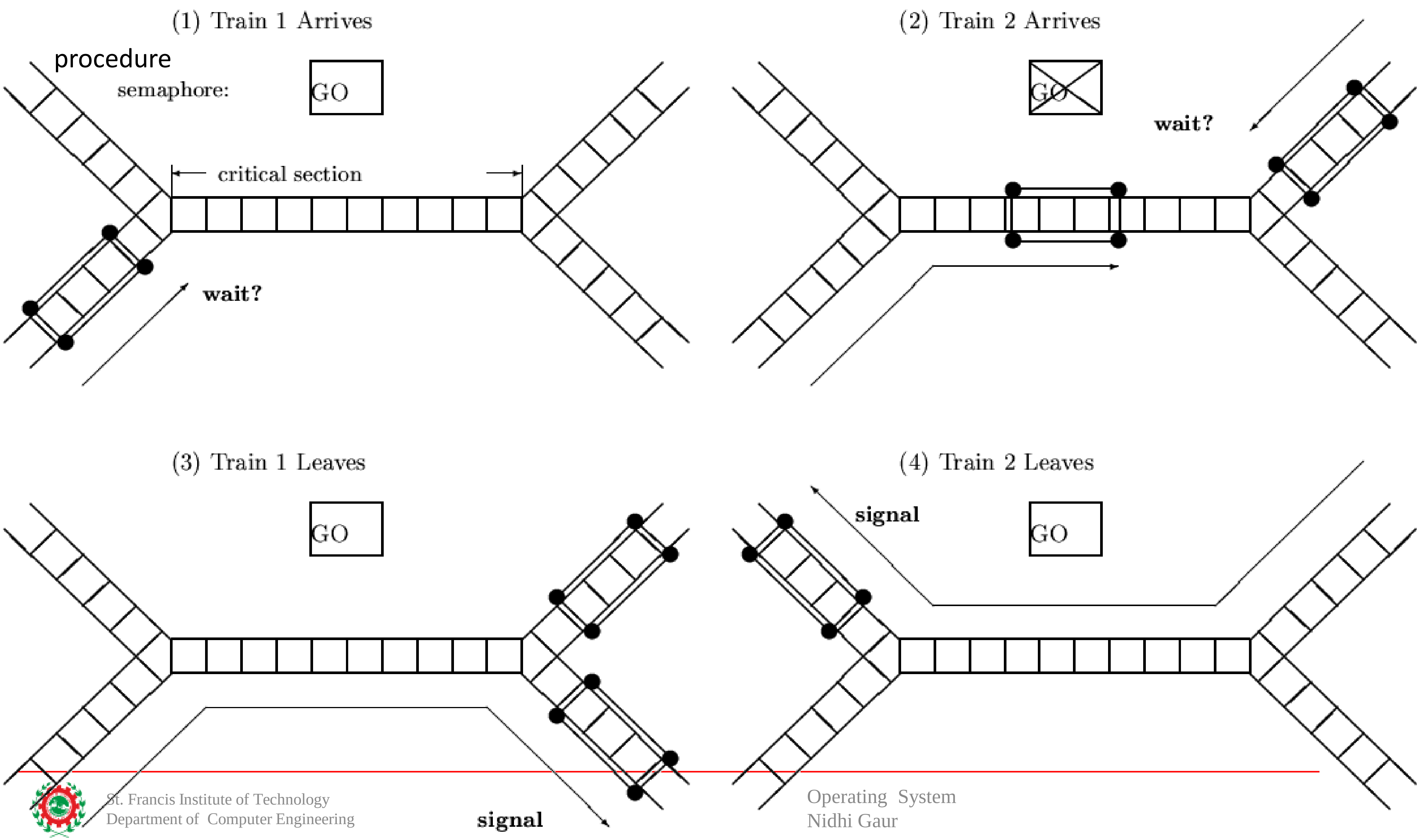
Semaphores

- If a process is waiting for a signal, it is suspended until that signal is sent
- Wait and signal operations cannot be interrupted
- The operations cannot be overlapped or interleaved with the execution of any other operations, known as atomic operations
- Queue is used to hold processes waiting on the semaphore

Semaphores

- Semaphore is a variable that has an integer value
  - May be initialized to a nonnegative number
  - Wait operation decrements the semaphore value
  - Signal operation increments semaphore value

Semaphores


(1) Train 1 Arrives
procedure
semaphore:
GO
critical section
wait?
(2) Train 2 Arrives
GO
wait?
(3) Train 1 Leaves
GO
signal
(4) Train 2 Leaves
signal
GO

Semaphore

- Semaphore *S* – an integer variable
- Two standard operations modify S: wait() and signal()
  - Originally called P() and V()
- 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

Counting semaphore – An integer value can range over an unrestricted domain

Binary semaphore – An integer value can range only between 0 and 1; a resource having single instance can use it.

–**Also mutex is a binary semaphore**, in which process that locks the CS can unlock it. Provides mutual exclusion.

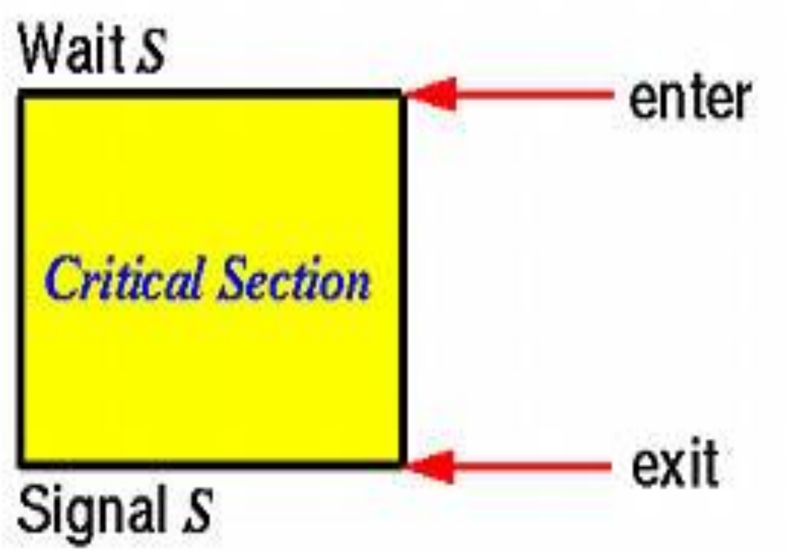
```
–Semaphore S;    //  initialized to 1
–wait (S);
        Critical Section
  signal (S);
```

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.

Semaphores

- Wait may cause a thread to block (*i.e.*, when the counter is zero), it has a similar effect of the lock operation of a mutex lock.
- A Signal may release waiting threads, and is similar to the unlock operation. In fact, semaphores can be used as mutex locks. Consider a semaphore *S* with initial value 1. Then, Wait and Signal correspond to lock and unlock:



Semaphore

- <u>Semaphore: Count</u>

PID 1	PID 2	count
sem=pm_seminit(1);		1
pm_wait(sem);		0
	pm_wait(sem);	-1
(critical)		
pm_signal(sem);		0
pm_wait(sem);		-1
	(critical)	
	pm_signal(sem);	0
	pm_wait(sem);	-1
(critical)		
pm_signal(sem);		0

Semaphores

- Policy should be decided to find out the order in which processes are removed from the queue of semaphore. Fairest algorithm is FIFO.
- A semaphore that specifies the order is called strong otherwise it is weak semaphore.

Producer/Consumer Problem

- One or more producers are generating data and placing these in a buffer
- Only one producer or consumer may access the buffer at any one time

Infinite Buffer

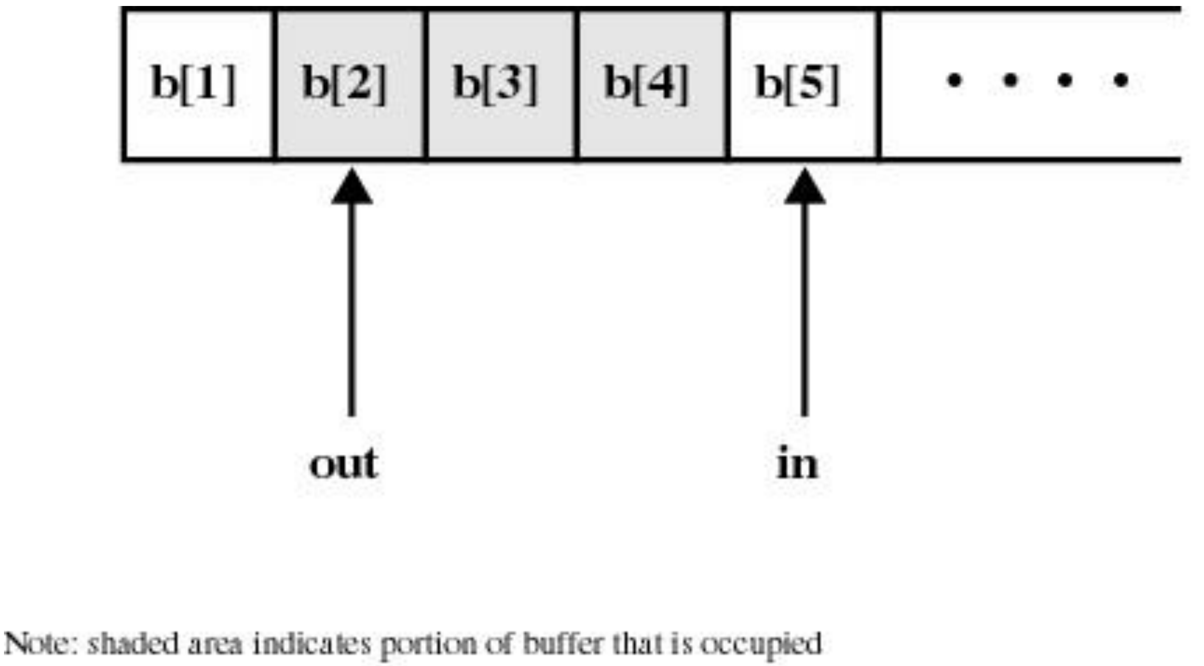


Figure 5.11 Infinite Buffer for the Producer/Consumer Problem

Producer/Consumer

The producer-consumer problem (also known as the bounded-buffer problem) is a classical example of a multi-process synchronization problem

The problem describes two processes, the producer and the consumer, who share a common, fixed-size buffer. The producer's job is to generate a piece of data, put it into the buffer and start again. At the same time the consumer is consuming the data (i.e. removing it from the buffer) one piece at a time. The problem is to make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.

Producer/Consumer

- **The solution for the producer is to go to sleep if the buffer is full.** The next time the consumer removes an item from the buffer, it wakes up the producer who starts to fill the buffer again.
- **In the same way, the consumer goes to sleep if it finds the buffer to be empty**. The next time the producer puts data into the buffer, it wakes up the sleeping consumer. The solution can be reached by means of inter-process communication, typically using semaphores. An inadequate solution could result in a deadlock where both processes are waiting to be awakened.

Producer/Consumer

- Incorrect implementation

```
int itemCount

procedure producer() {
   while (true) {
       item = produceItem()

        if (itemCount == BUFFER_SIZE) {
            sleep()
        }

       putItemIntoBuffer(item)
        itemCount = itemCount + 1

       if (itemCount == 1) {
            wakeup(consumer)
        }
   }
}
```

Producer/Consumer

```
procedure consumer() {
    while (true) {

        if (itemCount == 0) {
            sleep()
        }

        item = removeItemFromBuffer()
        itemCount = itemCount - 1

        if (itemCount == BUFFER_SIZE - 1) {
            wakeup(producer)
        }

        consumeItem(item)
    }
}
```

Producer/Consumer

The problem with this solution is that it contains a race condition that can lead into a deadlock. Consider the following scenario:

- The consumer has just read the variable itemCount, noticed it's zero and is just about to move inside the if-block.
- Just before calling sleep, the consumer is interrupted and the producer is resumed.
- The producer creates an item, puts it into the buffer, and increases itemCount.

```
procedure consumer() {
    while (true) {

        if (itemCount == 0) {
            sleep()
        }

        item = removeItemFromBuffer()
        itemCount = itemCount - 1

        if (itemCount == BUFFER_SIZE - 1) {
            wakeup(producer)
        }

        consumeItem(item)
    }
}
```

Producer/Consumer

- Because the buffer was empty prior to the last addition, the producer tries to wake up the consumer.
- Unfortunately the consumer wasn't yet sleeping, and the wakeup call is lost. When the consumer resumes, it goes to sleep and will never be awakened again. This is because the consumer is only awakened by the producer when itemCount is equal to 1

```
procedure consumer() {
    while (true) {

        if (itemCount == 0) {
            sleep()
        }

        item = removeItemFromBuffer()
        itemCount = itemCount - 1

        if (itemCount == BUFFER_SIZE - 1) {
            wakeup(producer)
        }

        consumeItem(item)
    }
}
```

Producer/Consumer

- The producer will loop until the buffer is full, after which it will also go to sleep. Since both processes will sleep forever, we have run into a deadlock. This solution therefore is unsatisfactory.

Producer/Consumer

- **Using semaphores**
  - Semaphores solve the problem of lost wakeup calls. In the solution we use two semaphores, **fillCount and emptyCount**, to solve the problem**; fillCount is incremented and emptyCount decremented when a new item has been put into the buffer**.
  - If the producer tries to decrement emptyCount while its value is zero, the producer is put to sleep(bcoz Buffer is full).
  - The next time an item is consumed, emptyCount is incremented and the producer wakes up. The consumer works analogously.

Producer/Consumer


```
semaphore fillCount = 0
semaphore emptyCount = BUFFER_SIZE

procedure producer() {
    while (true) {
        item = produceItem()
        down(emptyCount)
        putItemIntoBuffer(item)
        up(fillCount)
    }
}

procedure consumer() {
    while (true) {
        down(fillCount)
        item = removeItemFromBuffer()
        up(emptyCount)
        consumeItem(item)
    }
}
```

Producer/Consumer

- **The solution above works fine when there is only one producer and consumer**. Unfortunately, with multiple producers or consumers this solution contains a serious race condition that could result in two or more processes reading or writing into the same slot at the same time.

  If the procedure can be executed concurrently by multiple producers, the following scenario is possible:

  - Two producers decrement emptyCount
  - One of the producers determines the next empty slot in the buffer . Second producer determines the next empty slot and gets the same result as the first producer
  - Both producers write into the same slot

Producer/Consumer

- To overcome this problem, we need a way to make sure that only one producer is executing **putItemIntoBuffer()** at a time. In other words we need a way to execute a critical section with mutual exclusion. To accomplish this we use a binary semaphore called mutex. Since the value of a binary semaphore can be only either one or zero, only one process can be executing between down(mutex) and up(mutex).

Producer/Consumer

The solution for multiple producers and consumers is given below:

There are three semaphores.

- ***Full***, used for counting the number of slots that are full;
- ***empty***, used for counting the number of slots that are empty;
- ***mutex,*** used to enforce mutual exclusion.

Producer/Consumer

```
BufferSize = 3;

 semaphore mutex = 1;           // Controls access to critical section
 semaphore empty = BufferSize;  // counts number of empty buffer slots
 semaphore full = 0;            // counts number of full buffer slots

 Producer()
 {
   int widget;

   while (TRUE) {               // loop forever
     make_new(widget);          // create a new widget to put in the buffer
     down(&empty);              // decrement the empty semaphore
     down(&mutex);              // enter critical section
     put_item(widget);          // put widget in buffer
     up(&mutex);                // leave critical section
     up(&full);                 // increment the full semaphore
     }
 }
```

Producer/Consumer

```
Consumer()
 {
   int widget;

   while (TRUE) {                 // loop forever
     down(&full);                 // decrement the full semaphore
     down(&mutex);                // enter critical section
     remove_item(widget);         // take a widget from the buffer
     up(&mutex);                  // leave critical section
     up(&empty);                  // increment the empty semaphore
     consume_item(widget);        // consume the item
     }
 }
```

Problems with Semaphores

- Incorrect use of semaphore operations:
  - signal (mutex) …. wait (mutex)
  - wait (mutex) … wait (mutex)
  - Omitting of wait (mutex) or signal (mutex) (or both)