Operating Systems

Chapter 6: Process Synchronization

Dr. Ahmed Hagag

Scientific Computing Department,
Faculty of Computers and Artificial Intelligence
Benha University

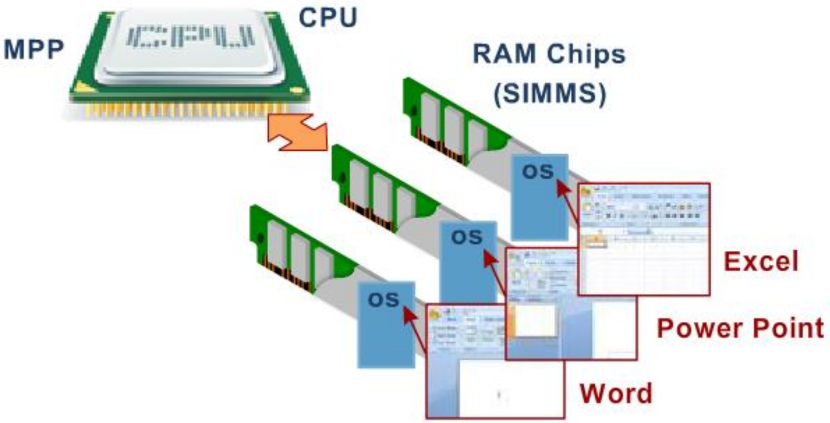
2019

Ch. 6: Process Synchronization

- Introduction.
- The Critical Section Problem.
- Solution to The Critical Section Problem.
    - ➢ Software Solution.
    - ➢ Hardware Solution.

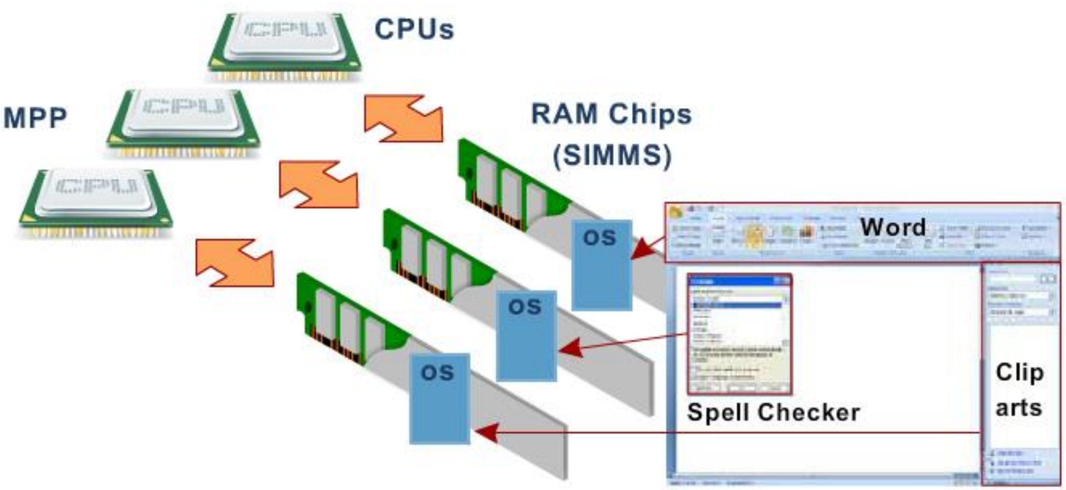
Introduction (1/10)

- The operating system supports multiple processes running in parallel that are the results from the following:
  - ➢ **Multiple applications:** Multi-programming allows processor time to be shared among a number of active applications.



Introduction (2/10)

- The operating system supports multiple processes running in parallel that are the results from the following:
  - **Structured application:** some applications can be implemented as a set of *concurrent processes*.



Introduction (3/10)

- A **cooperating process** is one that can affect or be affected by other processes executing in the system.
- Cooperating processes can either directly share a logical address space (that is, both code and data) or be allowed to share data only through files or messages. The former case is achieved through the use of threads, discussed in Chapter 4.

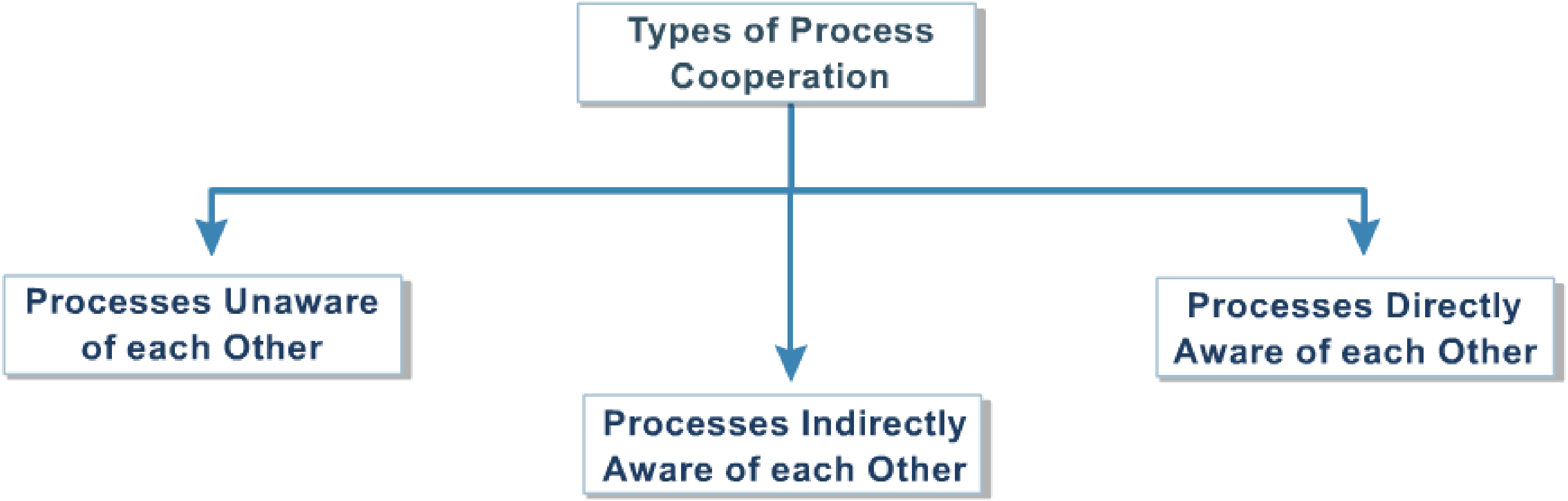
Introduction (4/10)

- Concurrent access to shared data may result in data inconsistency, however. In this chapter, we discuss various mechanisms to ensure the orderly execution of cooperating processes that share a logical address space, so that data consistency is maintained.

Introduction (5/10)

Types of Process Cooperation

- Processes interact with each other can be:



Types of Process Cooperation

Processes Unaware of each Other

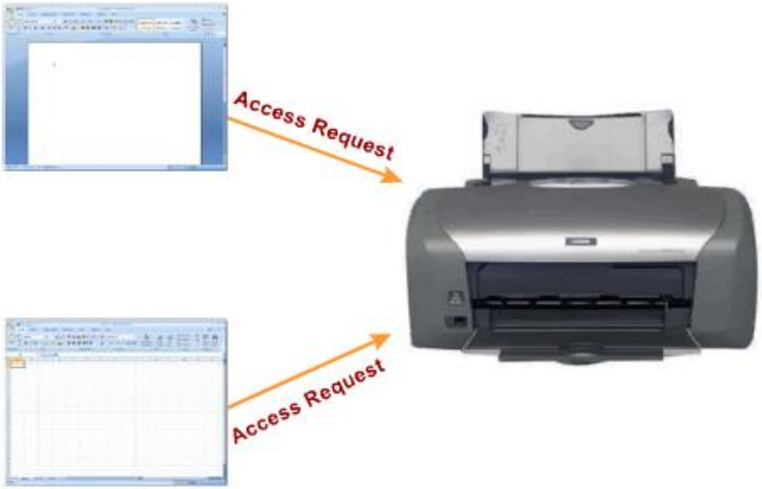
Processes Indirectly Aware of each Other

Processes Directly Aware of each Other

Introduction (6/10)

Processes unaware of each other (competition) (1/3)

- Those are independent processes that are not intended to work together.
- For example, in a single user environment, two independent applications may both want to access the same disk, file or printer.



Introduction (6/10)

Processes unaware of each other (competition) (2/3)

- In a multi-user environment, users are concurrent, use the same web server and need access to the same web page. A person can handle several tasks at once (have you every listened to music while doing other work and heard the phone ring?).

Access Request

Introduction (6/10)

Processes unaware of each other (competition) (3/3)

- The OS must resolve the competition for resources.
  - ➢ I/O, memory, printer, tape drive, etc.
- Each process should leave the state of any resource that it uses unaffected.

Introduction (7/10)

Processes indirectly aware of each other (cooperation by sharing) (1/2)

- These are processes that are not necessarily aware of each other but they share the same resources. Like for example, I/O buffers.

Introduction (7/10)

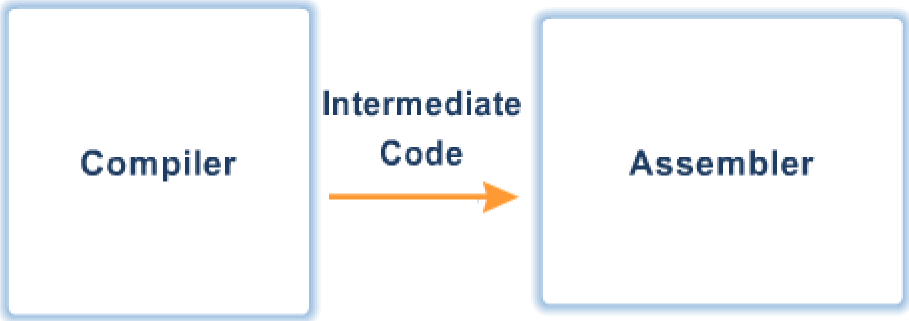
Processes indirectly aware of each other (cooperation by sharing) (2/2)

- Types of interactions could be in the form of shared access to some object:
  - ➢ shared variables, files, or databases
- Processes may use and update the shared data without reference to other process, but know that other processes may have access to the same data.

Introduction (8/10)

Process directly aware of each other(cooperation by communication) (1/2)

- These are processes that can work jointly on the same activity and can communicate with each other through their **process ID**.



Introduction (8/10)

Process directly aware of each other(cooperation by communication) (2/2)

- For example, when we have a process that produces an output that is considered an input for other processes, Interpose communication exists.
- This can be accomplished through sending and receiving of messages: This can be viewed when a compiler generates the intermediate code for the assemble to be processed and transferred to machine language during the compilation process.

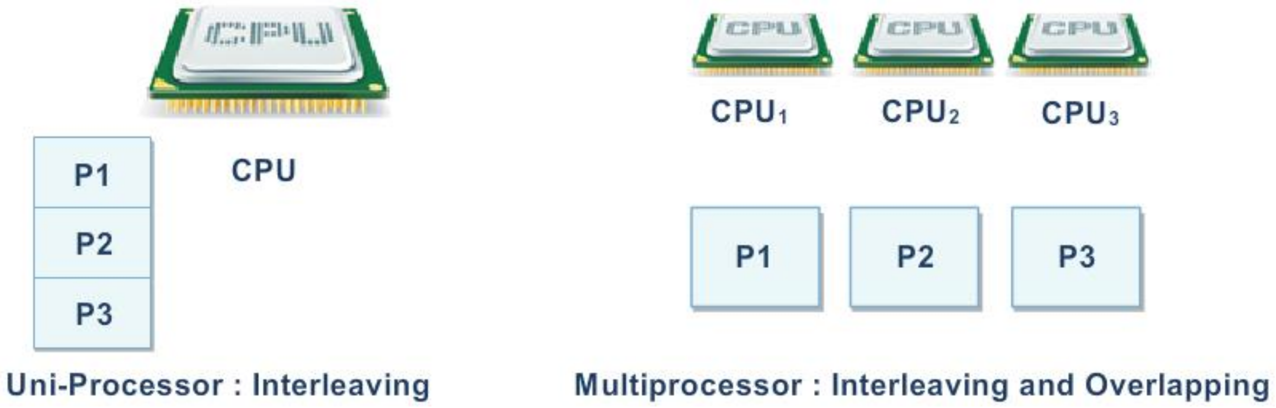
Introduction (9/10)

Definition of concurrency problem (1/4)

- Traditionally, the expression of a task in the form of multiple, possibly interacting subtasks, that may potentially be executed at the same time constitutes concurrency.
  - **Uniprocessor:** interleaving.
  - **Multiprocessor:** interleaving and overlapping.

Introduction (9/10)

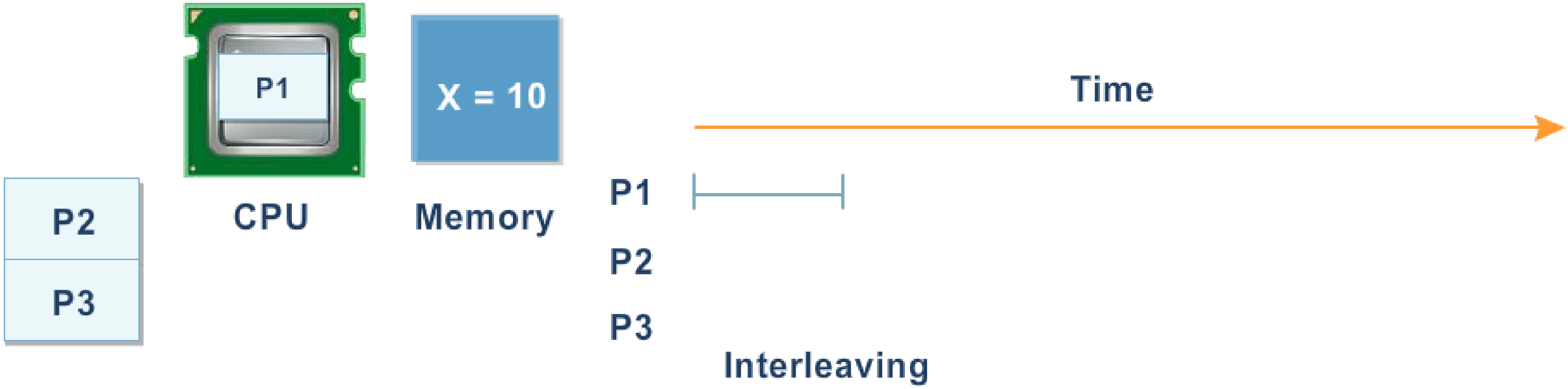
Definition of concurrency problem (2/4)


CPU
P1
P2
P3
Uni-Processor : Interleaving
CPU1
CPU2
CPU3
P1
P2
P3
Multiprocessor : Interleaving and Overlapping

Introduction (9/10)

Definition of concurrency problem (3/4)

- For the uni-processor environments, concurrent access to shared data appears when the task is split between three processes each of them use the process in a series which may result in data inconsistency.



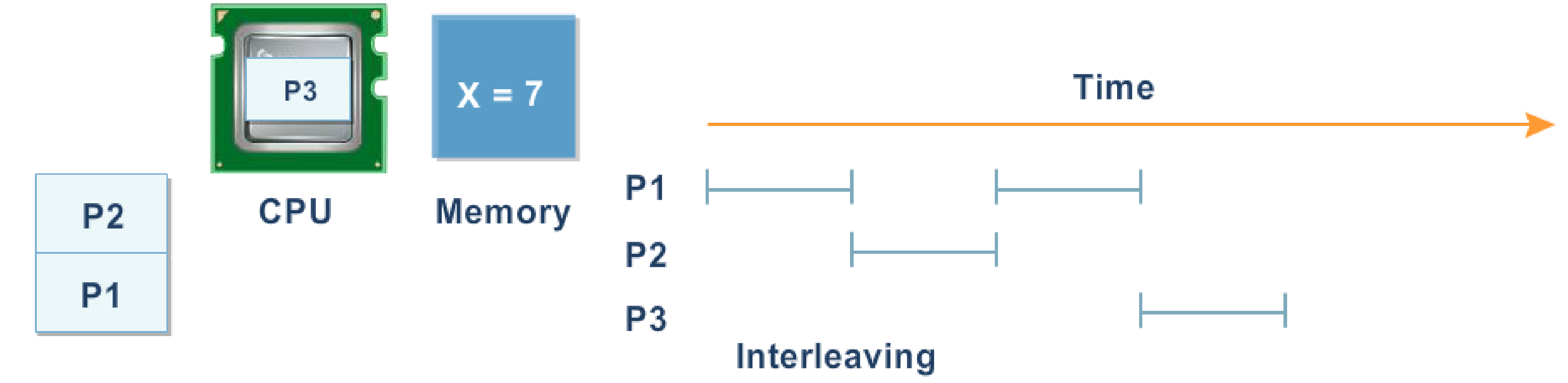
Uni-Processor : Interleaving

The Execution of Concurrent Processes

Introduction (9/10)

Definition of concurrency problem (3/4)

- For the uni-processor environments, concurrent access to shared data appears when the task is split between three processes each of them use the process in a series which may result in data inconsistency.



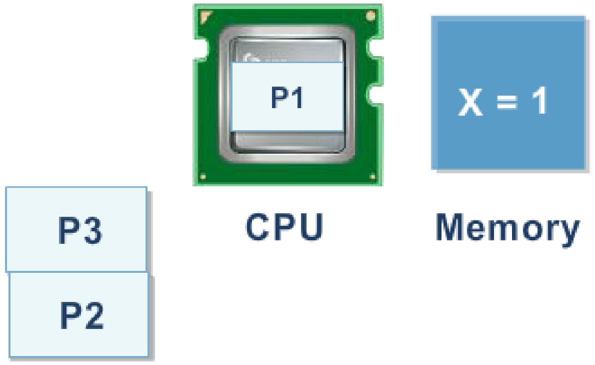
Uni-Processor : Interleaving

The Execution of Concurrent Processes

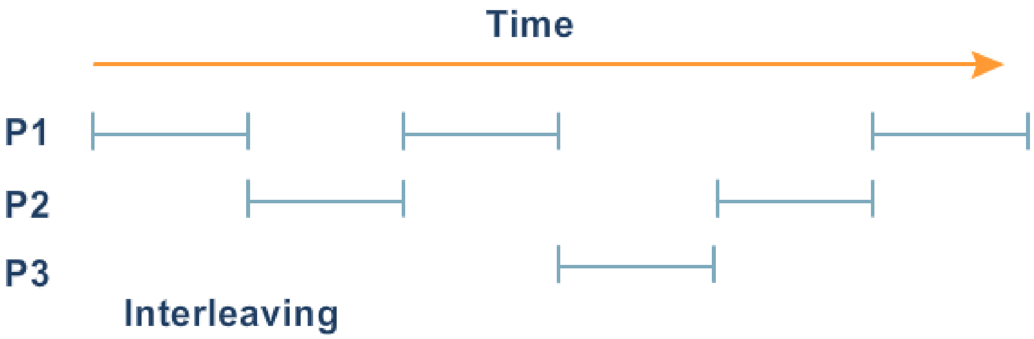
Introduction (9/10)

Definition of concurrency problem (3/4)

- For the uni-processor environments, concurrent access to shared data appears when the task is split between three processes each of them use the process in a series which may result in data inconsistency.



Uni-Processor : Interleaving

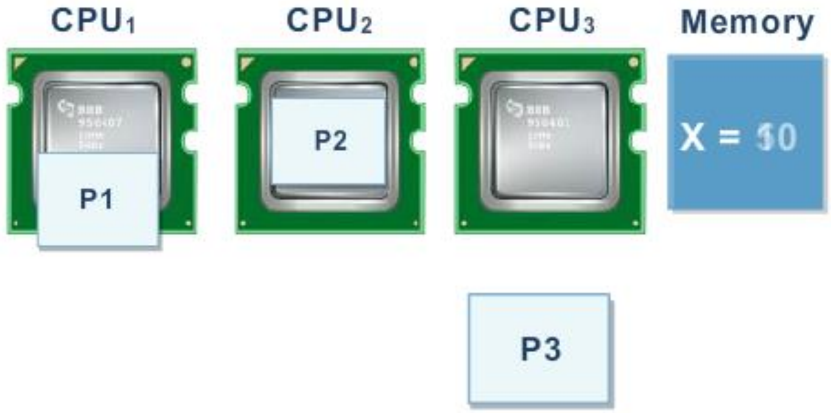


The Execution of Concurrent Processes

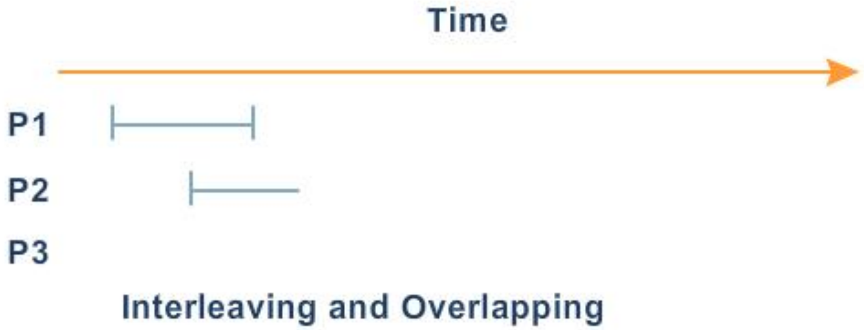
Introduction (9/10)

Definition of concurrency problem (4/4)

- Also in multi-processor environment, tasks are run on different processors in parallel which may lead to overlapping and data inconsistency.



Multiprocessor : Interleaving and Overlapping

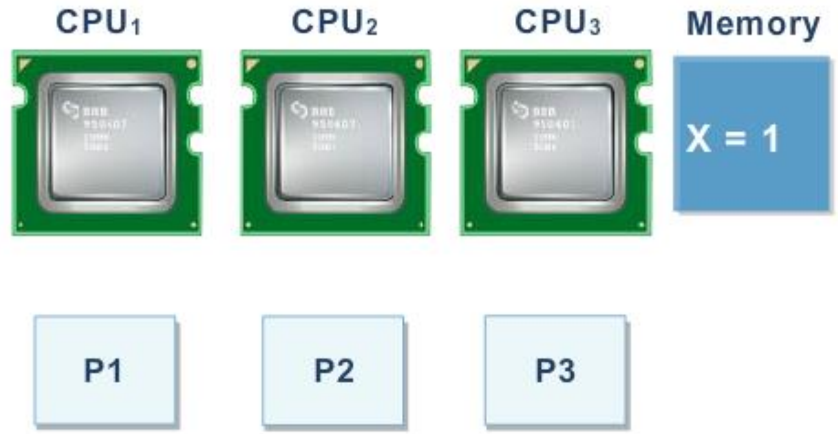


The Execution of Concurrent Processes

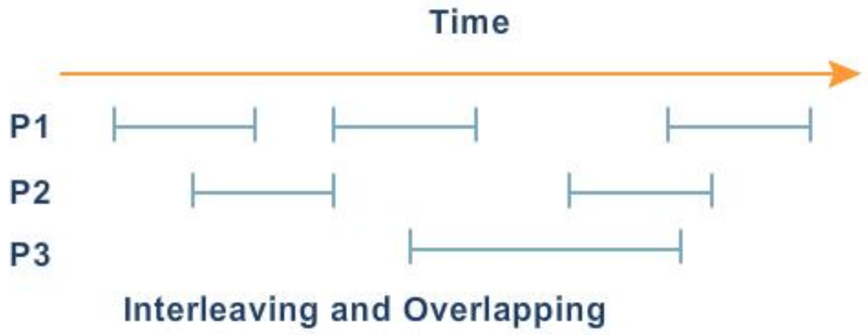
Introduction (9/10)

Definition of concurrency problem (4/4)

- Also in multi-processor environment, tasks are run on different processors in parallel which may lead to overlapping and data inconsistency.



Multiprocessor : Interleaving and Overlapping



The Execution of Concurrent Processes

Introduction (10/10)

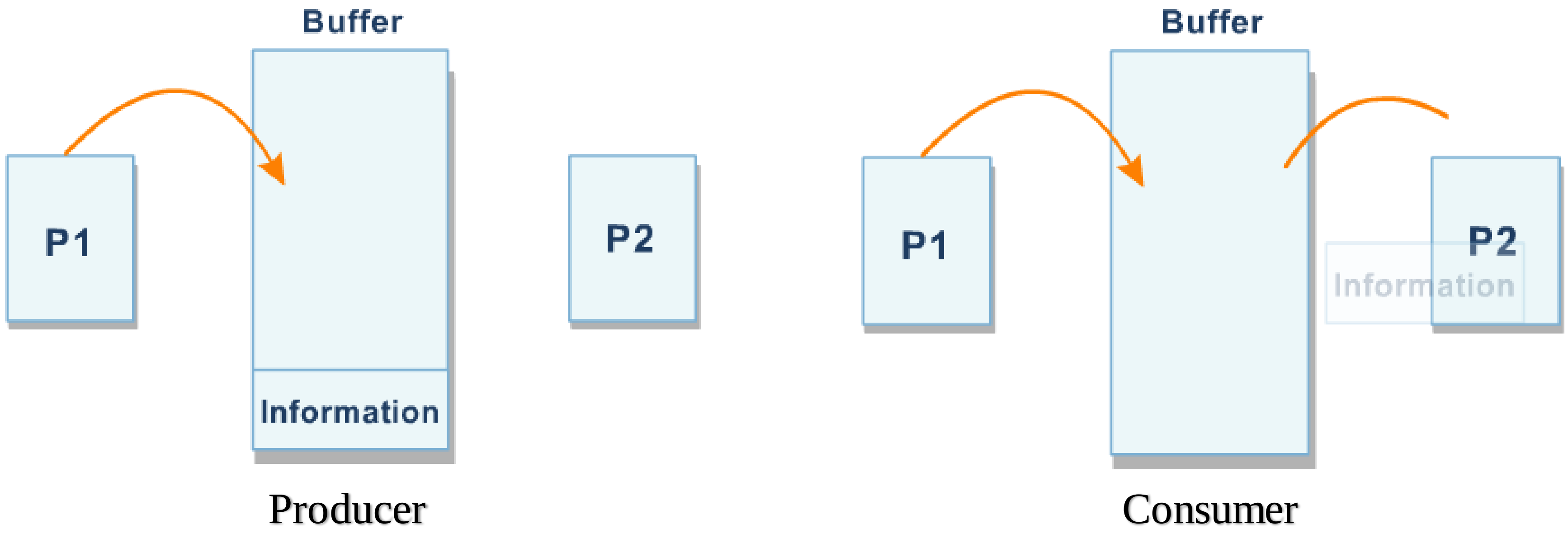
The producer-consumer problem (1/10)

- The producer-consumer problem illustrates the need for synchronization in systems where many processes share a resource. In this problem, two processes share a fixed-size (bounded) buffer.

Introduction (10/10)

The producer-consumer problem (2/10)

- One process **produces** information and puts it in the buffer, while the other process **consumes** information from the buffer. These processes do not take turns accessing the buffer, they both work concurrently.



Introduction (10/10)

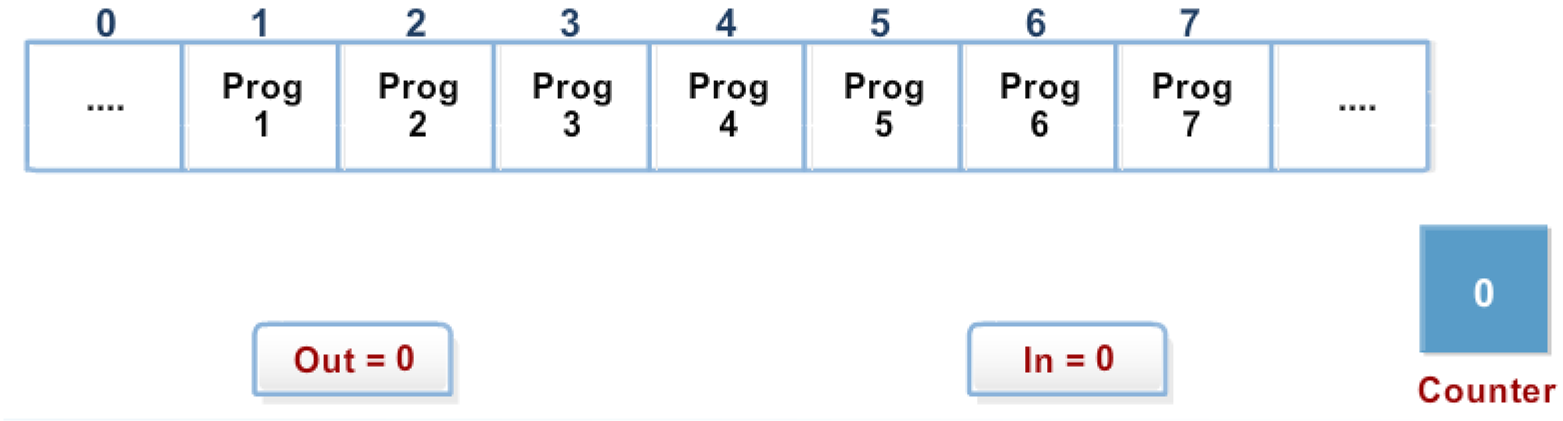
The producer-consumer problem (3/10)

- Here in lays the problem:
  - ➢ What happens if the producer tries to put an item into a full buffer?
  - ➢ What happens if the consumer tries to take an item from an empty buffer?

Introduction (10/10)

The producer-consumer problem (4/10)

- Suppose that we want to provide a solution to the consumer-producer problem that fills all the cells of the buffer:
  - ➢ We introduce an **integer counter** that keeps track of the number of full buffers.



0	1	2	3	4	5	6	7	
....	Prog 1	Prog 2	Prog 3	Prog 4	Prog 5	Prog 6	Prog 7	....

Out = 0

In = 0

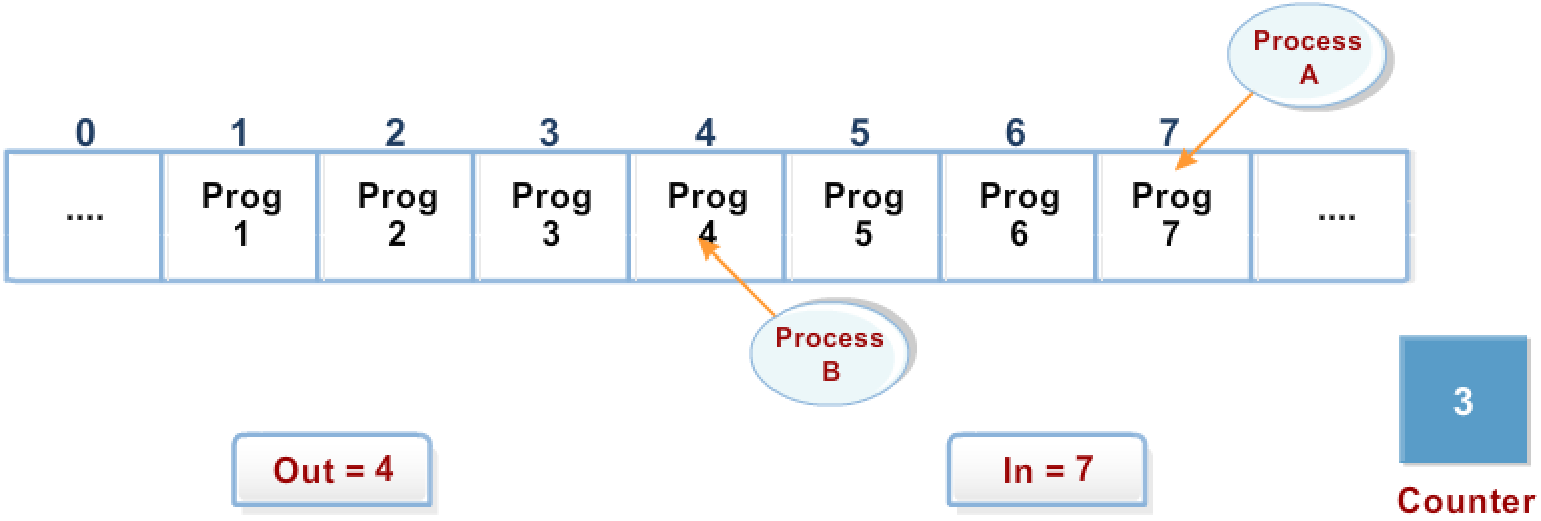
0

Counter

Introduction (10/10)

The producer-consumer problem (5/10)

- 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.



Introduction (10/10)

The producer-consumer problem (6/10)

Producer

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

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

Introduction (10/10)

The producer-consumer problem (7/10)

Consumer

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

The producer-consumer problem (8/10)

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

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

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

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

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

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

Introduction (10/10)

The producer-consumer problem (9/10)

- The value of count should be 5 … but is 4 (after the Consumer process updates count). It could also be 6, if the execution order of the last two steps is reversed.
- This will cause "problems" with the data, since the Producer process thinks that there are more free cells than there really are.

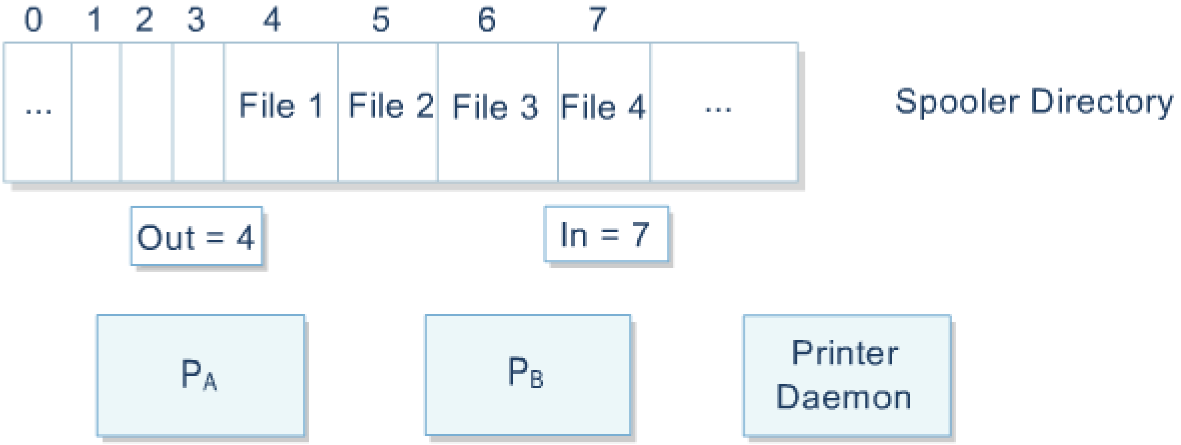
Introduction (10/10)

The producer-consumer problem (10/10)

- We can identify **two critical sections**, one being **count++** (in the Producer) the other **count– –** (in the Consumer). Both must be "protected", in order to ensure correct execution.

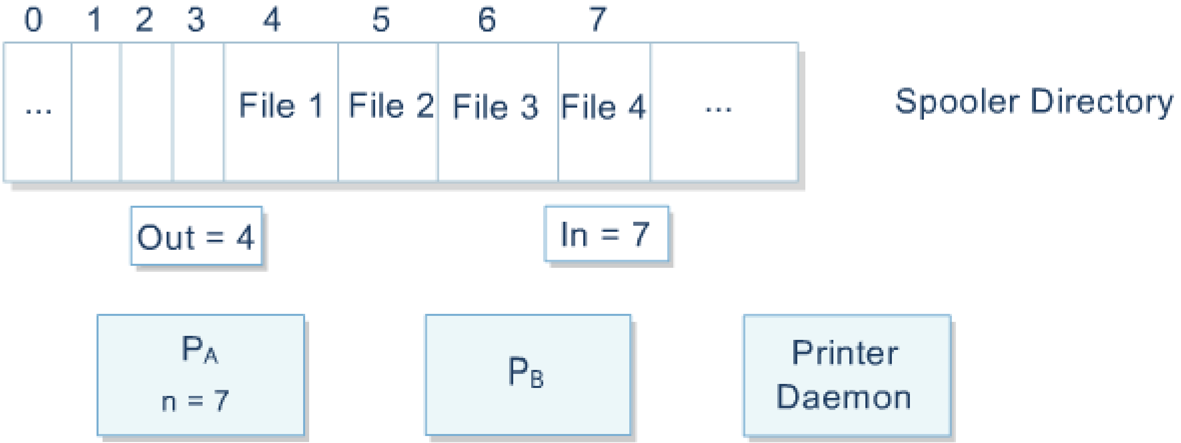
Critical Section Problem (1/14)

- The problem has a Spooler directory with **slots**; **index variable**; **two processes** attempt concurrent access.
- According to the figure, at a certain time, slot 0 to 3 are empty and slot 4 to 7 are full. Process A and B simultaneously want to queue a file for printing.



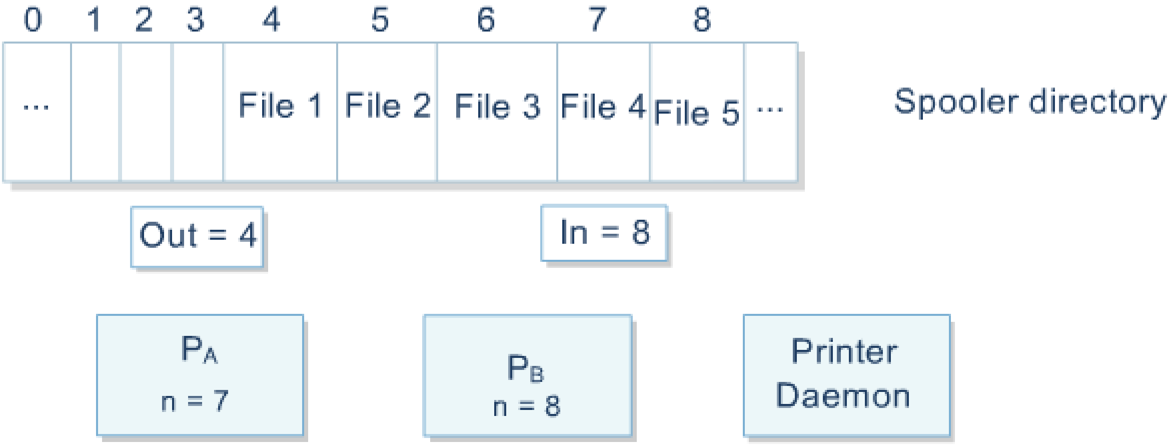
Critical Section Problem (2/14)

- Process A reads "in" with the value 7 and is switched to Process B.



Critical Section Problem (3/14)

- Process B reads "in", update it to 8 puts its product in the 'in' slot, increases its local in by one and sets in.



0	1	2	3	4	5	6	7	8	
...				File 1	File 2	File 3	File 4	File 5	...

Spooler directory

Out = 4

In = 8

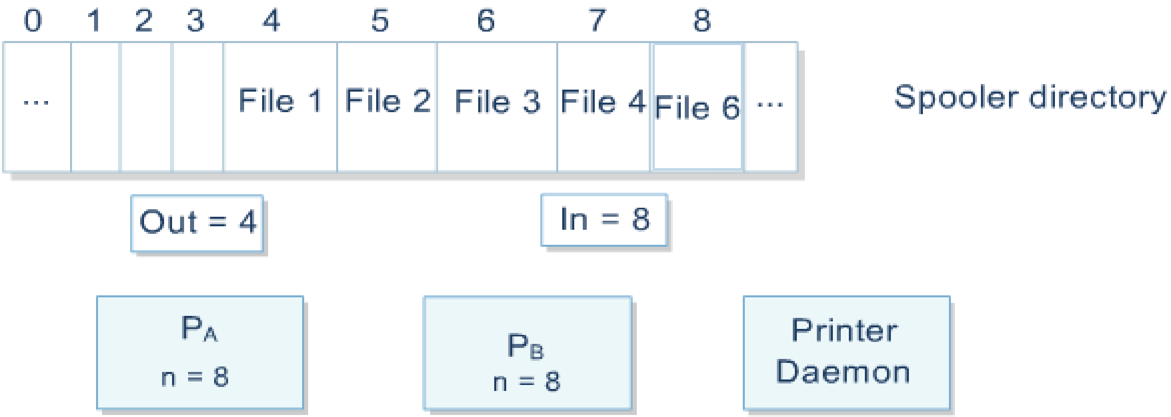
P_A
n = 7

P_B
n = 8

Printer Daemon

Critical Section Problem (4/14)

- Process A gets the control again. Enters its product to the 'in' slot, increments "its in" and sets in.
- Result: "in" has a correct value but **the printing job of B is lost**….
- In this situation, the value of the shared variable depends on who runs first and update the value which is called: Race condition.



0	1	2	3	4	5	6	7	8	
···				File 1	File 2	File 3	File 4	File 6	···

Spooler directory

Out = 4

In = 8

P_A
n = 8

P_B
n = 8

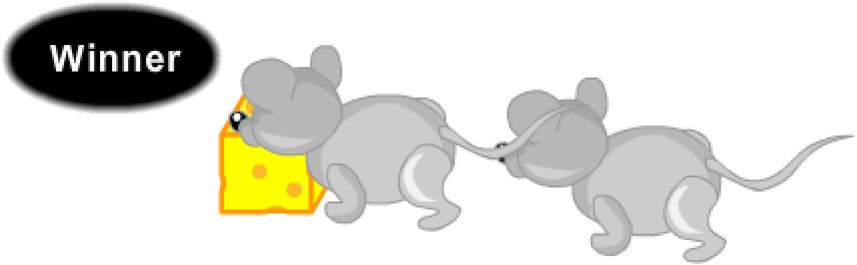
Printer Daemon

Critical Section Problem (5/14)

Description of Race Condition

Critical Section Problem (5/14)

Description of Race Condition



Critical Section Problem (6/14)

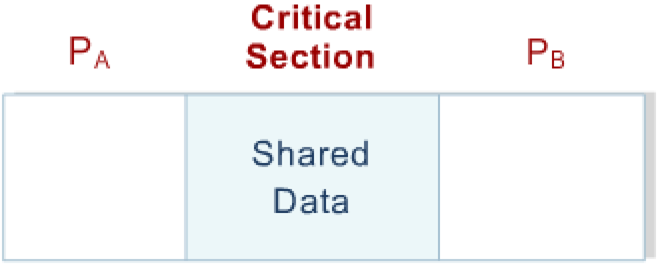
Race Condition

- **Race condition** is: A situation in which multiple processes read and write a shared data item and the final result depends on the relative timing of their execution.

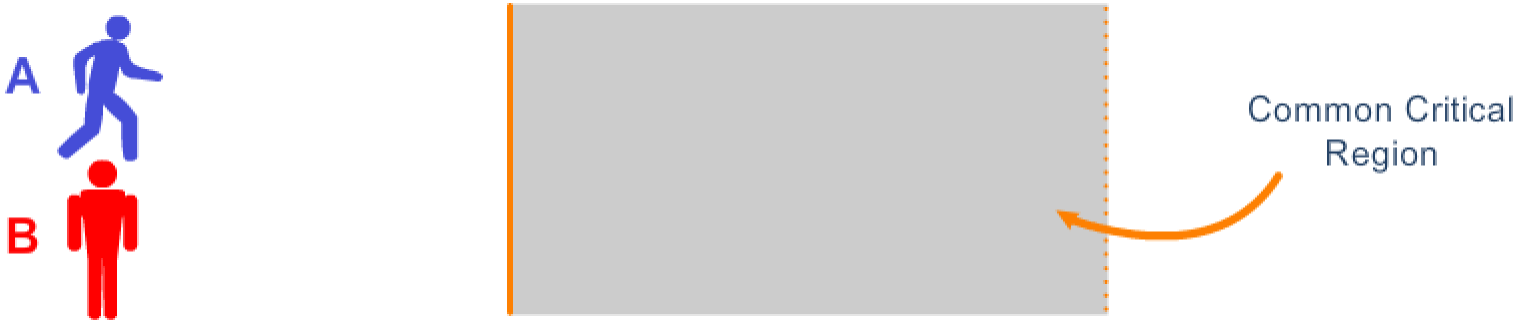
- Generally, the race conditions may occur whenever resources and/or data are shared (by processes unaware of each other or processes indirectly aware of each other).

Critical Section Problem (7/14)

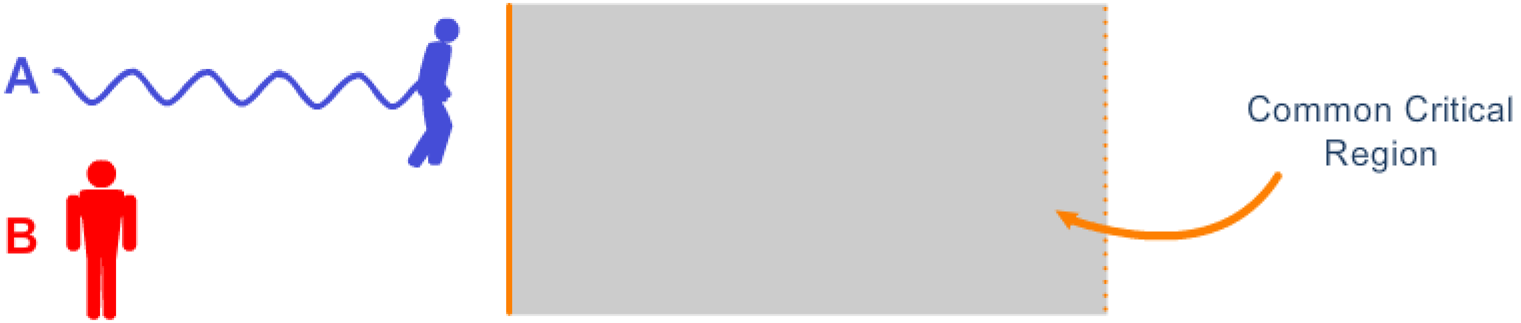
- Consider system of ***n*** processes $\{p_0, p_1, \ldots p_{n-1}\}$
- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc.
  - When one process in critical section, no other may be in its critical section.



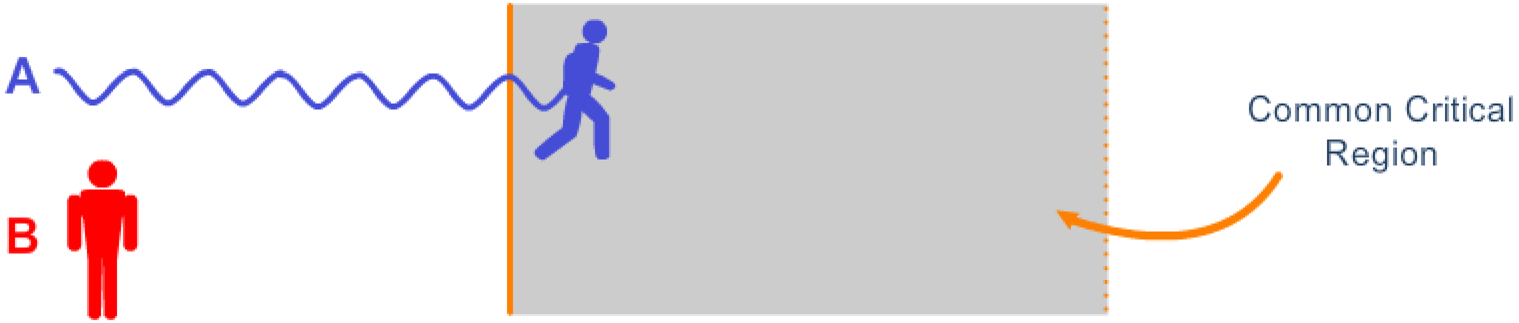
Critical Section Problem (8/14)



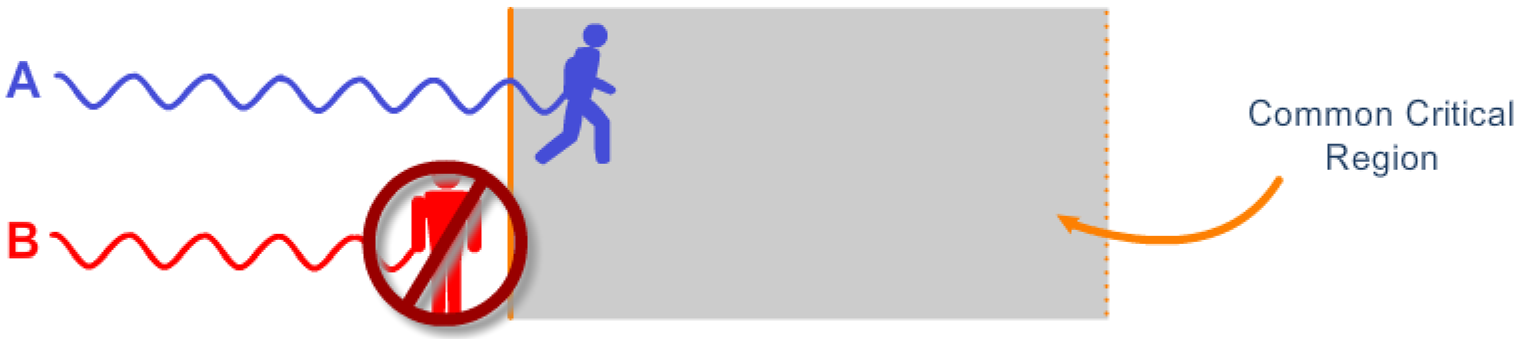
Critical Section Problem (8/14)



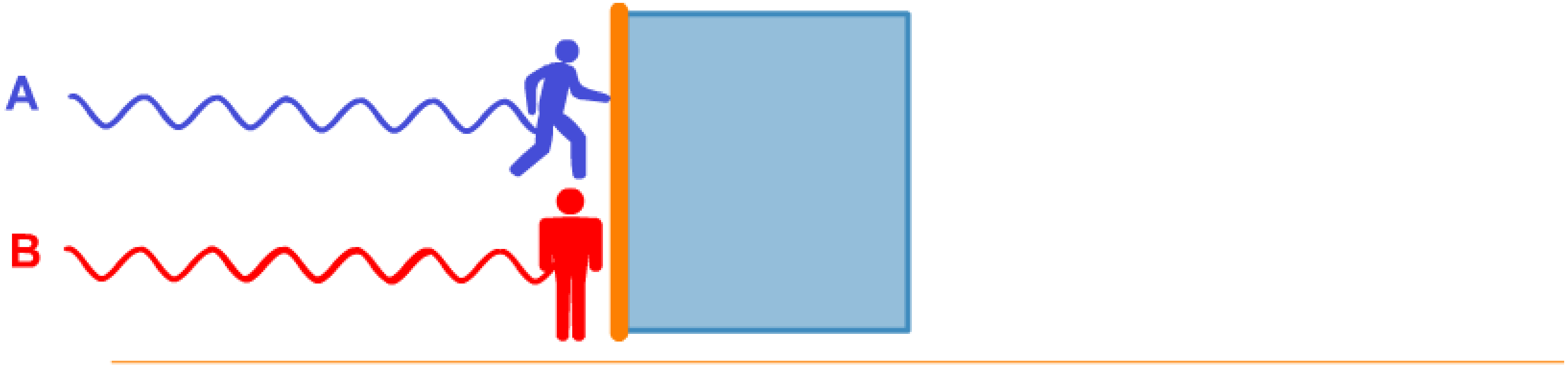
Critical Section Problem (8/14)



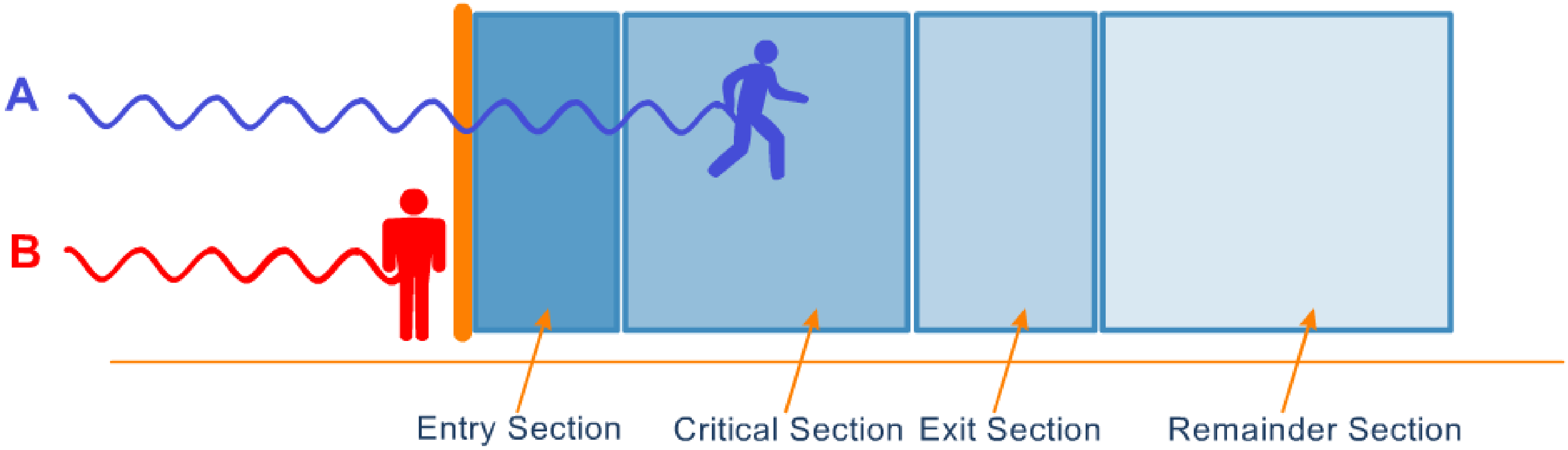
Critical Section Problem (8/14)



Critical Section Problem (9/14)

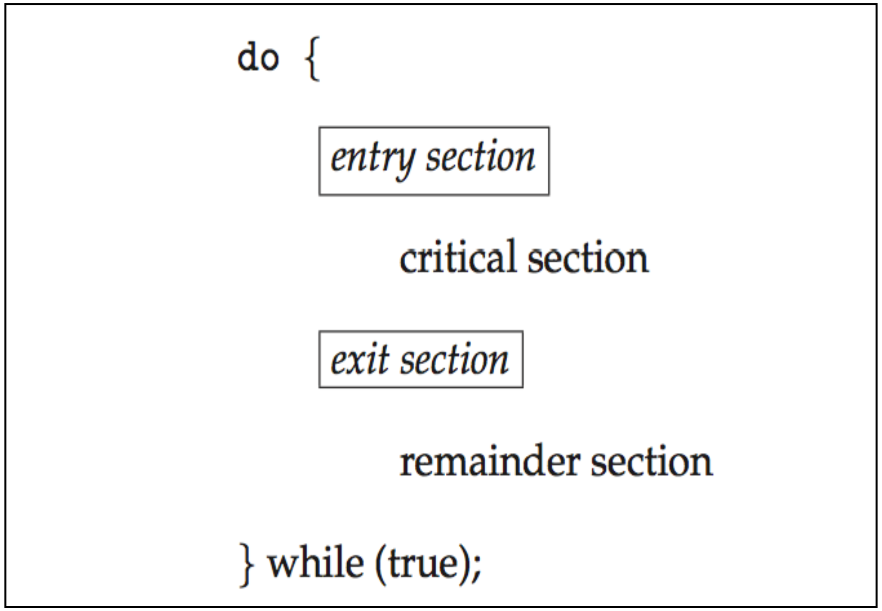


Critical Section Problem (9/14)



Critical Section Problem (10/14)

- ***Critical section problem*** is to design protocol to solve this
- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**.

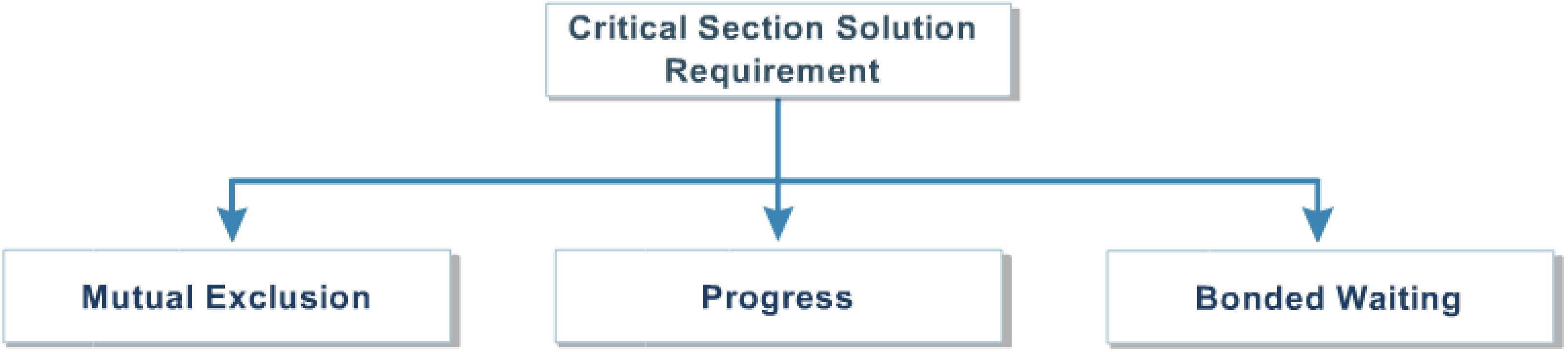


```
do {
        entry section
            critical section
        exit section
            remainder section
} while (true);
```

Critical Section Problem (11/14)

Solutions to the critical section problem

- A solution to the critical section problem must satisfy the following requirements:


Critical Section Solution Requirement

Mutual Exclusion

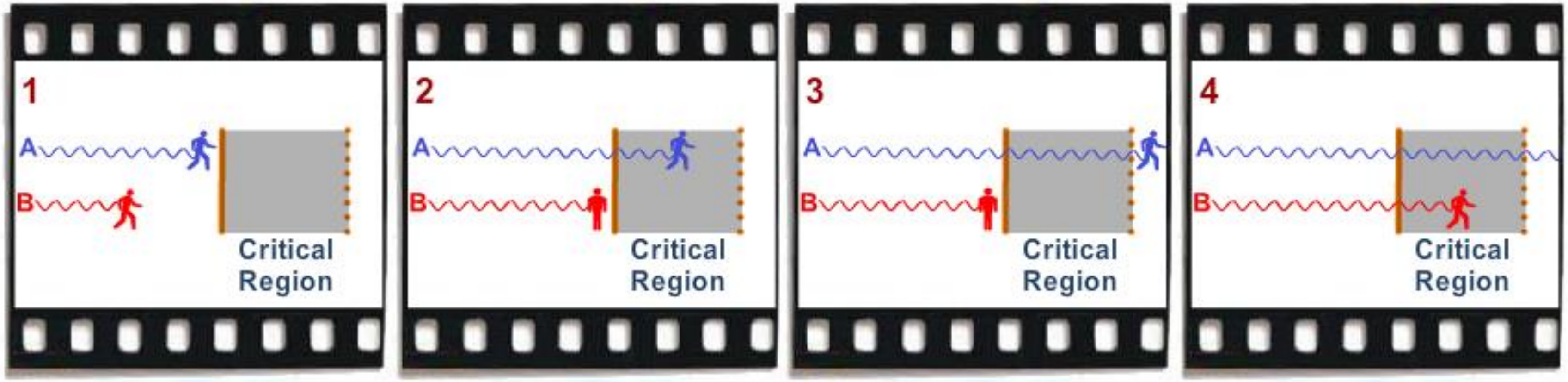
Progress

Bonded Waiting

Critical Section Problem (12/14)

Mutual Exclusion

- **Mutual Exclusion:** If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.



Critical Section Problem (13/14)

Progress

- **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.

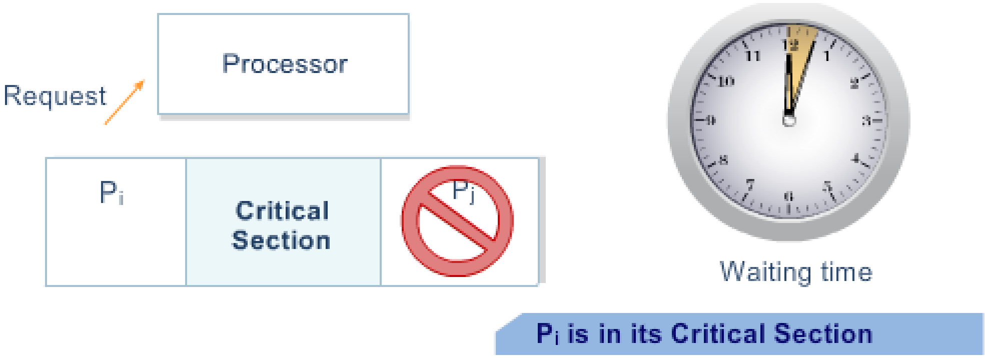
Critical Section Problem (14/14)

Bonded Waiting (1/3)

- **Bonded Waiting:** There exists a bound, or limit, 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. (Starvation).
  - ➢ Assume that each process executes at a nonzero speed.
  - ➢ No assumption concerning **relative speed** of the *n* processes.

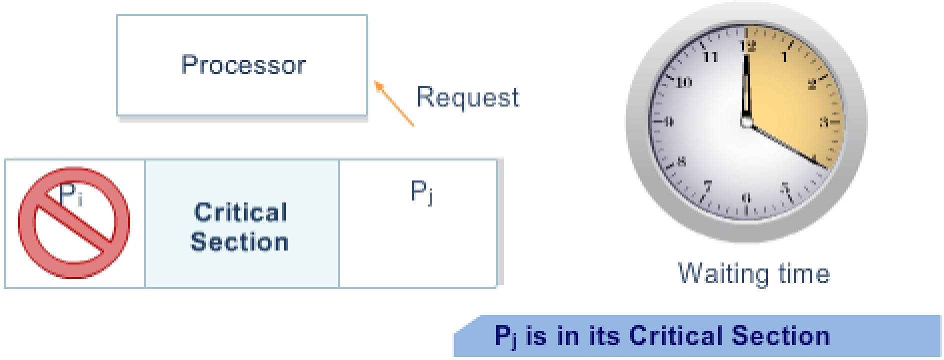
Critical Section Problem (14/14)

Bonded Waiting (2/3)



Critical Section Problem (14/14)

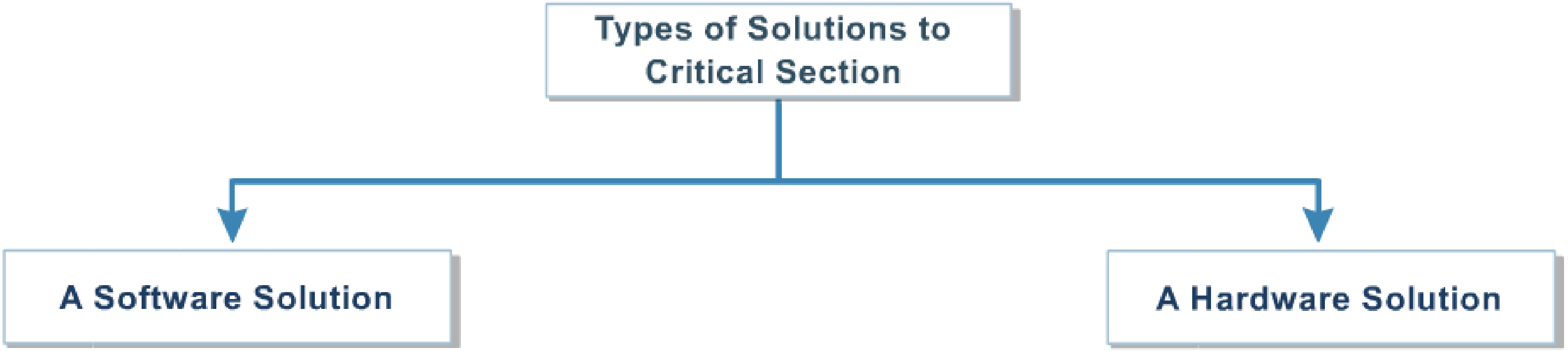
Bonded Waiting (3/3)



Solution to The CS Problem (1/9)

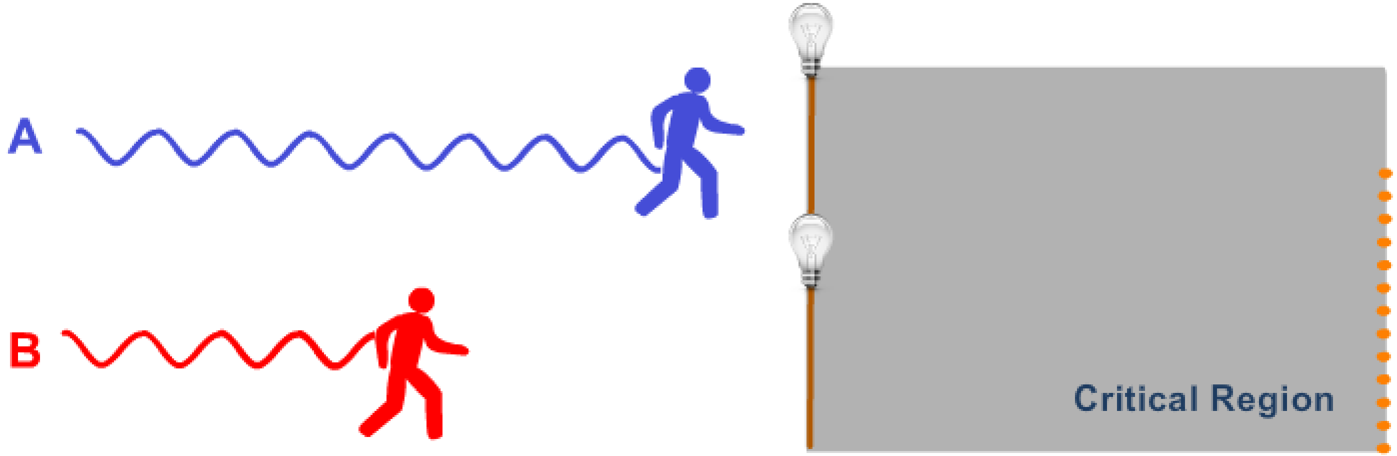
Types of Solutions

- A **software solution** is a code that should be run before entering the critical section to avoid race condition.
- A **hardware solution** is accomplished by utilizing one of the system components.



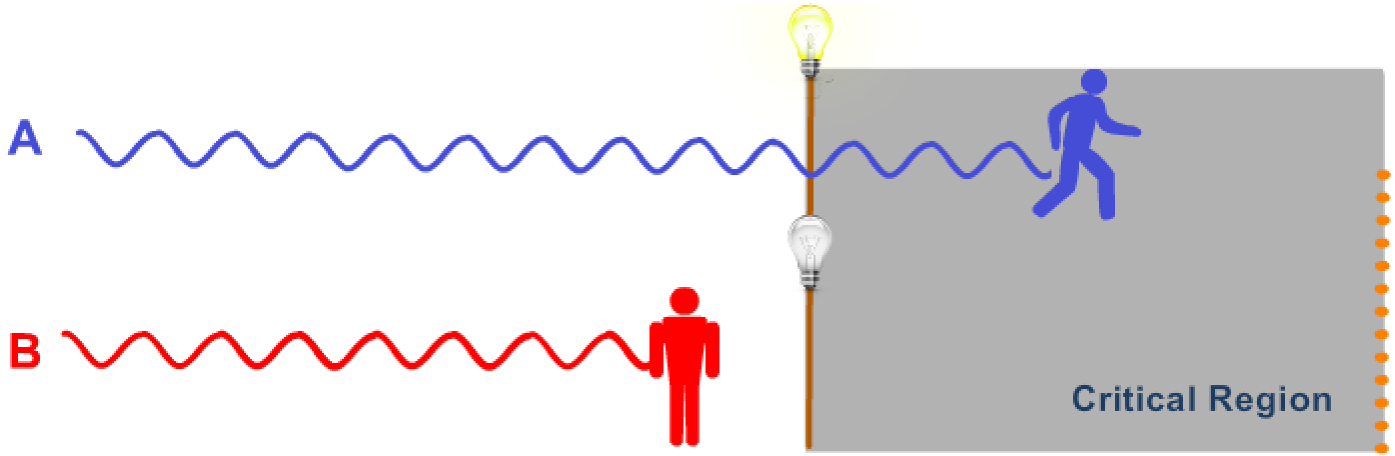
Solution to The CS Problem (2/9)

- In order to solve the critical region problem, a use of external software and/or hardware could be used as shown in the figure:



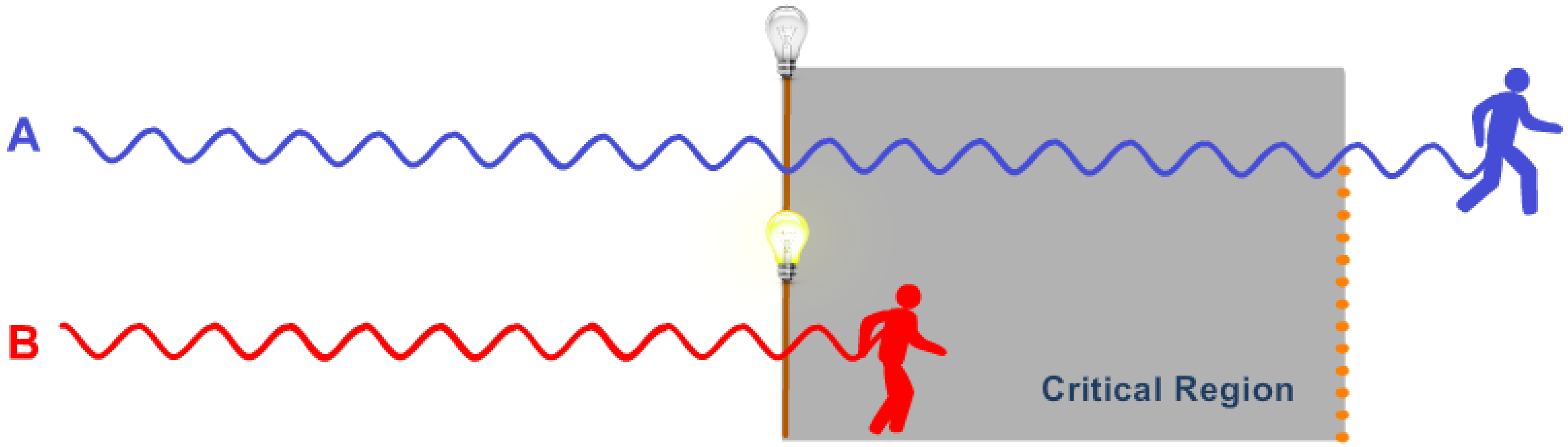
Solution to The CS Problem (3/9)

- In order to solve the critical region problem, a use of external software and/or hardware could be used as shown in the figure:
- When process A enters critical region, a flag is highlighted to indicate that no other process is allowed to enter this critical region.



Solution to The CS Problem (4/9)

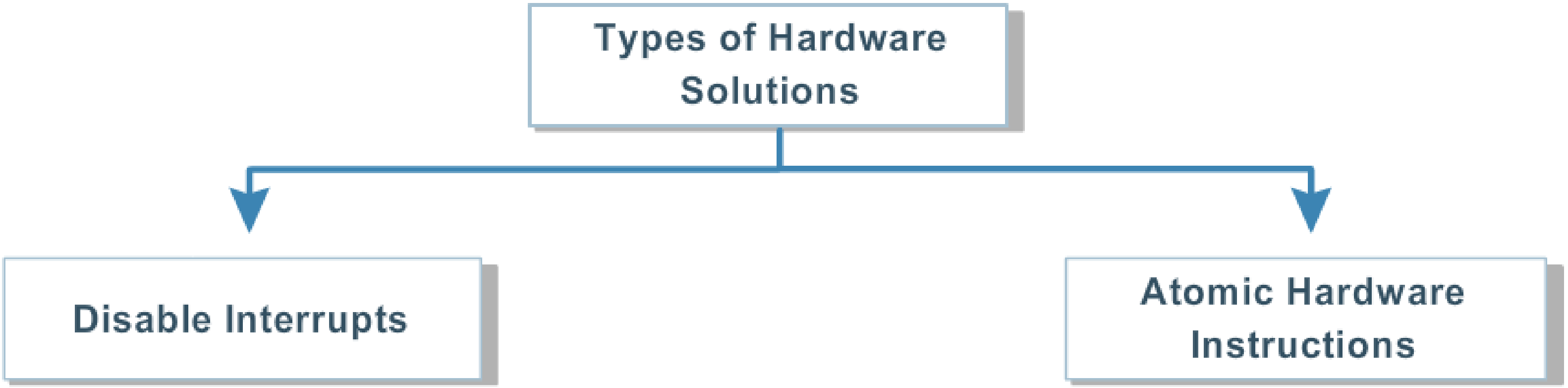
- In order to solve the critical region problem, a use of external software and/or hardware could be used as shown in the figure:
- Then after process A is finished its task, the flag is off allowing process B to continue in the critical region.



Solution to The CS Problem (5/9)

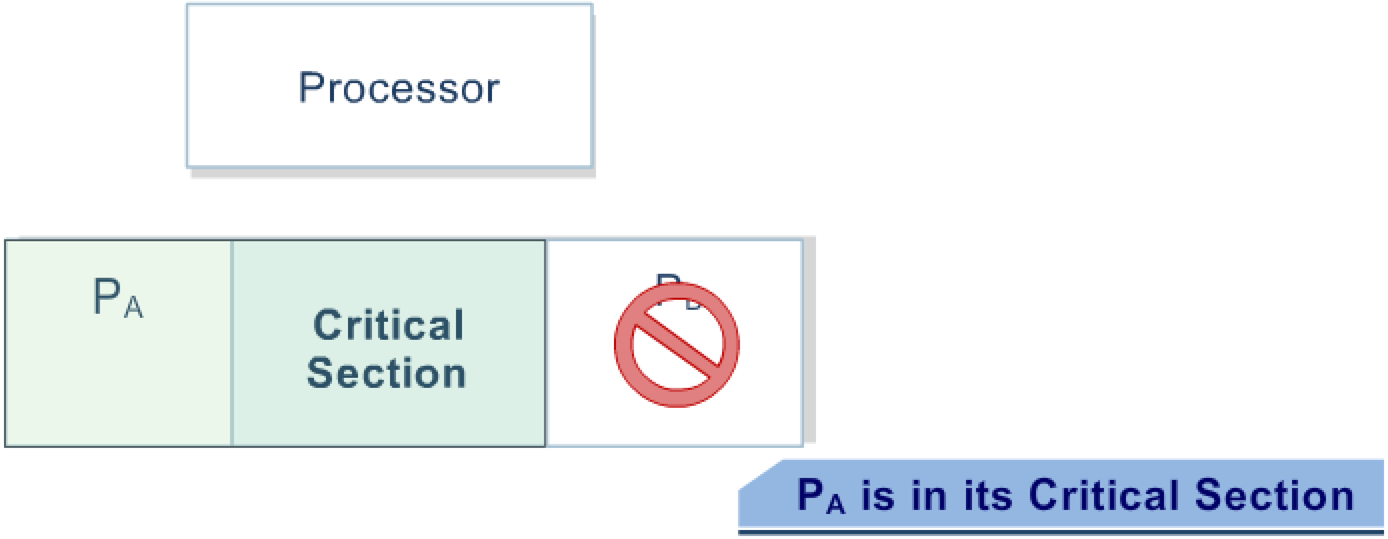
Types of Hardware solutions

- Many systems provide hardware support for critical section code.
- The same concept is used to solve the critical section problem but with the help of hardware.



Solution to The CS Problem (6/9)

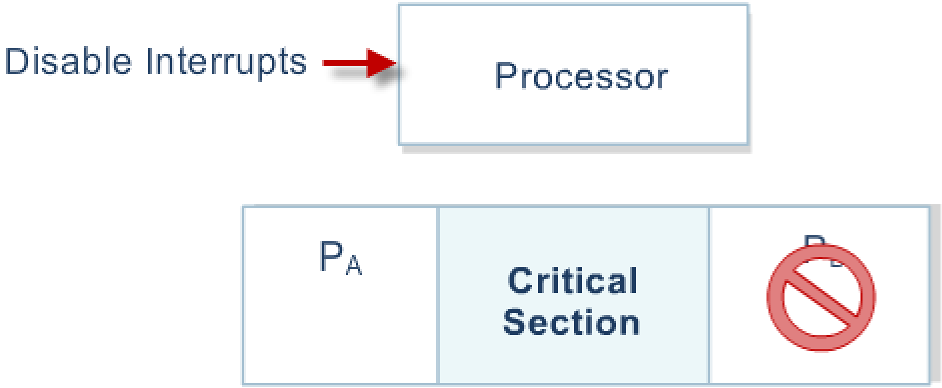
Disable interrupt in uni-processor (1/7)



P_A is in its Critical Section

Solution to The CS Problem (6/9)

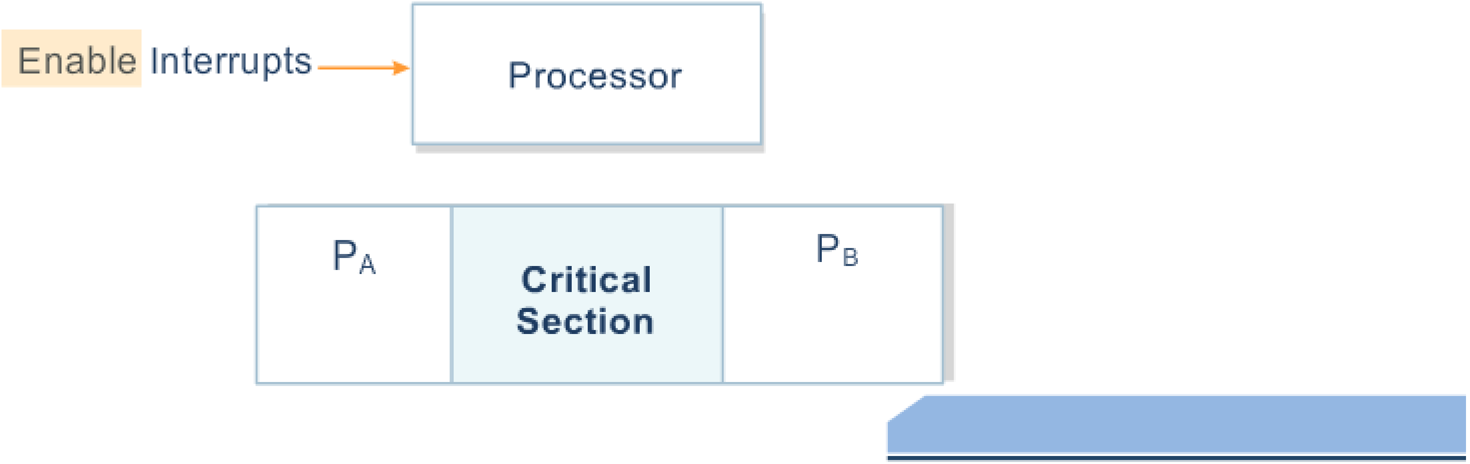
Disable interrupt in uni-processor (2/7)



P_A is in its Critical Section

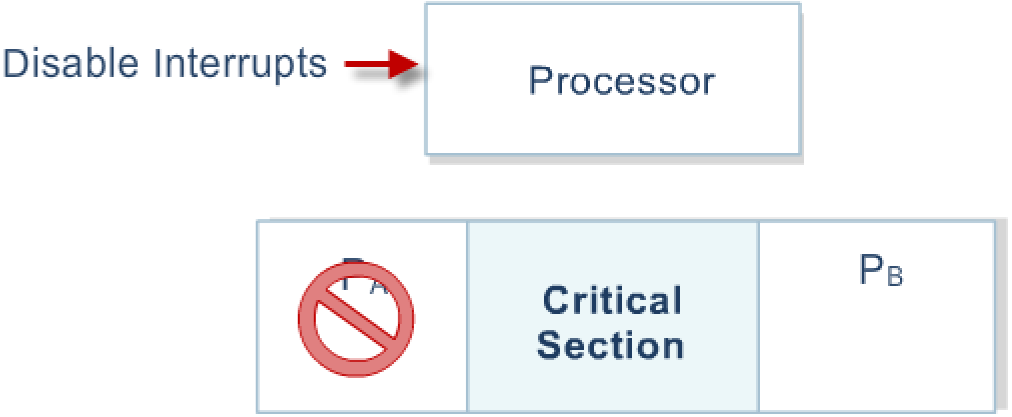
Solution to The CS Problem (6/9)

Disable interrupt in uni-processor (3/7)



Solution to The CS Problem (6/9)

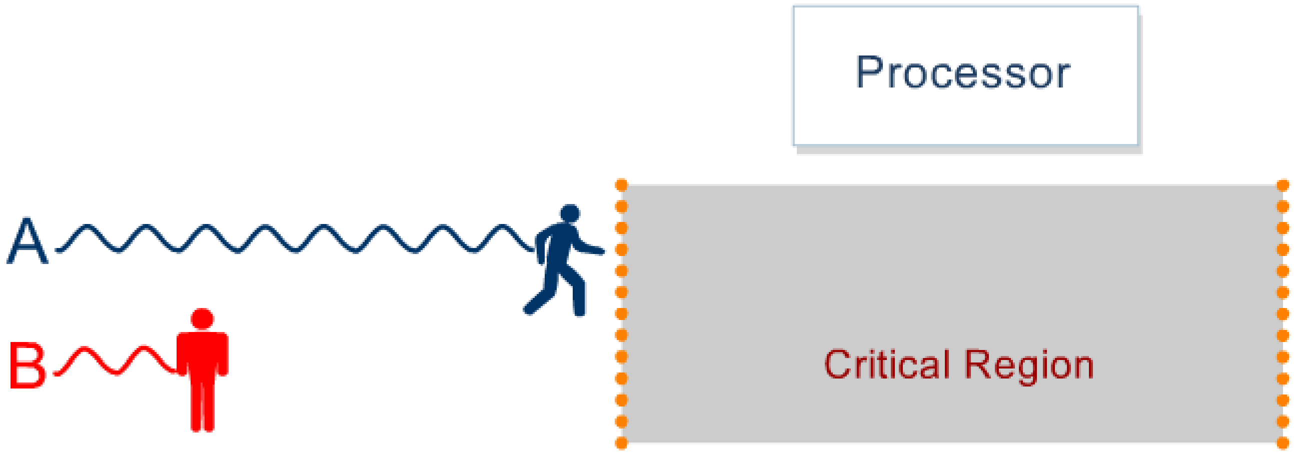
Disable interrupt in uni-processor (4/7)



P_B is in its Critical Section

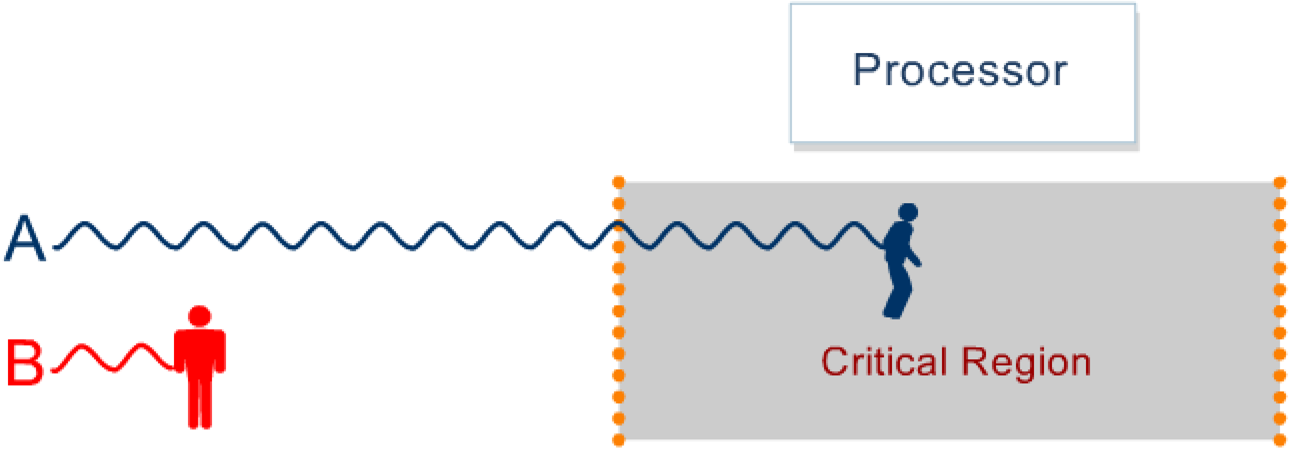
Solution to The CS Problem (6/9)

Disable interrupt in uni-processor (5/7)



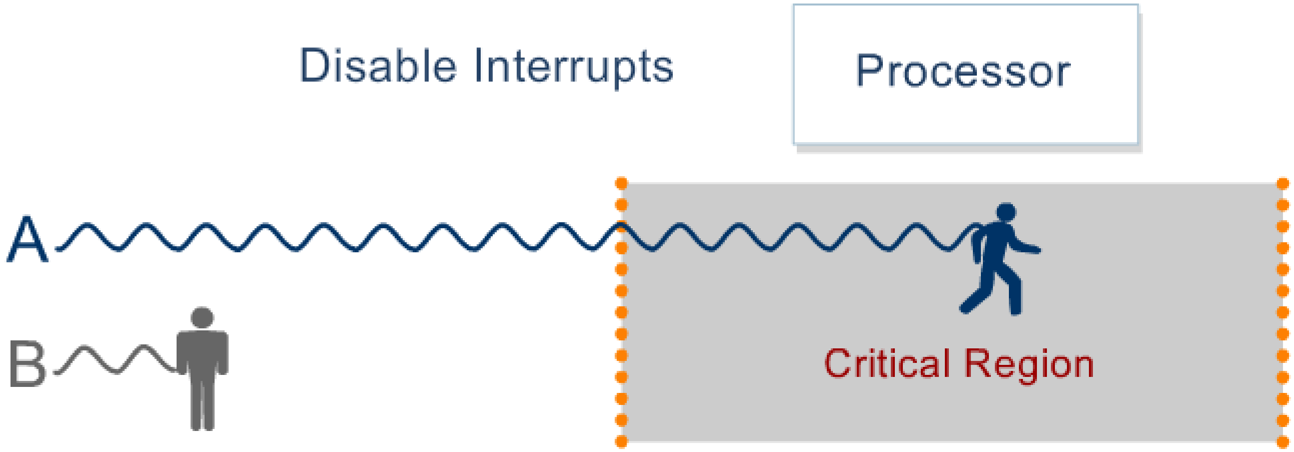
Solution to The CS Problem (6/9)

Disable interrupt in uni-processor (6/7)



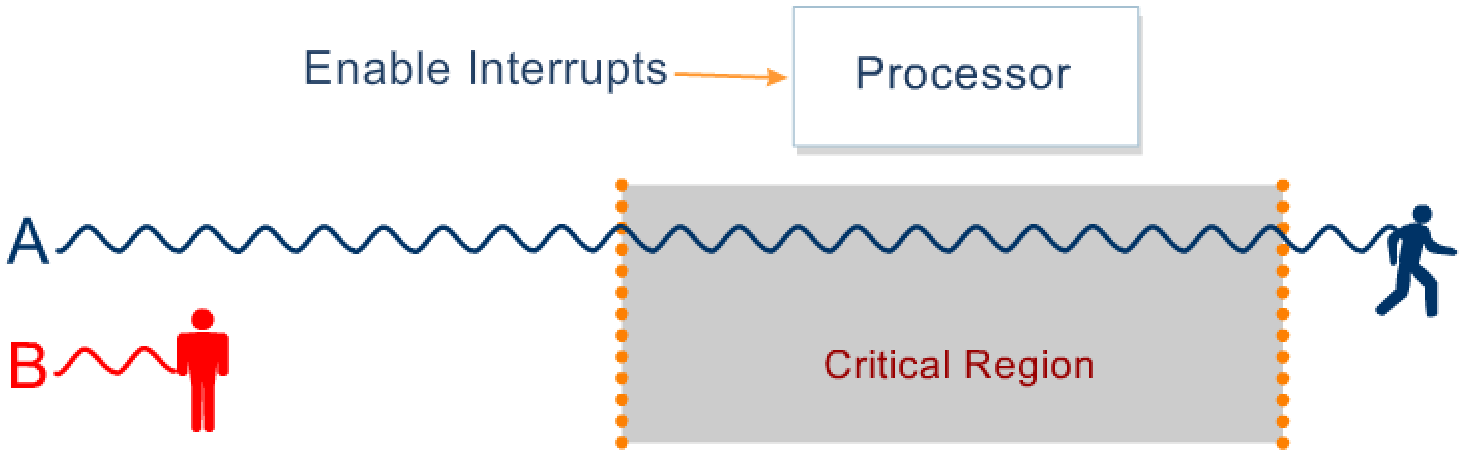
Solution to The CS Problem (6/9)

Disable interrupt in uni-processor (6/7)



Solution to The CS Problem (6/9)

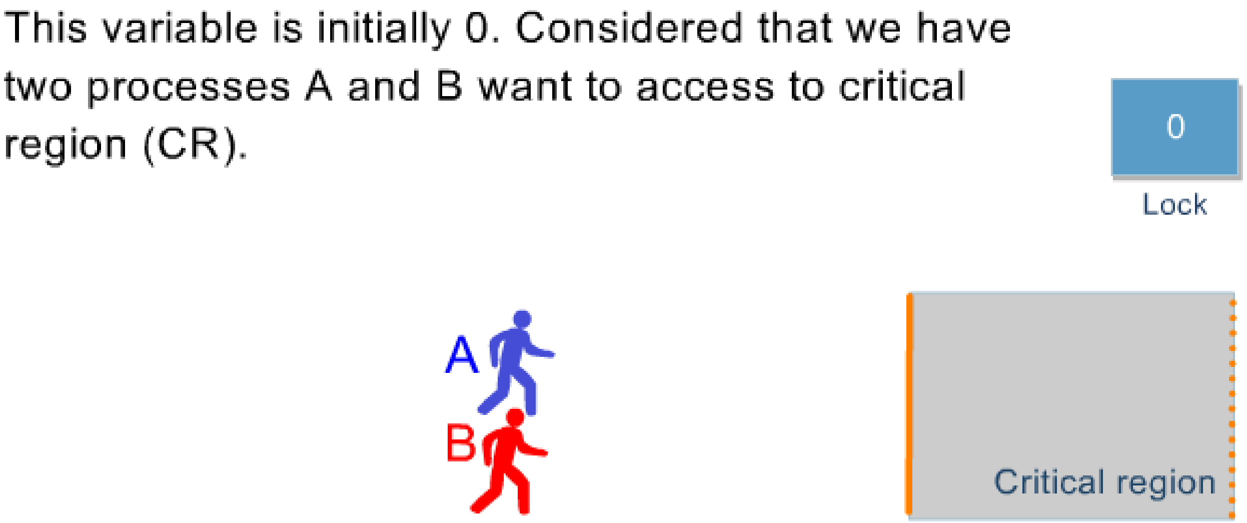
Disable interrupt in uni-processor (7/7)



Solution to The CS Problem (7/9)

Software Solutions (1/7)

- One of the simplest software solutions is based on having a simple common, shared (lock) variable.

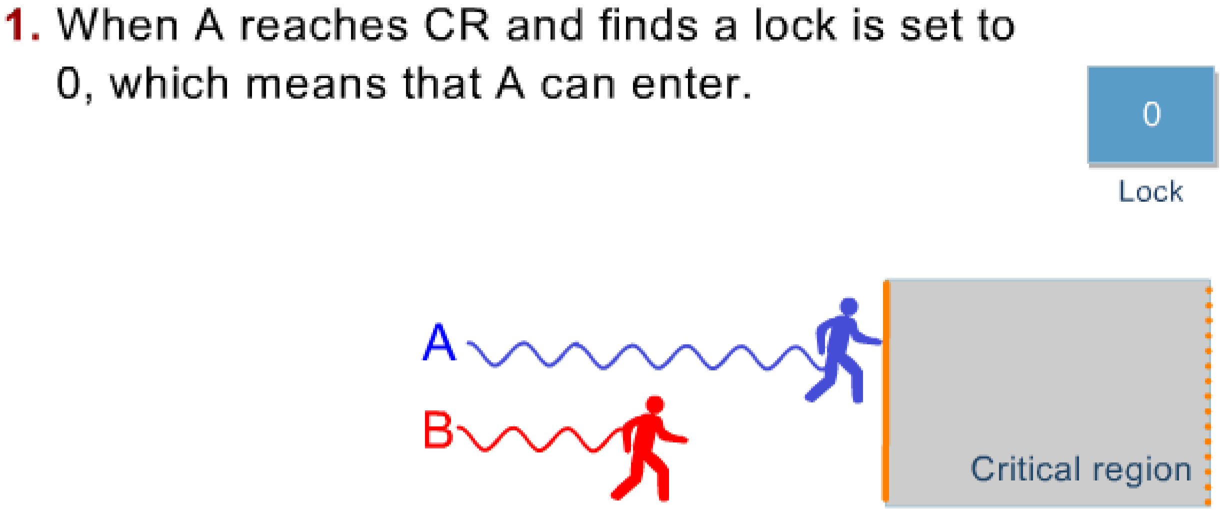


This variable is initially 0. Considered that we have two processes A and B want to access to critical region (CR).

Solution to The CS Problem (7/9)

Software Solutions (2/7)

- One of the simplest software solutions is based on having a simple common, shared (lock) variable.



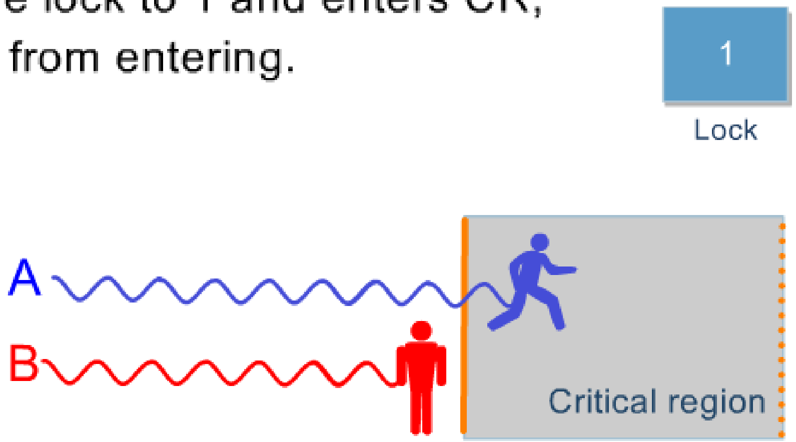
1. When A reaches CR and finds a lock is set to 0, which means that A can enter.

Solution to The CS Problem (7/9)

Software Solutions (3/7)

- One of the simplest software solutions is based on having a simple common, shared (lock) variable.

2. Process A sets the lock to 1 and enters CR, which prevents B from entering.

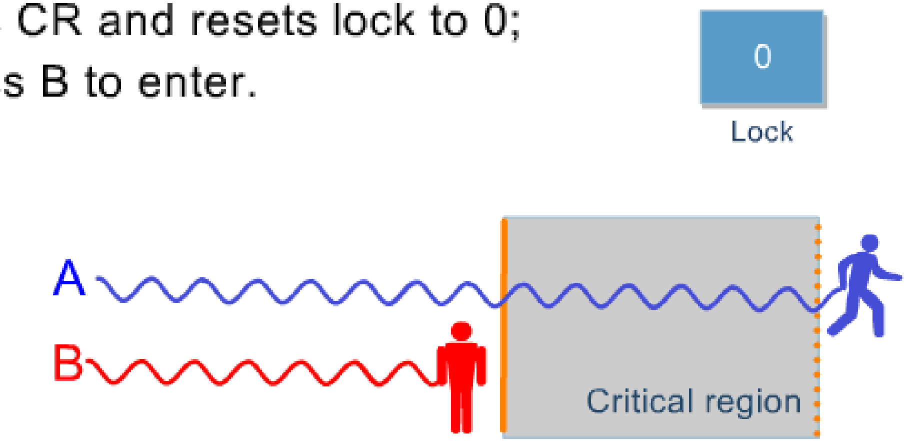


Solution to The CS Problem (7/9)

Software Solutions (4/7)

- One of the simplest software solutions is based on having a simple common, shared (lock) variable.

3. Process A exits CR and resets lock to 0; allowing process B to enter.

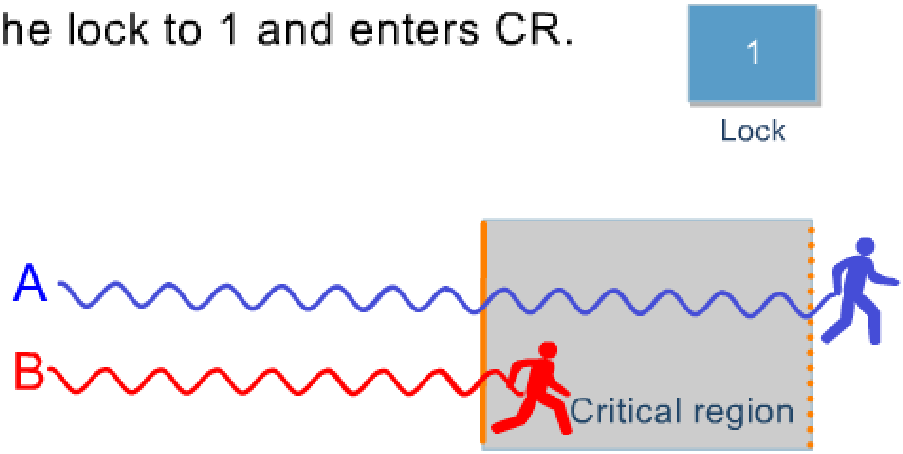


Solution to The CS Problem (7/9)

Software Solutions (5/7)

- One of the simplest software solutions is based on having a simple common, shared (lock) variable.

4. Process B sets the lock to 1 and enters CR.



Solution to The CS Problem (7/9)

Software Solutions (6/7)

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

Solution to The CS Problem (7/9)

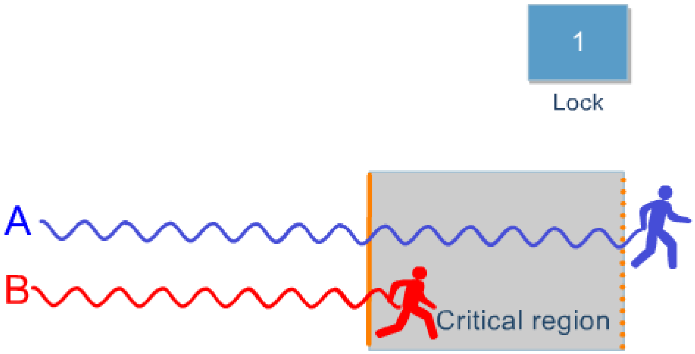
Software Solutions (7/7)

Disadvantage of Software Solution

This solution has the following problem:

The “lock” is a shared variable. Entering the critical region means testing and then setting the lock variable to 1 if applicable.

If A reaches CR and finds a lock at 0, which means that A can enter, but before A can set the lock to 1, B reaches CR and finds the lock is 0, too. Thus the lock variable becomes a CR.



Solution to The CS Problem (8/9)

Peterson’s Solution (software solutions) (1/6)

- The two processes share two variables:

```
int turn;
Boolean flag[2]
```

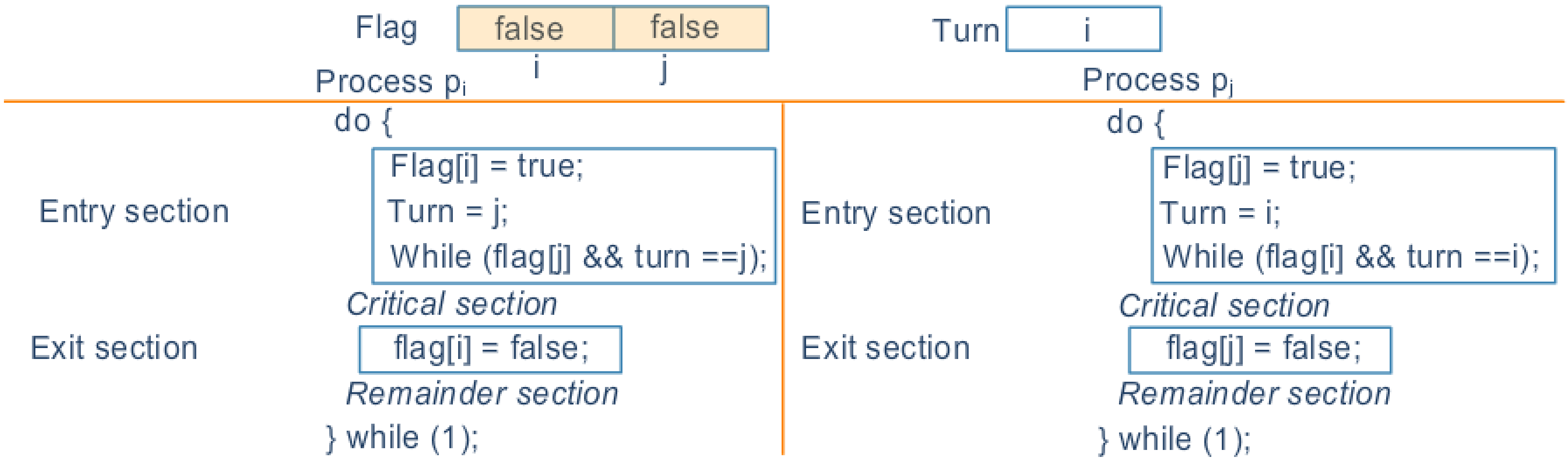
- The variable turn indicates whose turn it is to enter the critical section.

- The 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 $\mathbf{P}_i$ is ready!

Solution to The CS Problem (8/9)

Peterson’s Solution (software solutions) (2/6)



Flag	false	false
	i	j

Turn: i

Process p_i

```
do {
    Flag[i] = true;
    Turn = j;
    While (flag[j] && turn ==j);
        Critical section
    flag[i] = false;
        Remainder section
} while (1);
```

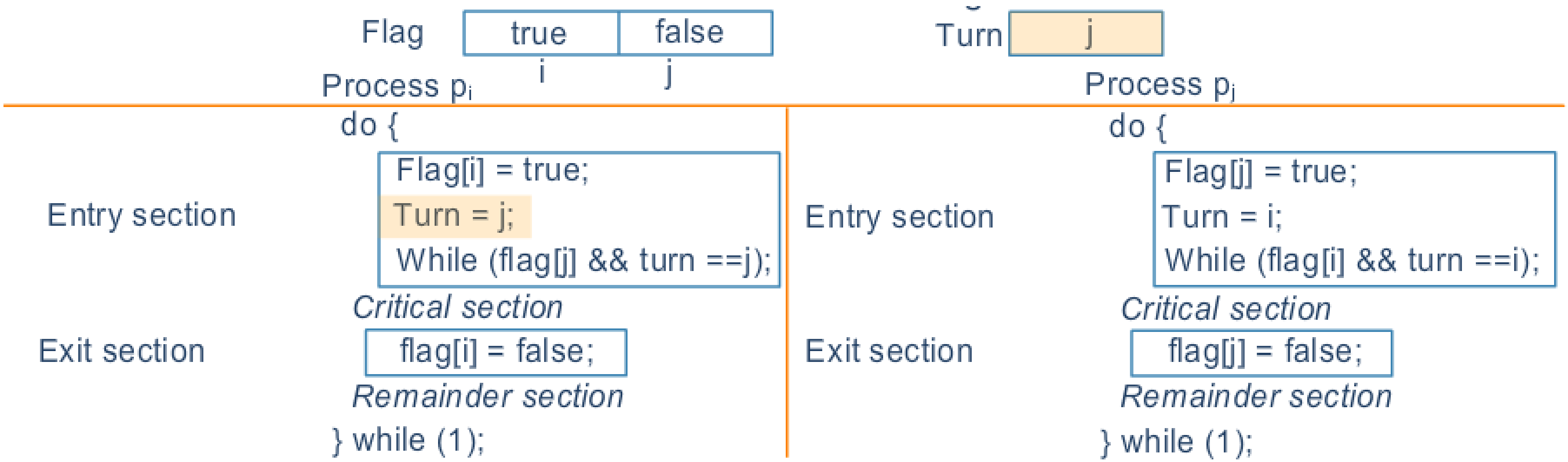
Process p_j

```
do {
    Flag[j] = true;
    Turn = i;
    While (flag[i] && turn ==i);
        Critical section
    flag[j] = false;
        Remainder section
} while (1);
```

- Initially, both processes are not in their critical region, and the array flag has all its elements set to false.

Solution to The CS Problem (8/9)

Peterson's Solution (software solutions) (3/6)



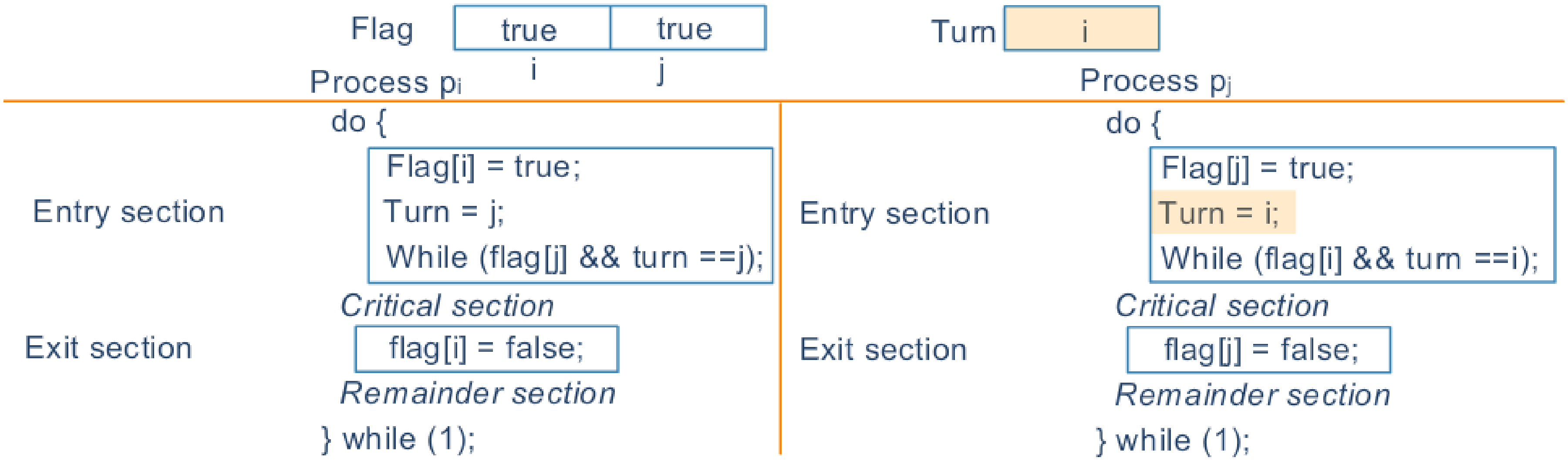
```
Process pi
do {
    Flag[i] = true;
    Turn = j;
    While (flag[j] && turn ==j);
  Critical section
    flag[i] = false;
  Remainder section
} while (1);
```

```
Process pj
do {
    Flag[j] = true;
    Turn = i;
    While (flag[i] && turn ==i);
  Critical section
    flag[j] = false;
  Remainder section
} while (1);
```

1. **P$_i$** arrives first, sets its flag, pushes the turn to the other flag and may enter.

Solution to The CS Problem (8/9)

Peterson’s Solution (software solutions) (4/6)



Flag	true	true
	i	j

Turn	i

Process p_i

```
                do {
Entry section       Flag[i] = true;
                    Turn = j;
                    While (flag[j] && turn ==j);
                 Critical section
Exit section        flag[i] = false;
                 Remainder section
                } while (1);
```

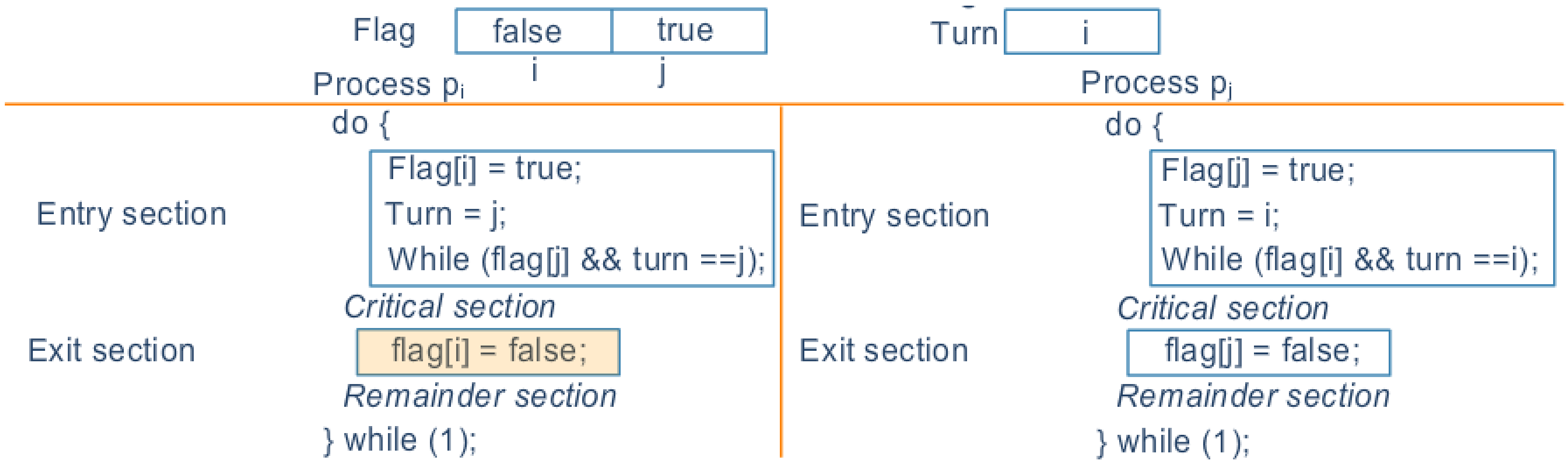
Process p_j

```
                do {
Entry section       Flag[j] = true;
                    Turn = i;
                    While (flag[i] && turn ==i);
                 Critical section
Exit section        flag[j] = false;
                 Remainder section
                } while (1);
```

2. Then, $\mathbf{P}_j$ also sets its flag & pushes the turn, but must wait until $\mathbf{P}_i$’s flag is reset Process $\mathbf{P}_j$ will only be allowed to continue when flag[0] is set to false which can only come about from process $\mathbf{P}_i$.

Solution to The CS Problem (8/9)

Peterson’s Solution (software solutions) (5/6)



Flag | false | true
i j

Turn | i

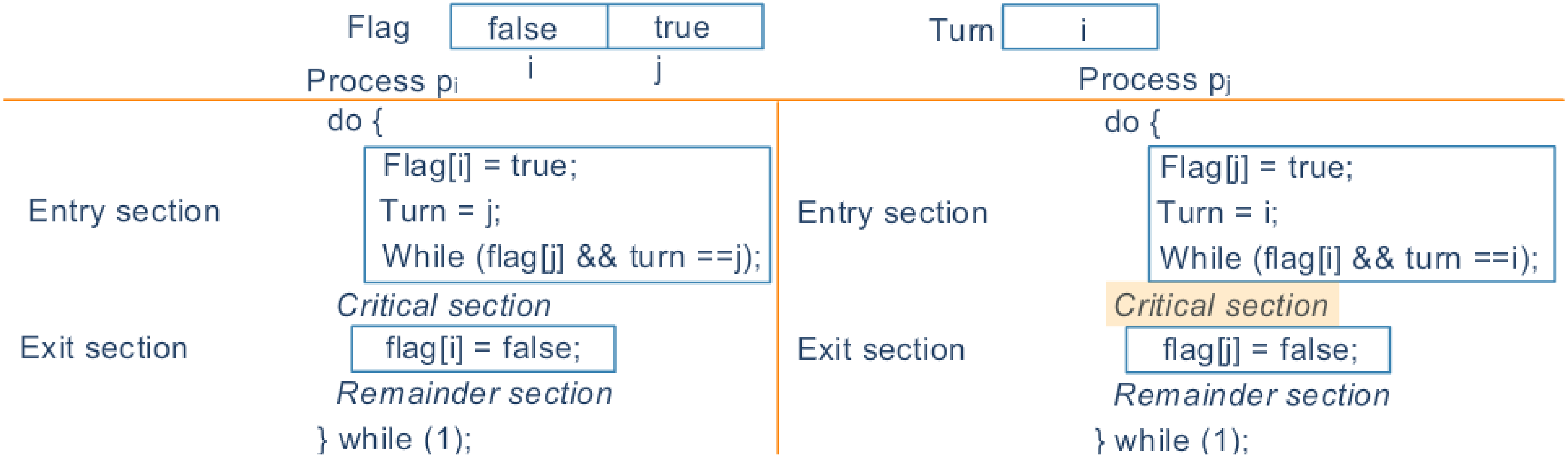
```
Process pi
do {
Entry section      Flag[i] = true;
                   Turn = j;
                   While (flag[j] && turn ==j);
                 Critical section
Exit section       flag[i] = false;
                 Remainder section
} while (1);
```

```
Process pj
do {
Entry section      Flag[j] = true;
                   Turn = i;
                   While (flag[i] && turn ==i);
                 Critical section
Exit section       flag[j] = false;
                 Remainder section
} while (1);
```

3. $\mathbf{P}_i$ exits the CR and resets its `flag [i]`; $\mathbf{P}_j$ may now enter.

Solution to The CS Problem (8/9)

Peterson’s Solution (software solutions) (6/6)



```
                Flag  | false | true  |                 Turn | i |
Process pi               i       j                Process pj
------------------------------------------------------------------------------
                do {                        |                     do {
Entry section     Flag[i] = true;           | Entry section         Flag[j] = true;
                  Turn = j;                 |                       Turn = i;
                  While (flag[j] && turn ==j); |                    While (flag[i] && turn ==i);
                Critical section            |                     Critical section
Exit section      flag[i] = false;          | Exit section          flag[j] = false;
                Remainder section           |                     Remainder section
                } while (1);                |                     } while (1);
```

3. **P**$_i$ exits the CR and resets its `flag [i]`; **P**$_j$ may now enter.

Solution to The CS Problem (9/9)

Semaphore (1/10)

- Semaphore **`S`** – integer variable
- Can only be accessed via two indivisible (atomic) operations
  - **`wait()`** and **`signal()`**
    - Originally called **`P()`** and **`V()`**
- Definition of the **`wait()` operation**

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

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

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

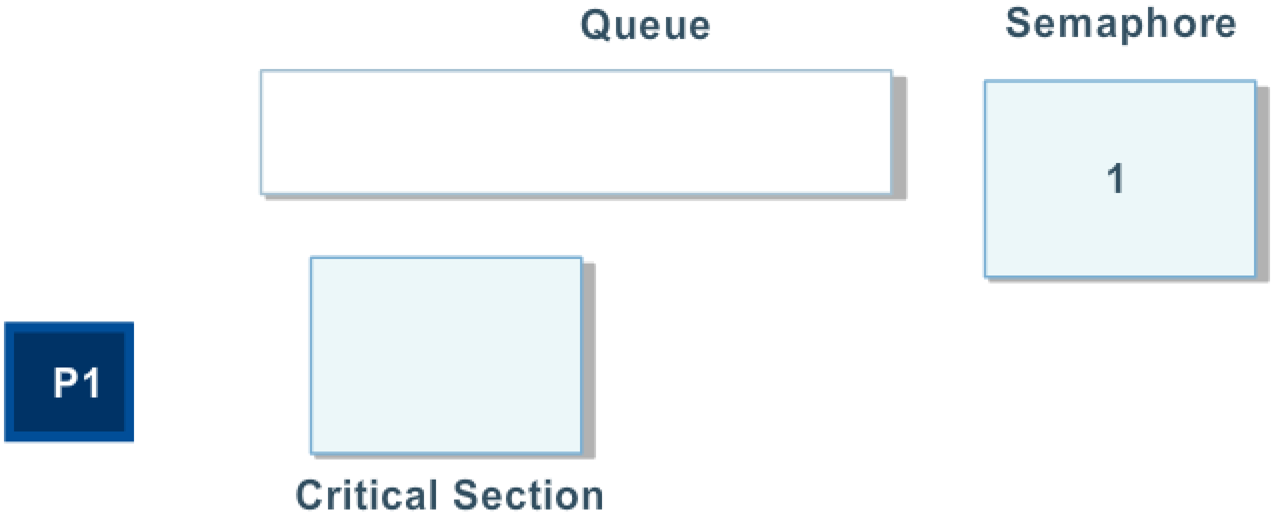
Solution to The CS Problem (9/9)

Semaphore (2/10)

- To overcome the **problem** of **busy waiting in semaphore**, we can modify the defining of the wait and signal operation.
- When a process executes the wait operation and find the value of semaphore is not positive, it must wait. However rather than engaging in a busy waiting, the process can block itself in waiting queue associated with each semaphore.
- In this case the state of the process is waiting in the **semaphore queue**.

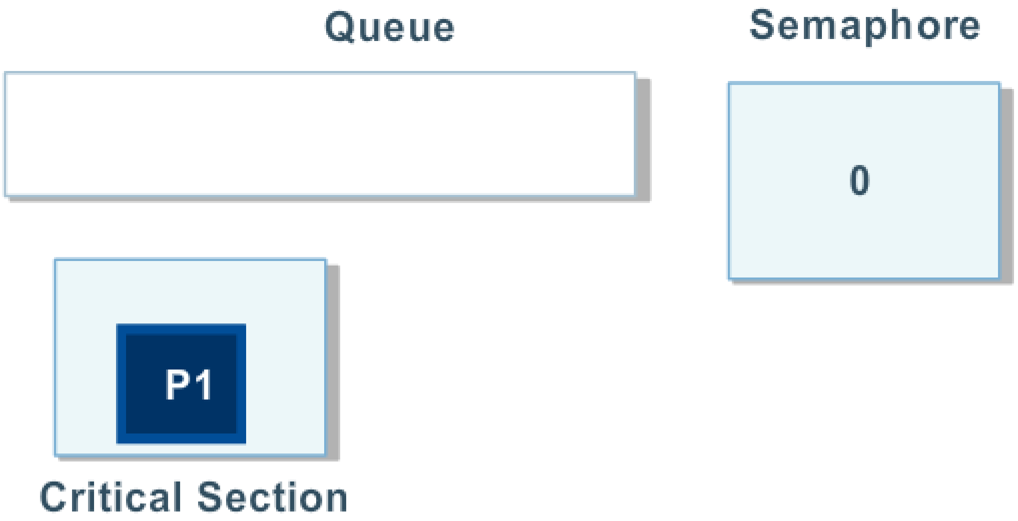
Solution to The CS Problem (9/9)

Semaphore (3/10)



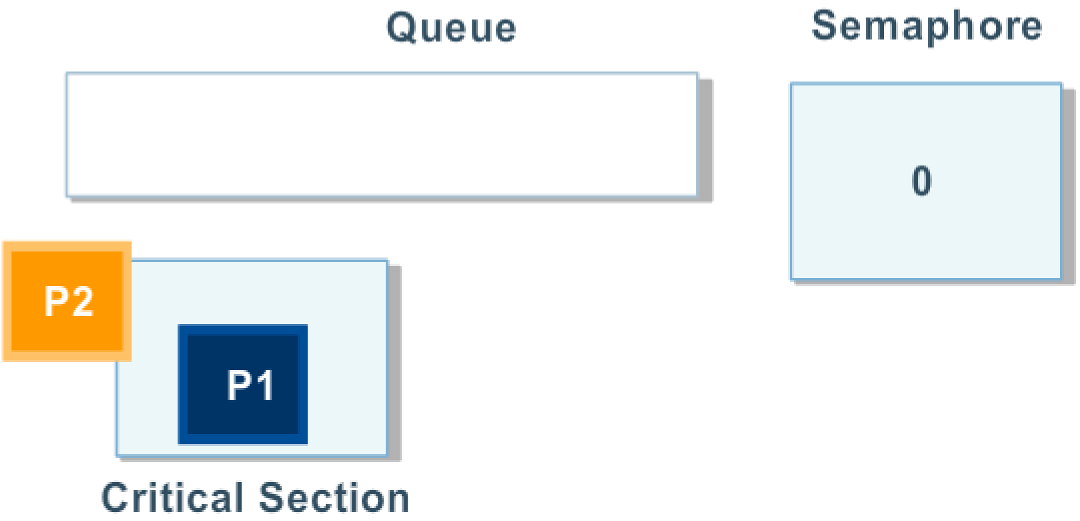
Solution to The CS Problem (9/9)

Semaphore (4/10)



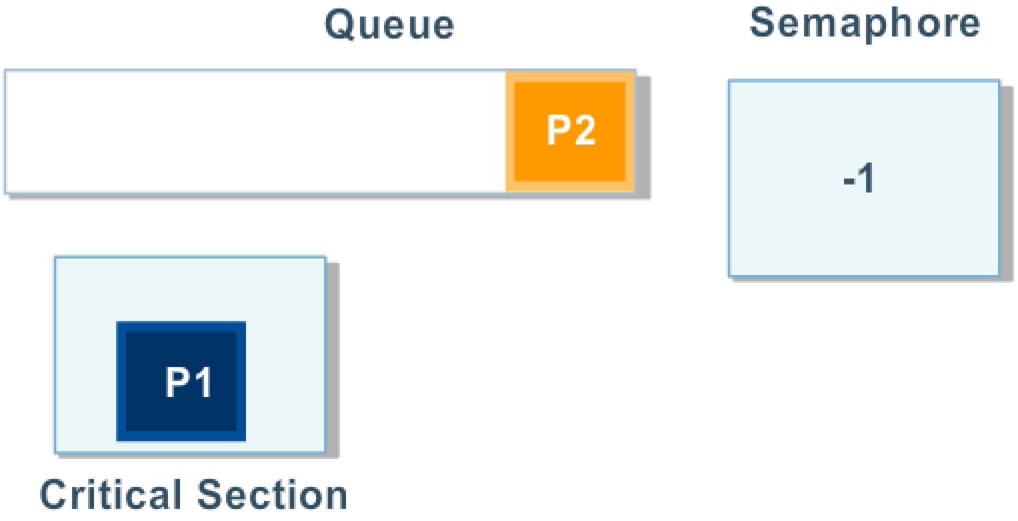
Solution to The CS Problem (9/9)

Semaphore (5/10)



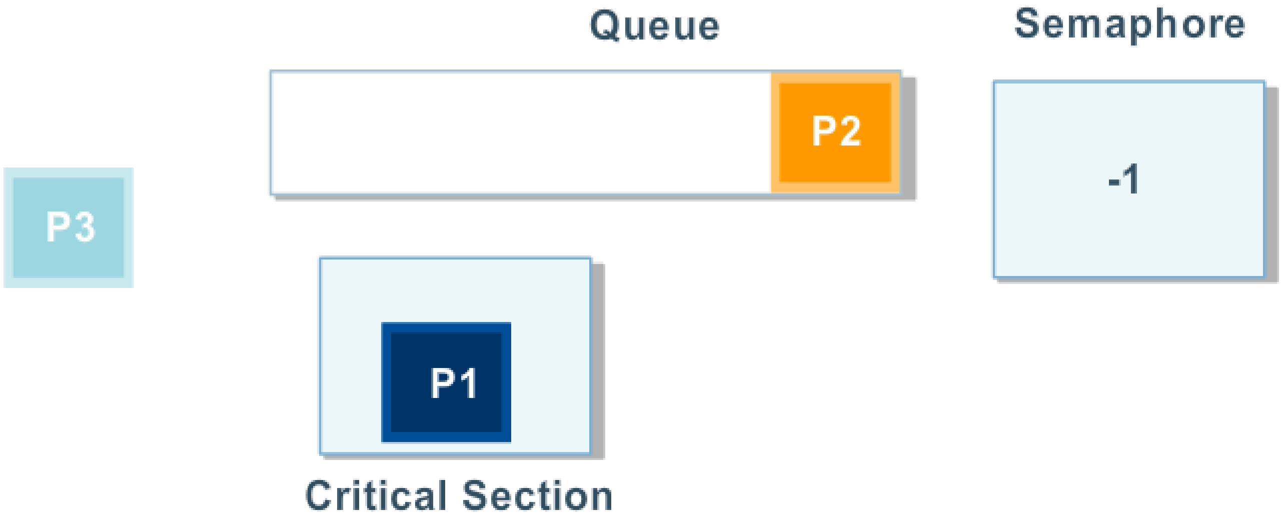
Solution to The CS Problem (9/9)

Semaphore (6/10)



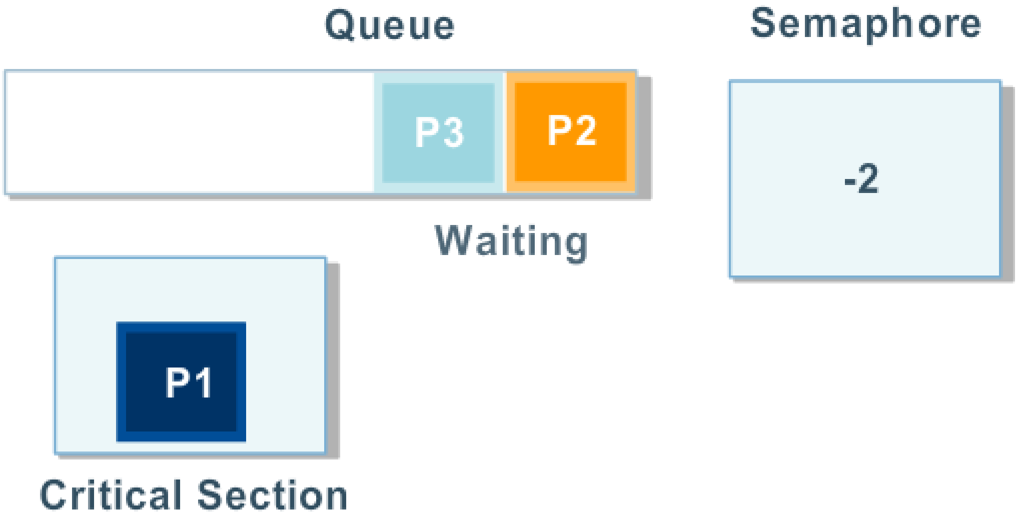
Solution to The CS Problem (9/9)

Semaphore (7/10)



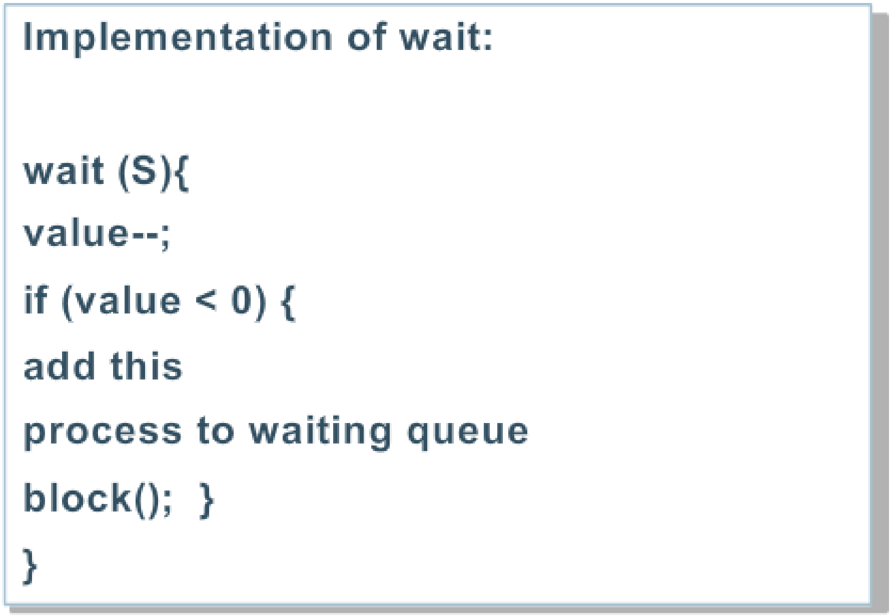
Solution to The CS Problem (9/9)

Semaphore (8/10)



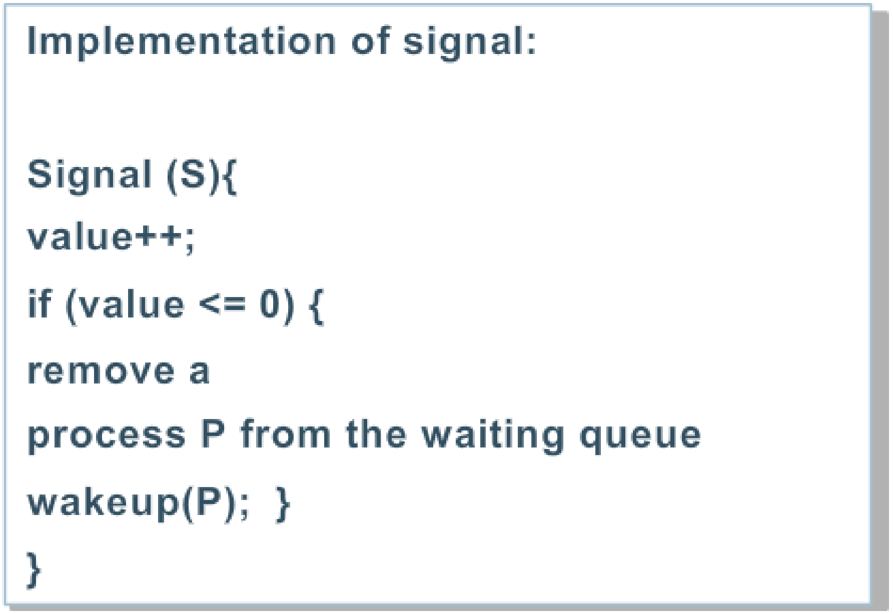
Solution to The CS Problem (9/9)

Semaphore (9/10)



Implementation of wait:

```
wait (S){
value--;
if (value < 0) {
add this
process to waiting queue
block(); }
}
```



Implementation of signal:

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

Solution to The CS Problem (9/9)

Semaphore (10/10)

- **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

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

Thank You

Dr. Ahmed Hagag

ahagag@fci.bu.edu.eg

https://www.youtube.com/channel/UCzYgAyyZTLfnLFjQexOKxbQ

https://www.facebook.com/ahmed.hagag.71/