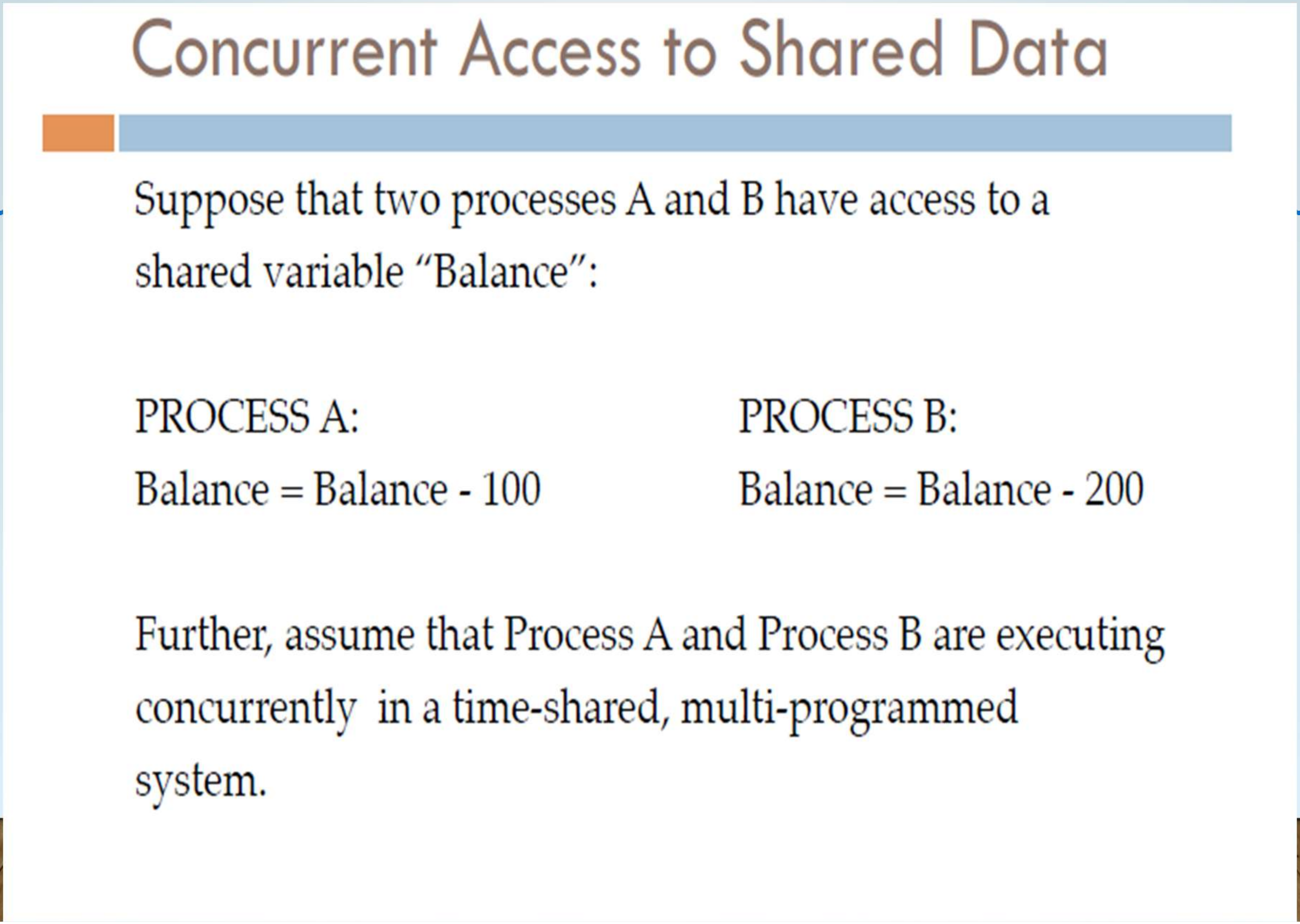
UNIT 3
INTER-PROCESS COMMUNICATION, SYNCHRONIZATION AND DEADLOCKS

CONCURRENCY AND PARALLELISM

- Concurrency
- Parallelism
- Principles of concurrency: interleaving , overlapping
- Problems in uniprocessor and multi processor systems

CONCURRENT ACCESS TO SHARED DATA

- Concurrent access to shared data may result in data inconsistency (e.g., due to race conditions)
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes

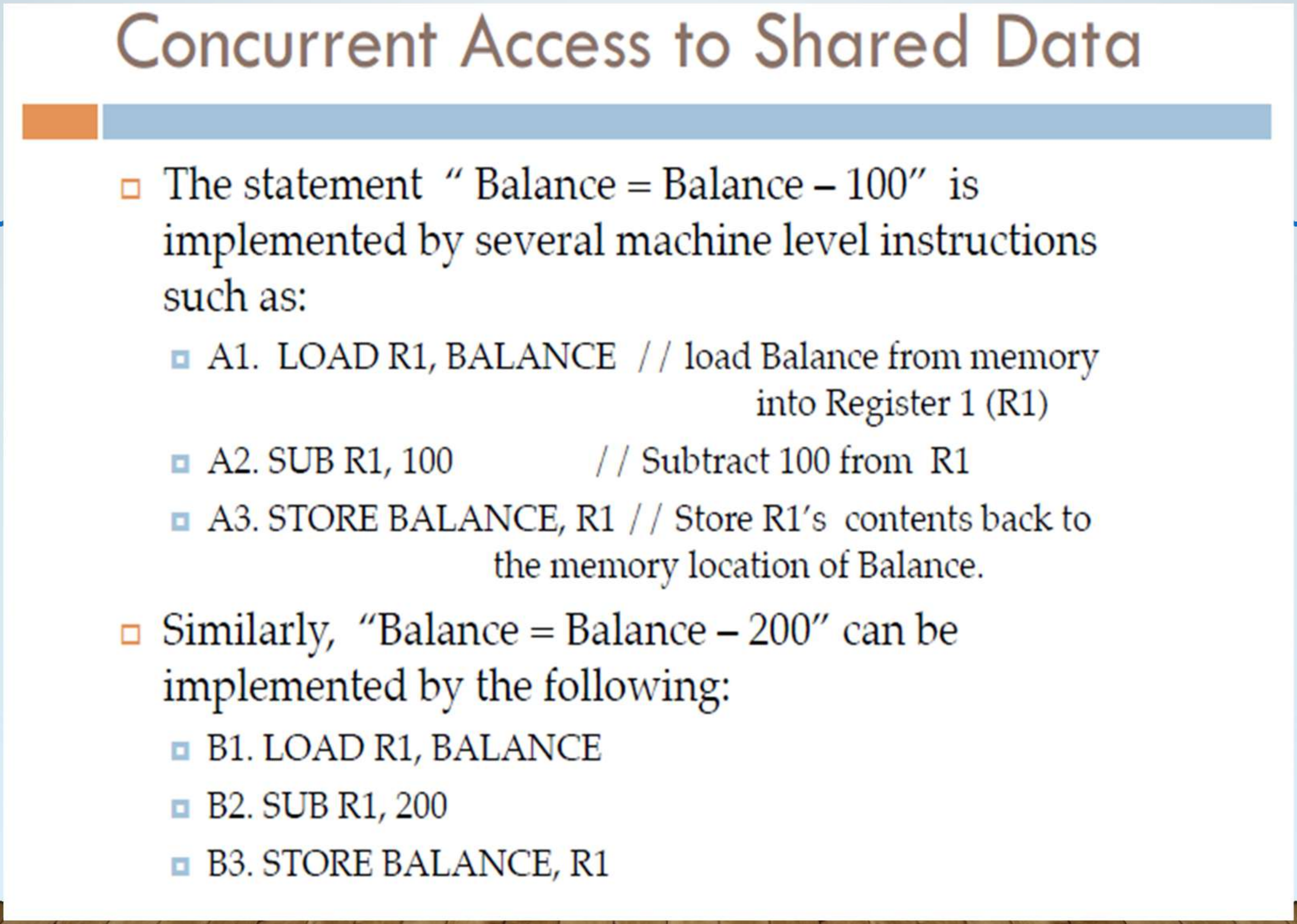


Concurrent Access to Shared Data

Suppose that two processes A and B have access to a shared variable "Balance":

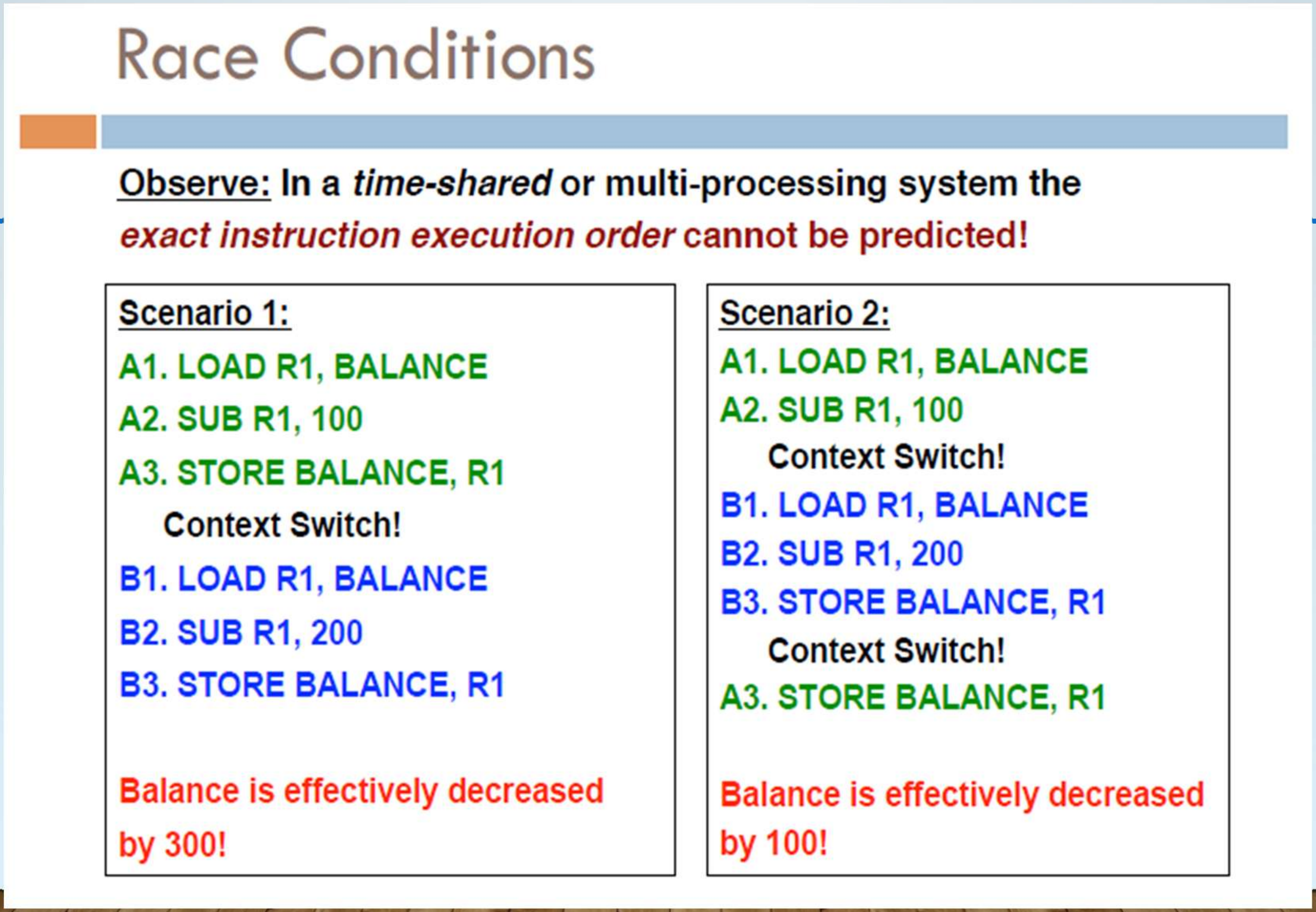
PROCESS A:	PROCESS B:
Balance = Balance - 100	Balance = Balance - 200

Further, assume that Process A and Process B are executing concurrently in a time-shared, multi-programmed system.



Concurrent Access to Shared Data

- The statement " Balance = Balance – 100" is implemented by several machine level instructions such as:
  - A1. LOAD R1, BALANCE // load Balance from memory into Register 1 (R1)
  - A2. SUB R1, 100 // Subtract 100 from R1
  - A3. STORE BALANCE, R1 // Store R1's contents back to the memory location of Balance.
- Similarly, "Balance = Balance – 200" can be implemented by the following:
  - B1. LOAD R1, BALANCE
  - B2. SUB R1, 200
  - B3. STORE BALANCE, R1



Race Conditions

<u>Observe:</u> In a *time-shared* or multi-processing system the *exact instruction execution order* cannot be predicted!

Scenario 1:

A1. LOAD R1, BALANCE

A2. SUB R1, 100

A3. STORE BALANCE, R1

Context Switch!

B1. LOAD R1, BALANCE

B2. SUB R1, 200

B3. STORE BALANCE, R1

Balance is effectively decreased by 300!

Scenario 2:

A1. LOAD R1, BALANCE

A2. SUB R1, 100

Context Switch!

B1. LOAD R1, BALANCE

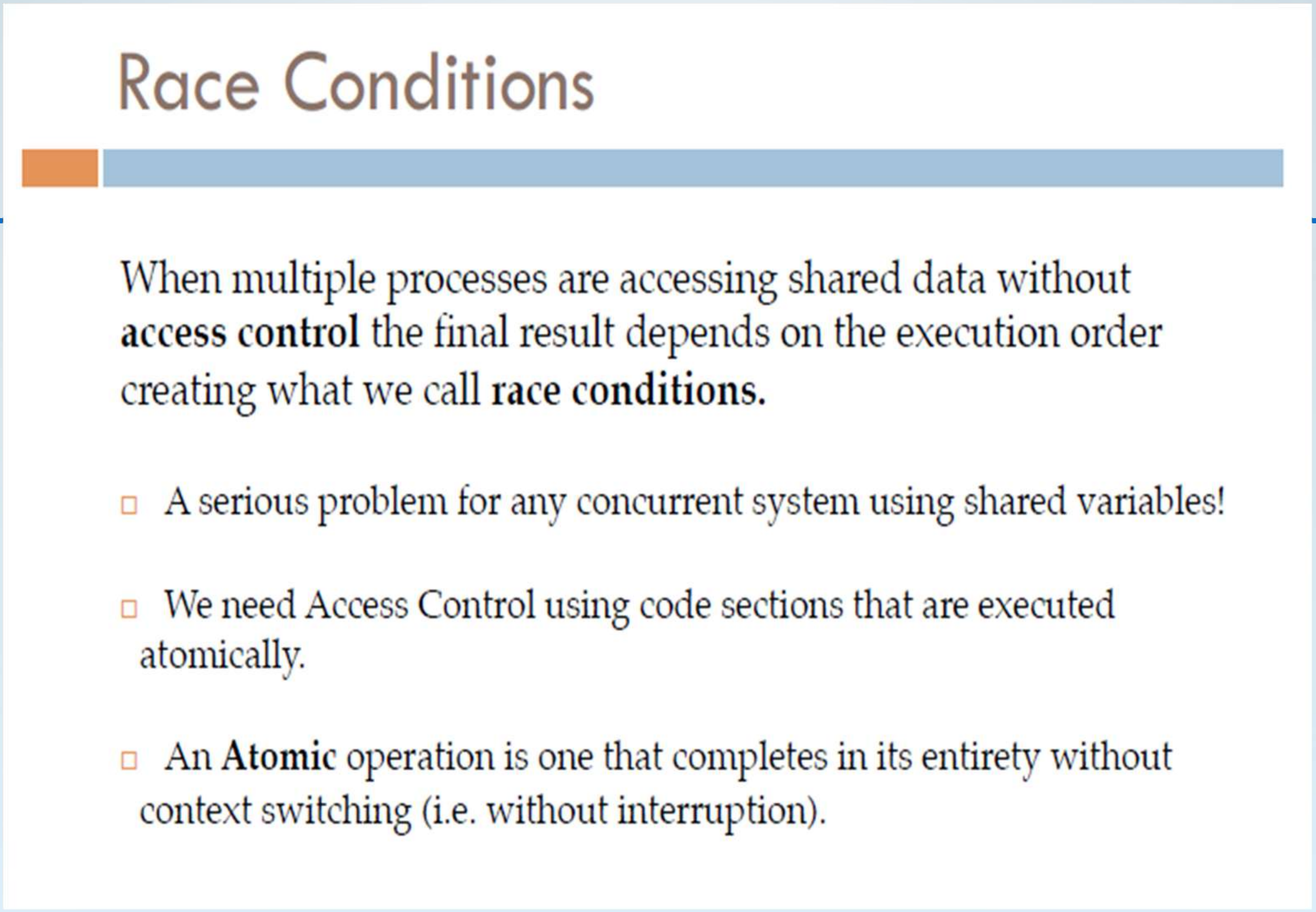
B2. SUB R1, 200

B3. STORE BALANCE, R1

Context Switch!

A3. STORE BALANCE, R1

Balance is effectively decreased by 100!



Race Conditions

When multiple processes are accessing shared data without **access control** the final result depends on the execution order creating what we call **race conditions.**

- A serious problem for any concurrent system using shared variables!
- We need Access Control using code sections that are executed atomically.
- An **Atomic** operation is one that completes in its entirety without context switching (i.e. without interruption).

OPERATING SYSTEM CONCERNS

- Keep track of various process
- PCB
- Resources allocation
- processor time, memory, files, IO devices
- Process should not interfere each other data and resources
- Speed of process and output produced should not affect meaning interaction of processes must be handled carefully
- Process Interaction

PROCESS INTERACTION

- Processes unaware of each other
- Processes indirectly aware of each other
- Processes directly aware of each other

The Critical-Section Problem

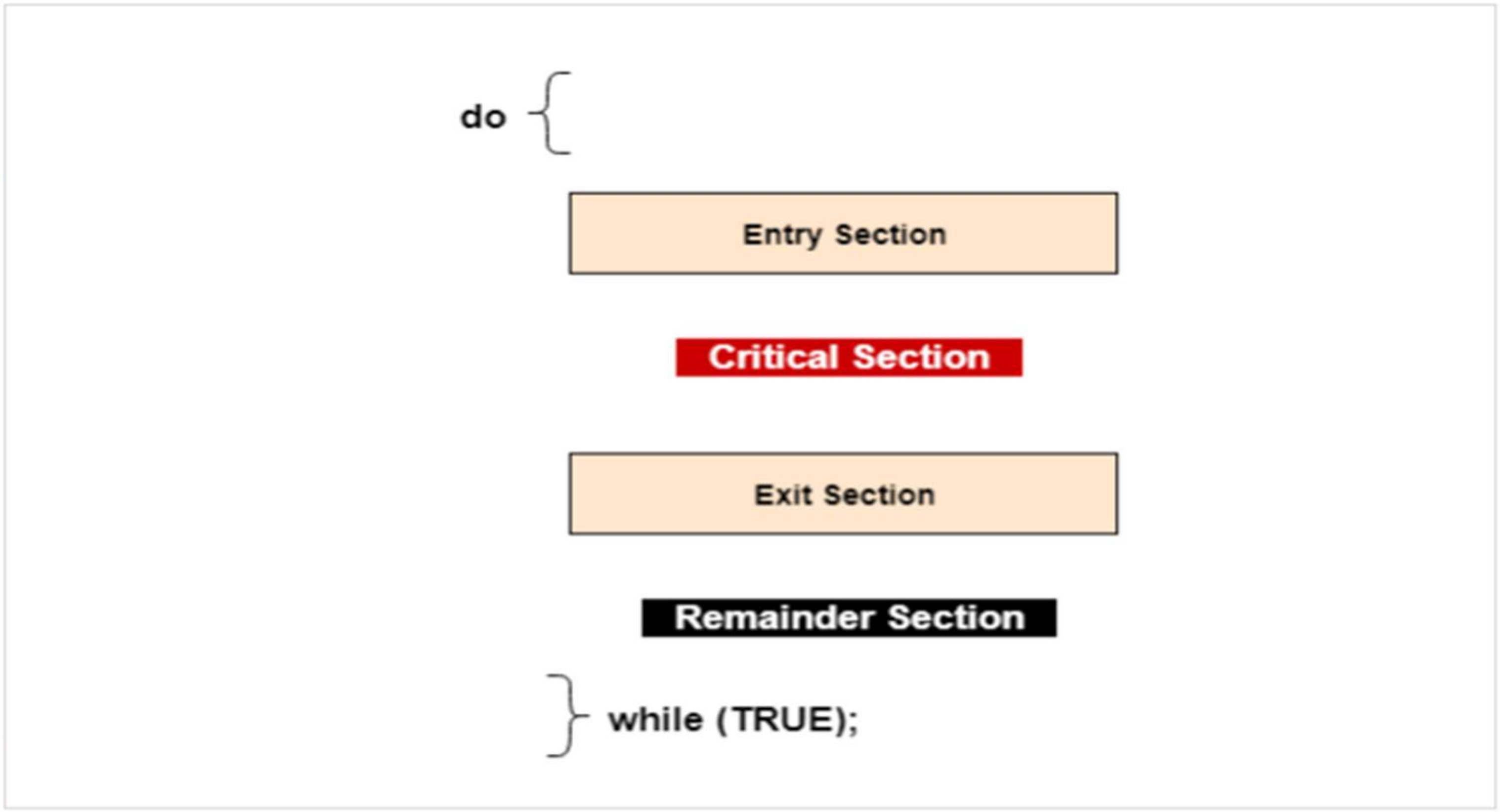
The **critical section problem** is used to design a protocol followed by a group of processes, so that when one process has entered its critical section, no other process is allowed to execute in its critical section.

The **critical section** refers to the segment of code where processes access shared resources, such as common variables and files, and perform write operations on them. Since processes execute concurrently, any process can be interrupted mid-execution. In the case of shared resources, partial execution of processes can lead to data inconsistencies.

When two processes access and manipulate the shared resource concurrently, and the resulting execution outcome depends on the order in which processes access the resource; this is called a *race condition.*

Race conditions lead to inconsistent states of data. Therefore, we need a synchronization protocol that allows processes to cooperate while manipulating shared resources, which essentially is the critical section problem.

A diagram that demonstrates the critical section is as follows –



In the above diagram, the entry section handles the entry into the critical section. It acquires the resources needed for execution by the process. The exit section handles the exit from the critical section. It releases the resources and also informs the other processes that the critical section is free.

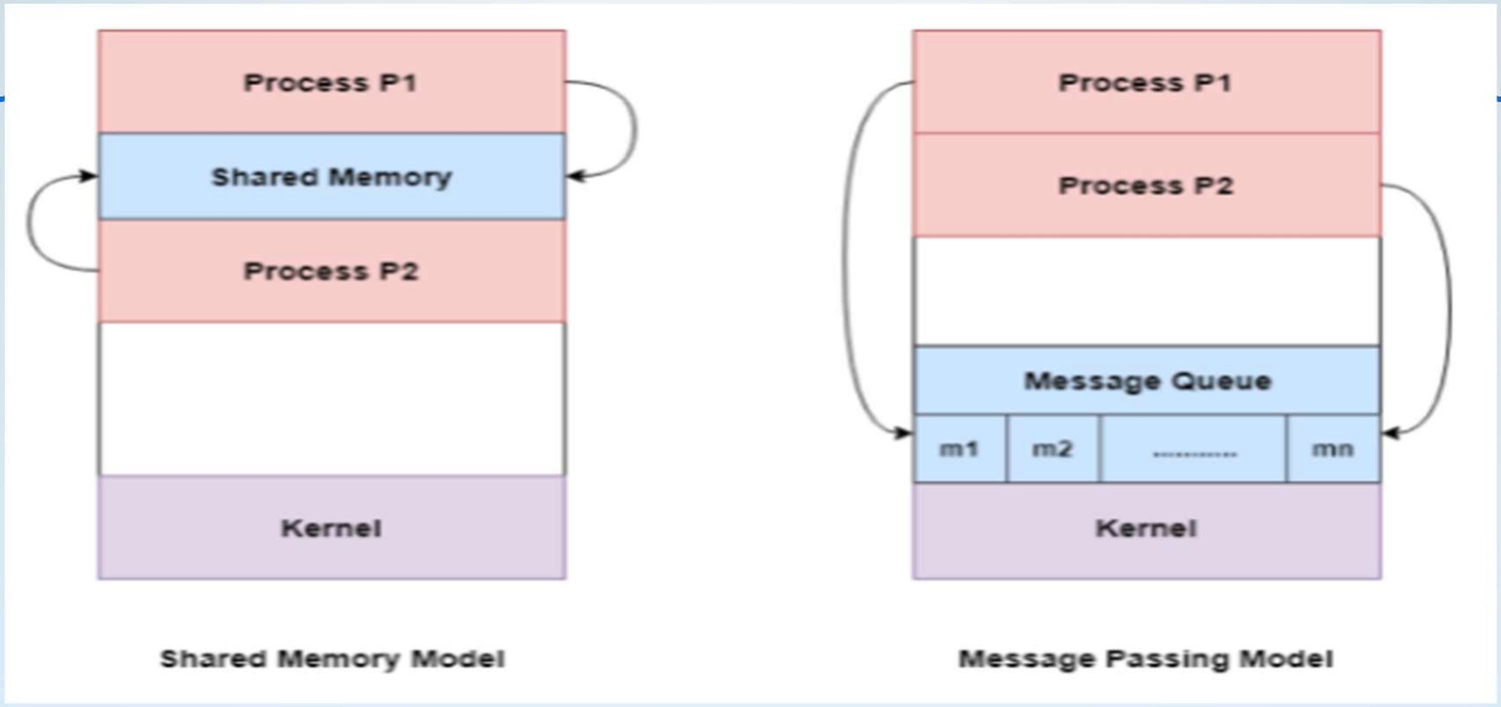
INTER-PROCESS COMMUNICATION (IPC)

- On the basis of synchronization, processes are categorized as one of the following two types:
  - Independent Process : Execution of one process does not affects the execution of other processes.
  - Cooperative Process : Execution of one process affects the execution of other processes.
- Process synchronization problem mainly arises in the case of Cooperative process because resources are shared in Cooperative processes.
- There are certain reasons due to which cooperative environment is created:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience

INTER-PROCESS COMMUNICATION (IPC)

- Mechanism followed by cooperating processes to communicate with each other is called as Inter-process Communication (IPC)
- IPC allows processes to exchange data and information
- There are two fundamental models which allow Inter-process Communication :
  - Shared memory: In this model, sharing region is established. From this sharing region, cooperating processes exchange data or information. They can read and write data from and to this region
  - Message passing: allows multiple processes to read and write data to the message queue without being connected to each other. Messages are stored on the queue until their recipient retrieves them.

INTER-PROCESS COMMUNICATION MODELS



Shared memory is the memory that can be simultaneously accessed by multiple processes. This is done so that the processes can communicate with each other

Both the processes P1 and P2 can access the message queue and store and retrieve data.

SHARED MEMORY MODEL

- Shared memory systems allow processes to share data or information using shared memory region. Shared memory region resides in the address space of the process creating the shared memory segment.
- Other processes which need to share this shared region should connect their address space with the segment of the memory region
- Generally, operating system tries to avoid one process accessing memory of another process. Shared memory concept is used when two or more processes are ready to avoid this restriction. They can then read and write the data to and from shared memory area
- Processes need to ensure that they don’t write at the same location at a time.

MESSAGE PASSING MODEL

- In Message passing model, processes can pass the messages to each other without sharing of their address space. This type of communication is mainly used in distributed environment (where processes reside on different computers connected by the network)
- The best example to illustrate message passing system is the chat which takes place on the World Wide Web, where multiple users chat from different locations. They exchange messages from different nodes/locations
- Two facilities: send and receive are used to exchange the messages
- Messages sent by the processes are of either fixed or variable size. System level implementation is straightforward if the messages sent are of the fixed length and it is complicated for variable length messages

SHARED MEMORY

- Common address space for processes are in same machine
- The shared memory code to be written explicitly by the application programmer
- Provides maximum speed of communication
- Processes should not be writing to the same location simultaneously

MESSAGE PASSING

It is used for communication

It is used in distributed environment

It facilitates mechanism for communication and synchronization of actions , so no separate code required

It implements through system calls so time consuming process

Message passing is useful for sharing small amounts of data so that conflicts need not occur

In message passing the communication is slower when compared to shared memory technique

PRODUCER CONSUMER PROBLEM

- Compiler Assembler
- Server Client
- Web Server (allows HTML files to run and images)
- These are consumed by the client web browser requesting the resources

PRODUCER-CONSUMER PROBLEM

- Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process.
- E.g A compiler may produce assembly code and that would be consumed by an assembler. An assembler produces object modules which are consumed by loader.

Shared memory can be used to solve the P-C problem.

For that purpose a buffer of items can be used which can be placed in the shared memory.

P-C must be synchronized.

Two types of buffers can be used:

- *unbounded-buffer* places no practical limit on the size of the buffer so consumer has to wait if buffer is empty.
- *bounded-buffer* assumes that there is a fixed buffer size so consumer has to wait if buffer is empty and producer has to wait if buffer is full.

CONSUMER

- void consumer (void)
- { int itemc;
- while(true)
- {
- while(count= =0);
- itemc = buffer [out];
- out = (out+1) % n;
- count = count – 1
- } }

PRODUCER

- int count = 0;
- void producer (void)
- { int itemp;
- while(true)
- {
- while(count= = n);
- buffer [in]=itemp;
- in = (in +1) % n;
- count = count + 1
- } }

Solving Critical Section Problem

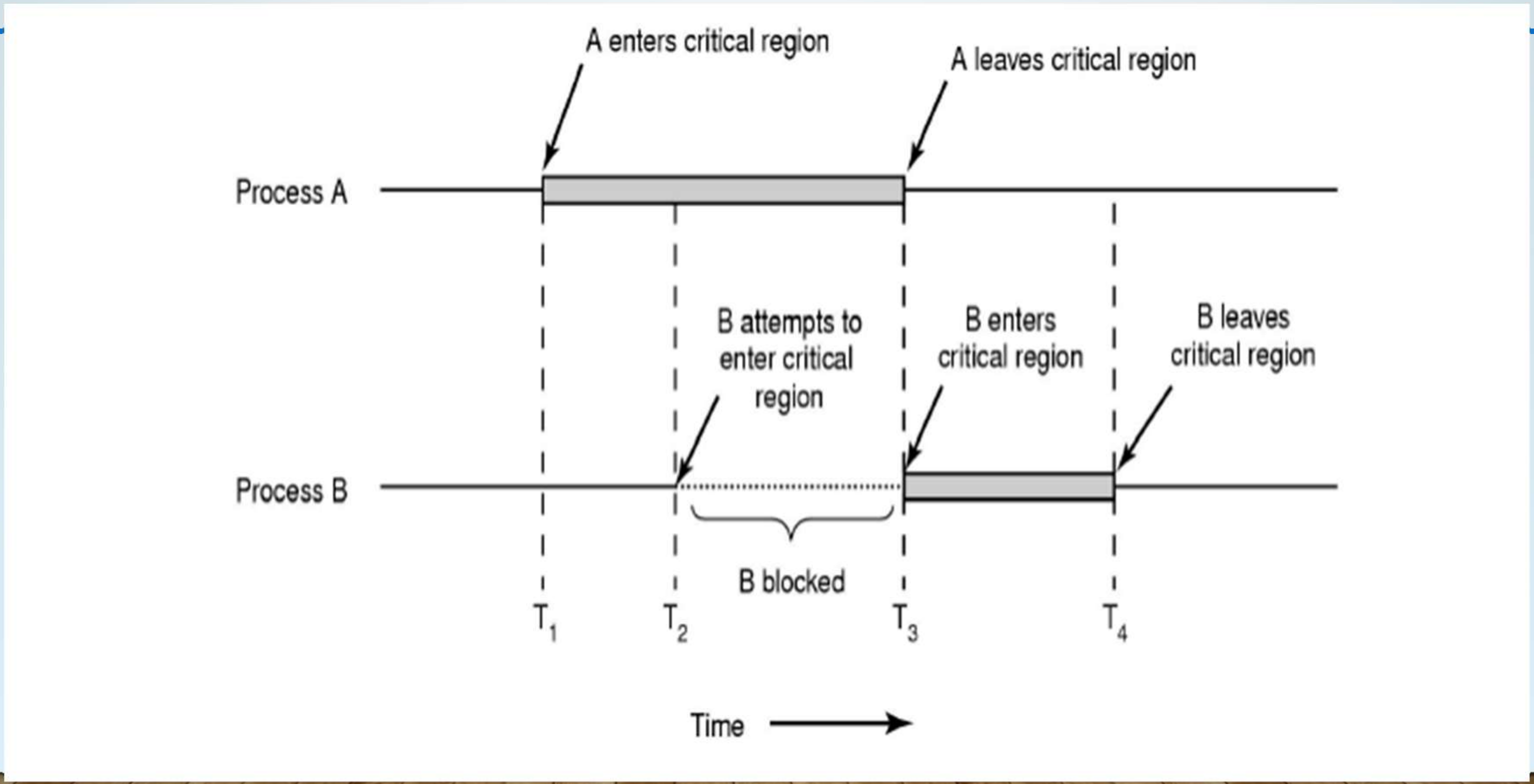
The critical section problem needs a solution to synchronize the different processes. The solution to the critical section problem must satisfy the following conditions −

- **Mutual Exclusion:** Mutual exclusion implies that only one process can be inside the critical section at any time. If any other processes require the critical section, they must wait until it is free.

- **Progress:** Progress means that if a process is not using the critical section, then it should not stop any other process from accessing it. In other words, any process can enter a critical section if it is free.

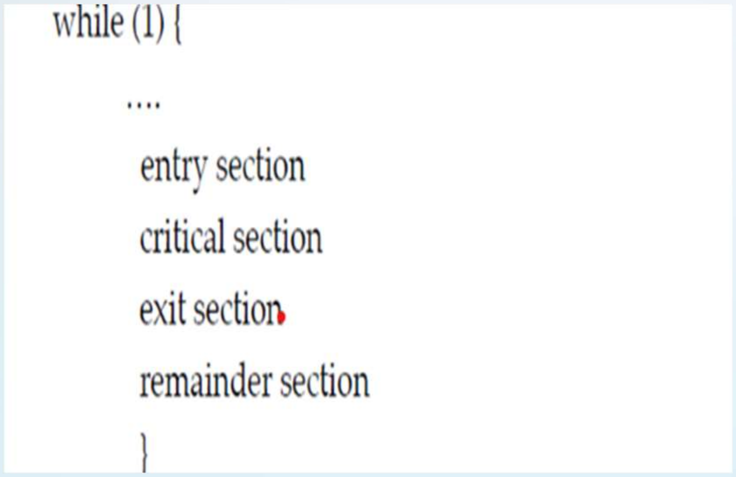
- **Bounded Waiting:** Bounded waiting means that each process must have a limited waiting time. It should not wait endlessly to access the critical section.
- No Assumption related to hardware or speed

CRITICAL SECTION AND MUTUAL EXCLUSION



Source: https://slideplayer.com/slide/10458277/

General Structure of a typical process



```
while (1) {
    ....
     entry section
     critical section
     exit section
     remainder section
     }
```

Ref. Book by Galvin

SOLUTION TO CRITICAL SECTION PROBLEM

- Synchronization :
- Lock variable
- Hardware Synchronization (Interrupts)
- Peterson’s Solution
- Semaphores

- **Lock variable**

```
do{
    acquire lock
    CS
    release lock
}
```

1. while (Lock==1); // entry code
2. Lock = 1
3. CS
4. Lock = 0 // exit code

Analyse if mutual exclusion is guaranteed or not

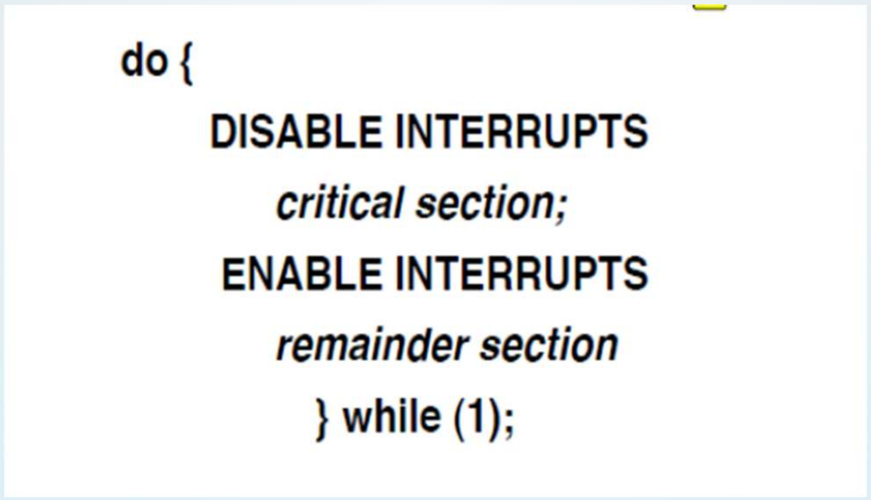
Mutual exclusion is not guaranteed so what is the solution to the problem

MUTUAL EXCLUSION WITH TEST AND SET

```
• do
• { while(TestAndSet(&Lock)) ; // do nothing
•     // critical section
•   lock=FALSE;
•     //remainder section
• } while (TRUE);
•
```

```
•   boolean    TestAndSet(boolean *target)
• {
•      boolean rv=*target;
•       *target = TRUE;
•       return   rv;
• }
```

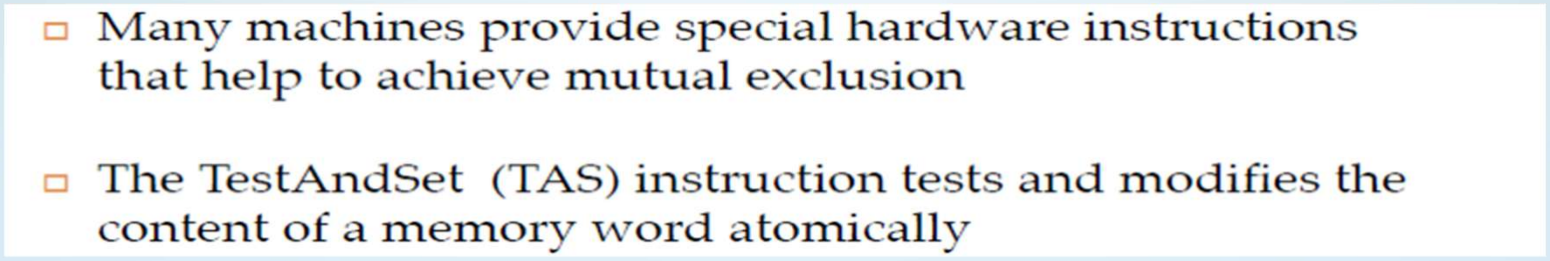
Getting help from hardware



```
do {
        DISABLE INTERRUPTS
            critical section;
         ENABLE INTERRUPTS
            remainder section
              } while (1);
```

Because of interrupts, CPU makes switching from one process to another. When interrupts are disabled, CPU switching will not happen

In lock variable mechanism, sometimes Process reads the old value of lock variable and enters the critical section. Due to this reason, more than one process might get into critical section. The solution is to use TAS or TSL Mechanism

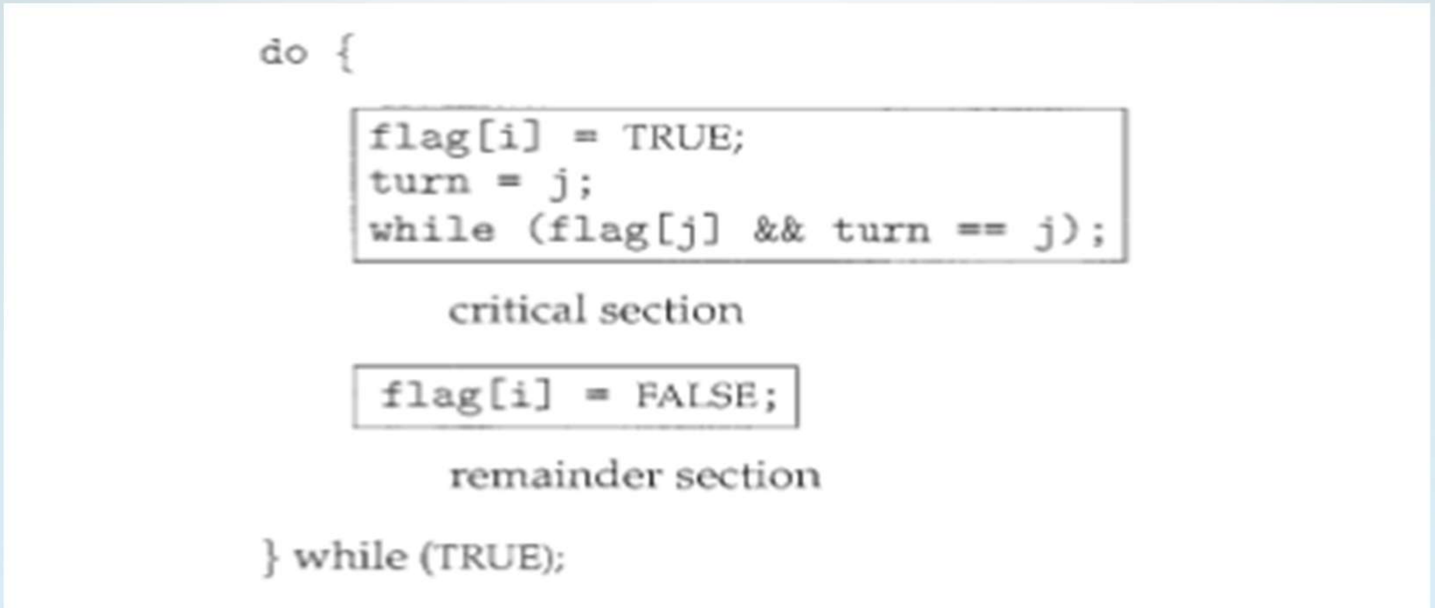


- Many machines provide special hardware instructions that help to achieve mutual exclusion
- The TestAndSet (TAS) instruction tests and modifies the content of a memory word atomically

PETERSON’S SOLUTION

- It Is a classic software-based solution for Critical Section problem.
- Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections. The processes are ~~numbered P0 and P1. For convenience, when presenting Pi, we use Pj to denote~~ the other process; that is, j equals 1 - i.
- Peterson’s solution requires the two processes to share two variables:
    - int turn;
    - Boolean flag[2]   Initially: flag[0]=flag[1]= false (because initially no one is interested in entering the CS)
- The variable turn indicates which of the two processes is allowed to enter its critical section first. That is, if turn == i, then process Pi is allowed to execute in its 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 to enter critical section!

To enter the critical section, process Pi first sets flag [i] to be true and then sets turn to the value j, thereby asserting that if the other process wishes to enter the critical section, it can do so.

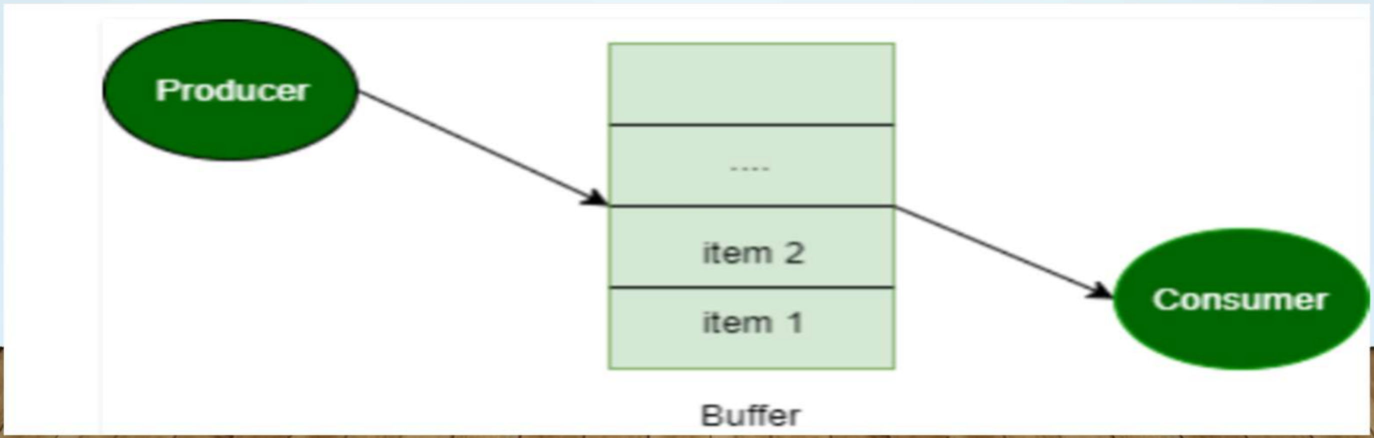


```
do {

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

        critical section

    flag[i] = FALSE;

        remainder section

} while (TRUE);
```

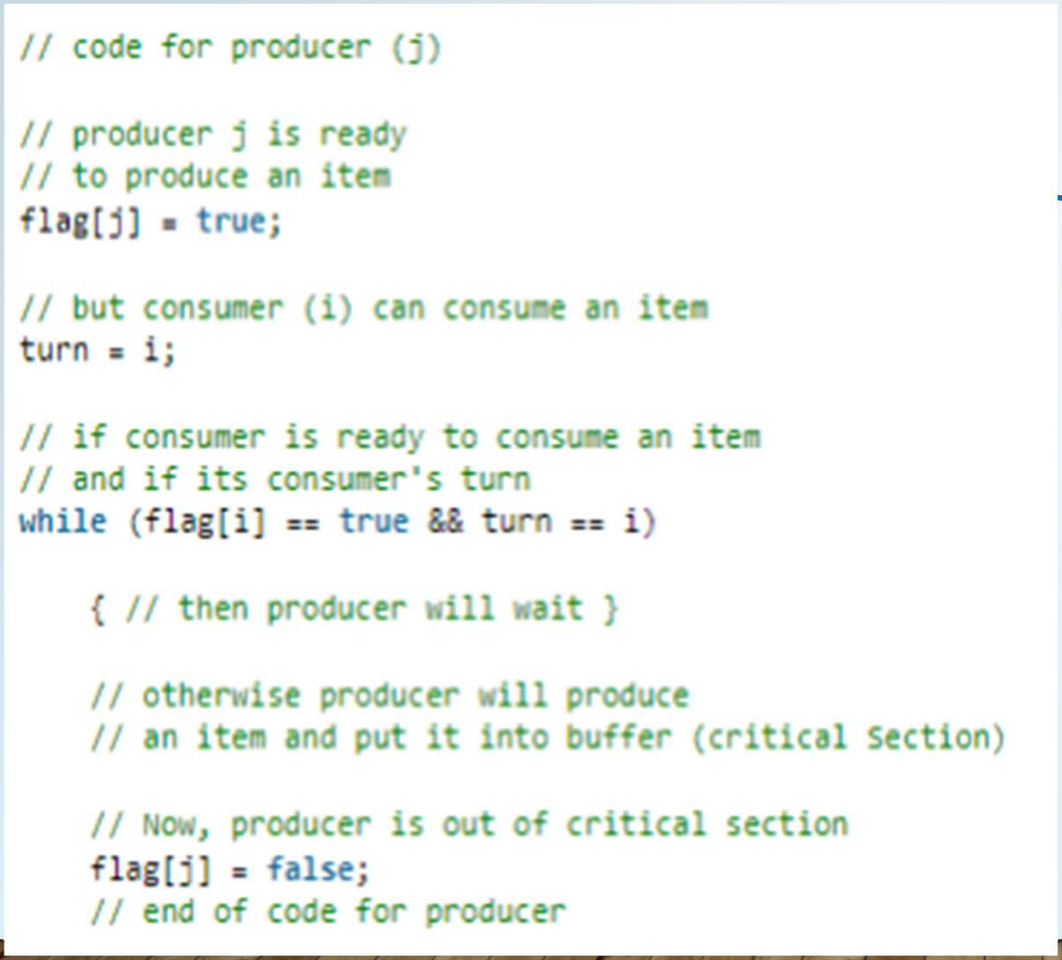
Figure 6.2 The structure of process P_i in Peterson's solution.

PERTERSON'S ALGORITHM IN PROCESS SYNCHRONIZATION

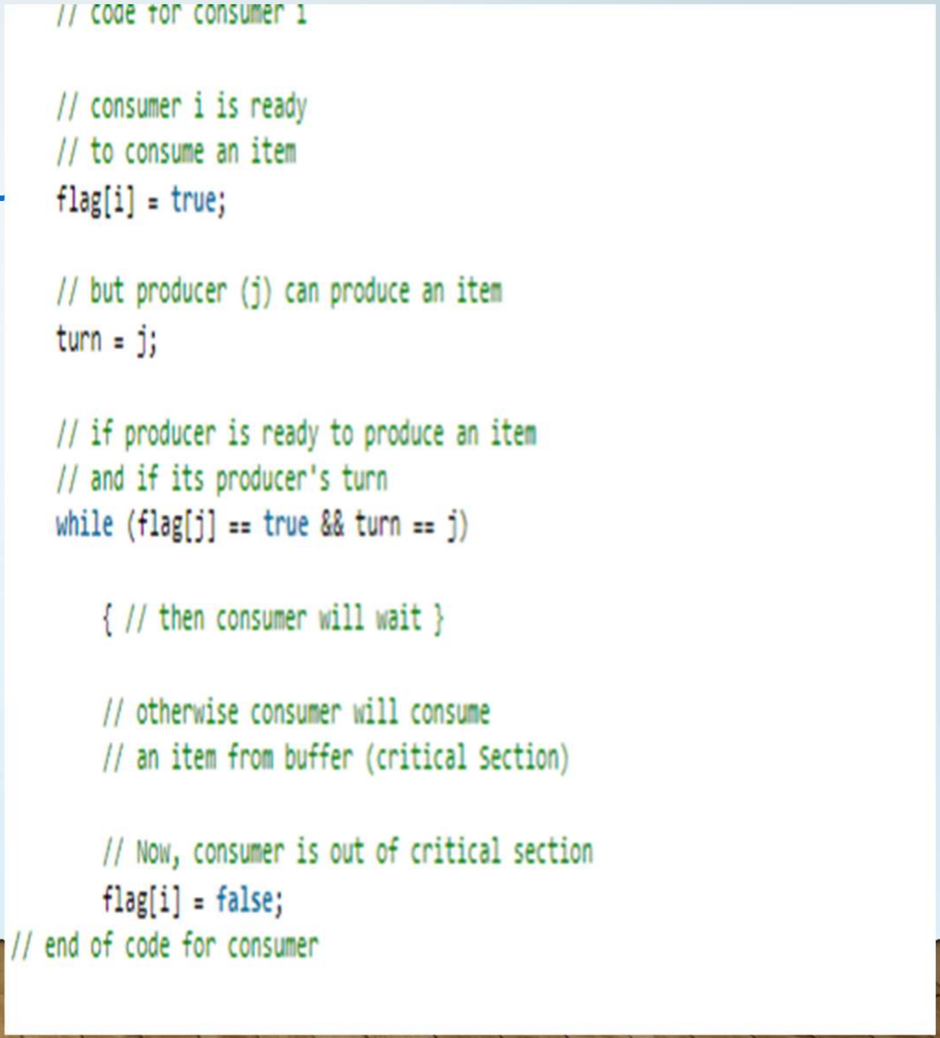
- **Problem:** The producer consumer problem (or bounded buffer problem) describes two processes, the producer and the consumer, which share a common, fixed-size buffer used as a queue. Producer produce an item and put it into buffer. If buffer is already full then producer will have to wait for an empty block in buffer. Consumer consume an item from buffer. If buffer is already empty then consumer will have to wait for an item in buffer. Implement Peterson's Algorithm for the two processes using shared memory such that there is mutual exclusion between them. The solution should be free from synchronization problems.



PETERSON'S ALGORITHM –



```
// code for producer (j)

// producer j is ready
// to produce an item
flag[j] = true;

// but consumer (i) can consume an item
turn = i;

// if consumer is ready to consume an item
// and if its consumer's turn
while (flag[i] == true && turn == i)

    { // then producer will wait }

    // otherwise producer will produce
    // an item and put it into buffer (critical Section)

    // Now, producer is out of critical section
    flag[j] = false;
    // end of code for producer
```



```
   // code for consumer i

   // consumer i is ready
   // to consume an item
   flag[i] = true;

   // but producer (j) can produce an item
   turn = j;

   // if producer is ready to produce an item
   // and if its producer's turn
   while (flag[j] == true && turn == j)

       { // then consumer will wait }

       // otherwise consumer will consume
       // an item from buffer (critical Section)

       // Now, consumer is out of critical section
       flag[i] = false;
// end of code for consumer
```

Explanation of Peterson’s algorithm –

Peterson’s Algorithm is used to synchronize two processes. It uses two variables, a bool array **flag** of size 2 and an int variable **turn** to accomplish it.

In the solution i represents the Consumer and j represents the Producer. Initially the flags are false. When a process wants to execute its critical section, it sets its flag to true and turn as the index of the other process. This means that the process wants to execute but it will allow the other process to run first. The process **performs busy waiting** until the other process has finished its own critical section.

After this the current process enters its critical section and adds or removes a random number from the shared buffer. After completing the critical section, it sets its own flag to false, indication it does not wish to execute anymore.

flag: array [0 1] of boolean

```
turn : 0,1
begin flag [0] : false
          flag [1]:=false
repeat for process i,j  (i=0, j=1)
   flag [0]=true
  turn = 1
  while flag [1] and turn =1
            do {nothing};
{CS}
flag [0]: = false
{remainder section}
```

```
repeat for process 1
  flag [1]=true
 turn = 0
 while flag [0] and turn =0
          do {nothing};
{CS}
flag [1]: = false
{remainder section}
```

Prove that this ensures three properties

```
do {
flag[1]= True
turn=2
while (flag[2] && turn = =2); // p2 will not enter p1 will
CS
flag [1] =false
RS
}
while true
```

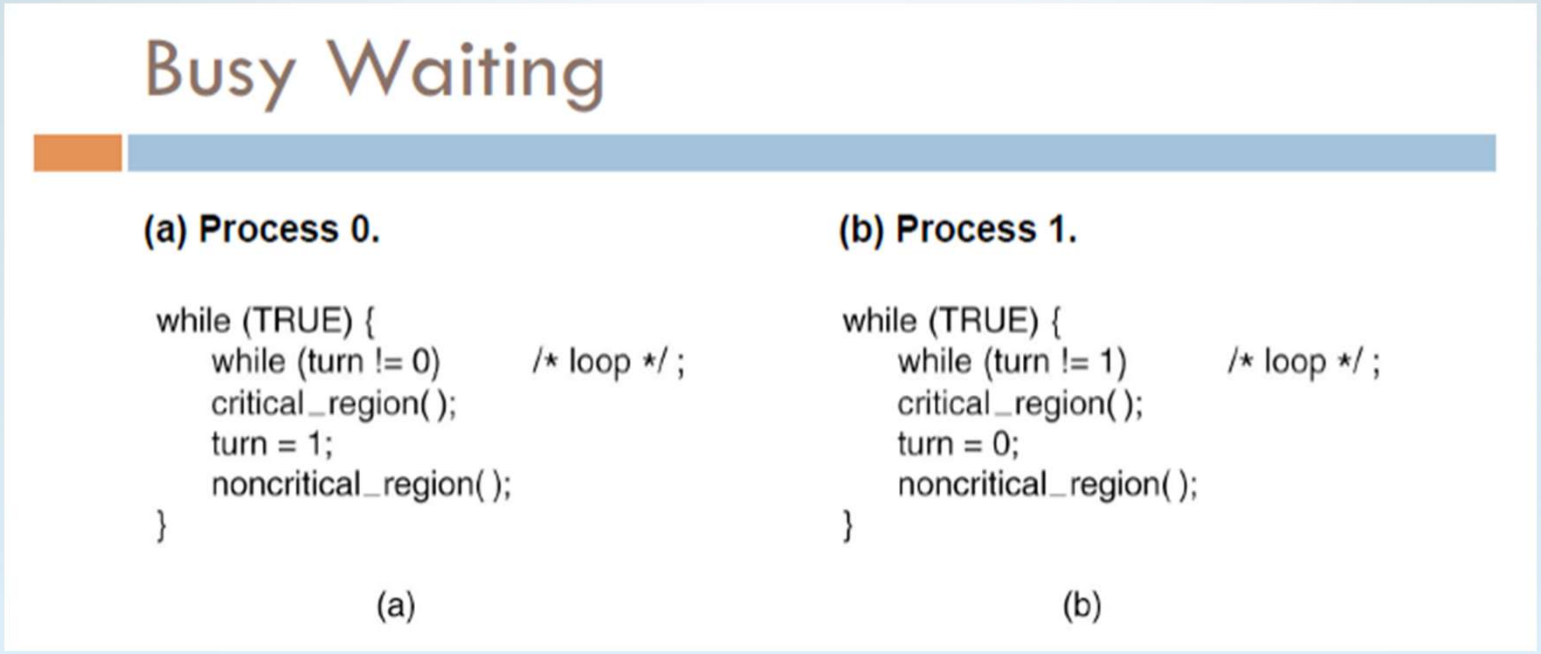
```
P2 came
flag [1] = false
turn = 2
while (flag [2] && turn ==2) // p2 enters
CS
```

- Peterson’s Solution preserves all three conditions :
  - Mutual Exclusion is assured as only one process can access the critical section at any time.
  - Progress is also assured, as a process outside the critical section does not block other processes from entering the critical section.
  - Bounded Waiting is preserved as every process gets a fair chance.

- Disadvantages of Peterson’s Solution
  - It involves Busy waiting
  - It is limited to 2 processes.
  - Software-based solutions like Peterson's are not guaranteed to work on modern computer architectures

- Both Peterson's and the TSL solutions solve the mutual exclusion problem.

- However, both of these solutions have the problem of **busy waiting**. That is, if the process is not allowed to enter its critical section it sits in a tight loop waiting for a condition to be met. This is obviously wasteful in terms of CPU usage.



Busy Waiting

(a) Process 0.

```
while (TRUE) {
    while (turn != 0)        /* loop */ ;
    critical_region( );
    turn = 1;
    noncritical_region( );
}
```

(a)

(b) Process 1.

```
while (TRUE) {
    while (turn != 1)        /* loop */ ;
    critical_region( );
    turn = 0;
    noncritical_region( );
}
```

(b)