Marwadi University

Unit 3
Inter Process Communication (Part-1)

Prepared By: Prof. Foram Chovatiya
Computer Engineering Dept.

Inter Process Communication (IPC)

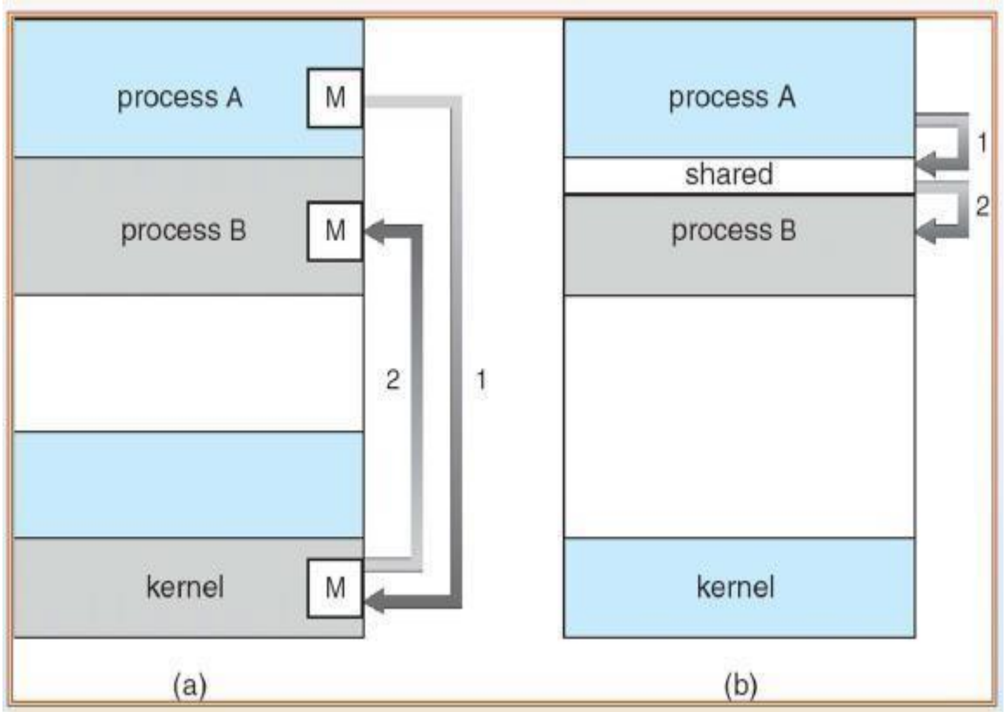
- **Exchange of data** between two or more separate, independent **processes/threads**.
- ***IPC is a mechanism which provides the communication between the process and synchronized their action.***

Inter Process Communication (IPC)

Operating systems provide facilities/resources for inter-process communications (**IPC**), such as

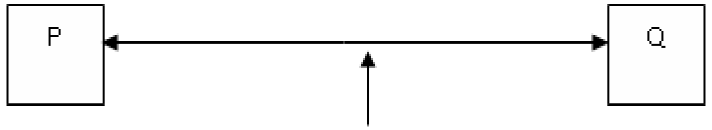
(a)Message queues

(b)Shared memory



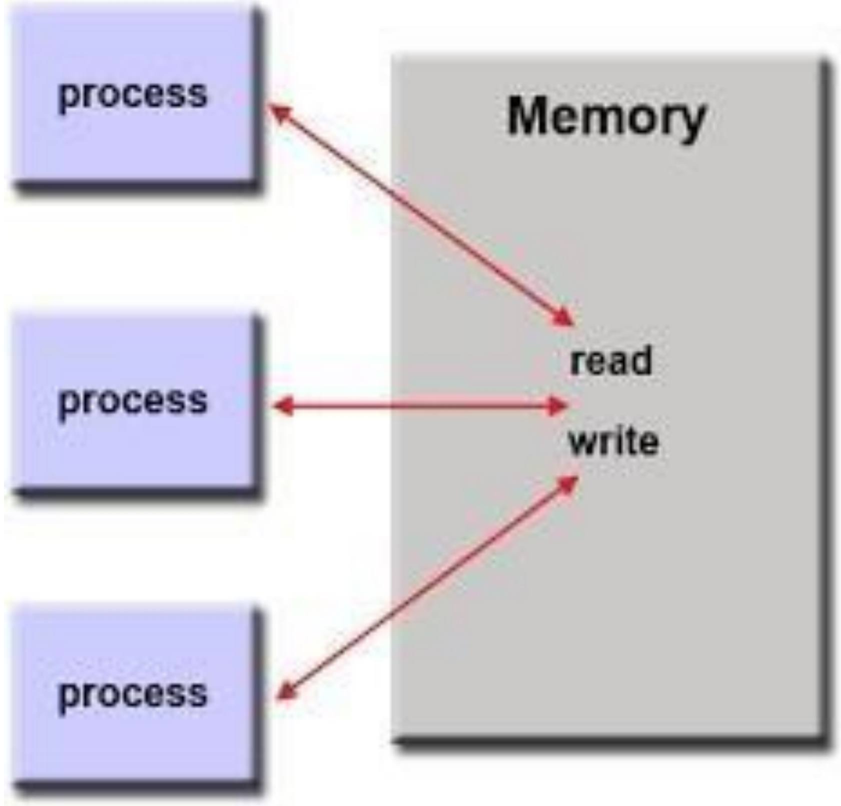
Message System

- In messagepassing system no requirement of share variable.
- IPC provide two operation send (message) and receive (message).
- Message sent by a process may be fixed size or variable size.
- If process P and Q want to communicate, they send message to each other using communication link.
- Communication link may be unidirectional or bidirectional.



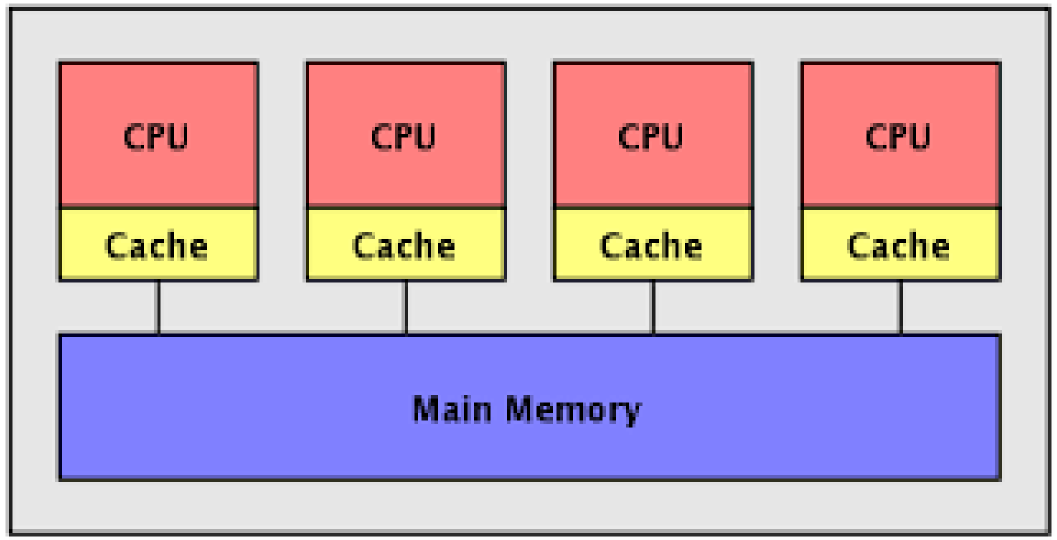
Shared Memory

- **Shared memory** is memory that may be simultaneously accessed by multiple programs with an intent to provide communication among them or avoid redundant copies.



Shared Memory

- Shared memory refers to a (typically large) block of random access memory (RAM) that can be accessed by several different processing units (CPUs) in a multiple –processor computer system.

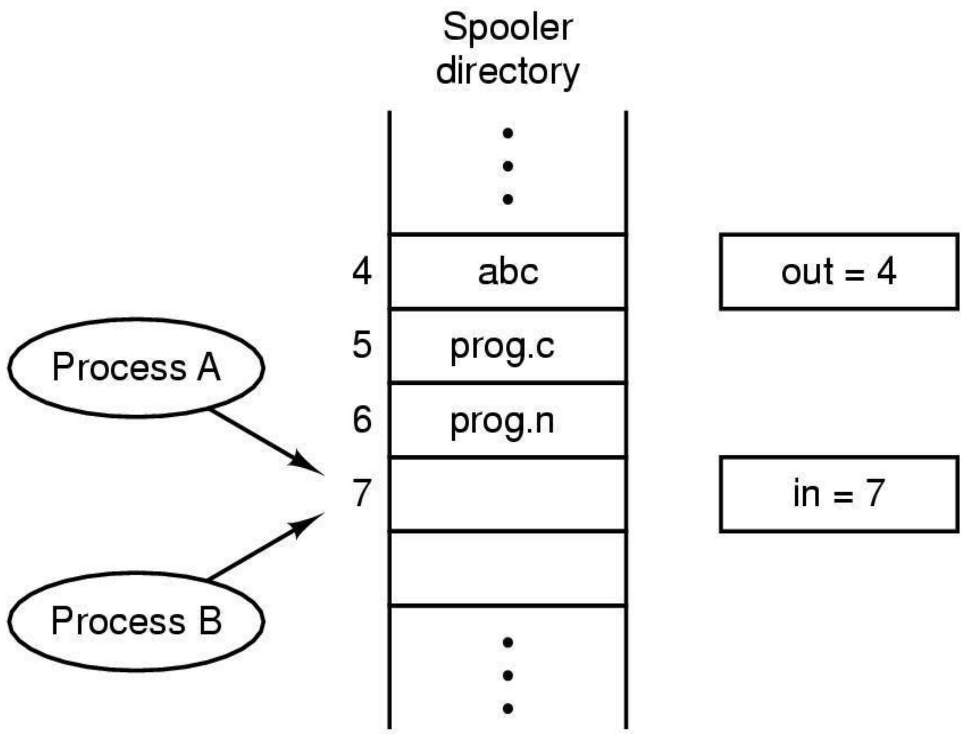


Race Condition

Race Condition

- Race conditions arise in software when separate processes or threads execution depend on some shared state.

Figure : Two processes want to access shared memory at the same time



Race Condition

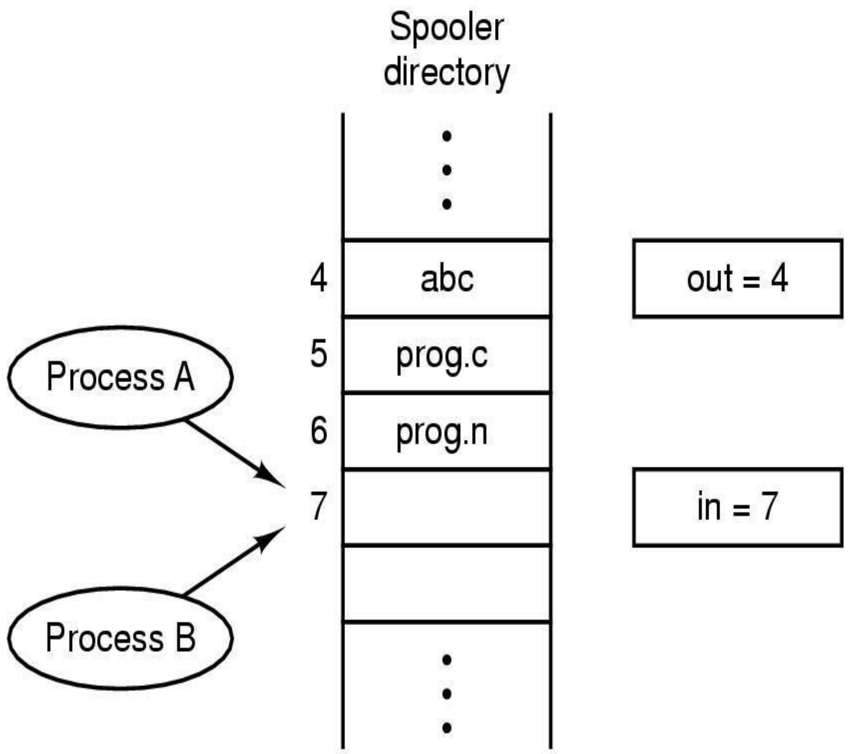
- A process wants to print a file.
- It enters the file name in a special **printer directory**
- The **Printer Daemon** periodically checks to see if there is any file to print
- If any file is there the Printer Daemon prints them and removes their names from the directory

Race Condition

- Imagine our directory has a very large number of slots (numbered 0,1,2,…) and each one can hold a file name
- There are two shared variables- out that points to the next file to be printed and in that points to the next free slot in the directory

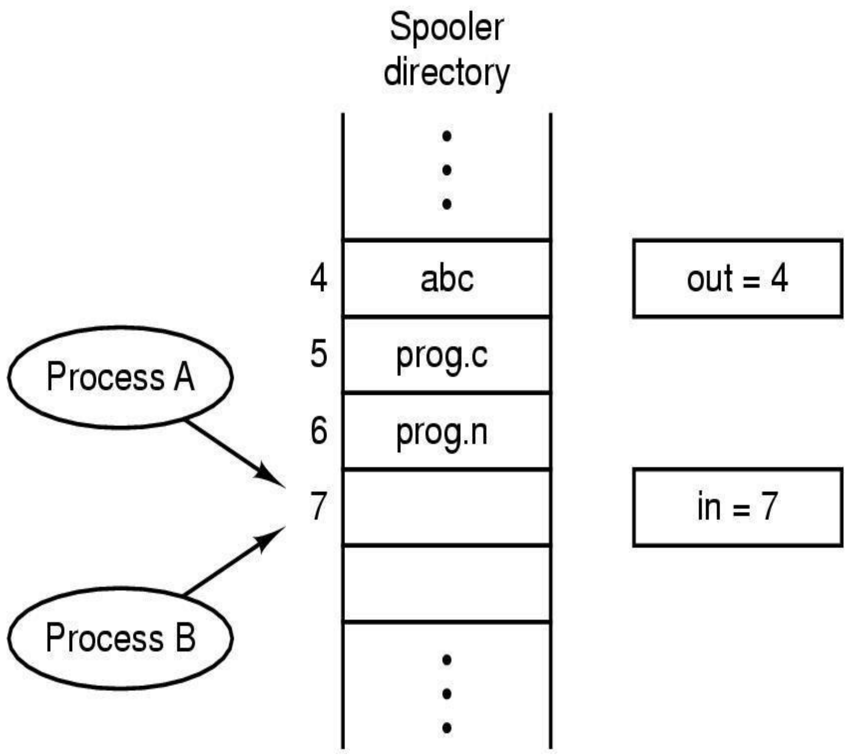
Race Condition

- **A** reads in and stores the values 7 to its local variable
- Then a context switch from A to B occurs
- **B** also reads in and stores the value 7 to its local variable
- **B** continues to run and it stores the name of its file in slot 7 and updates in to 8.
- Then it goes off and does other things



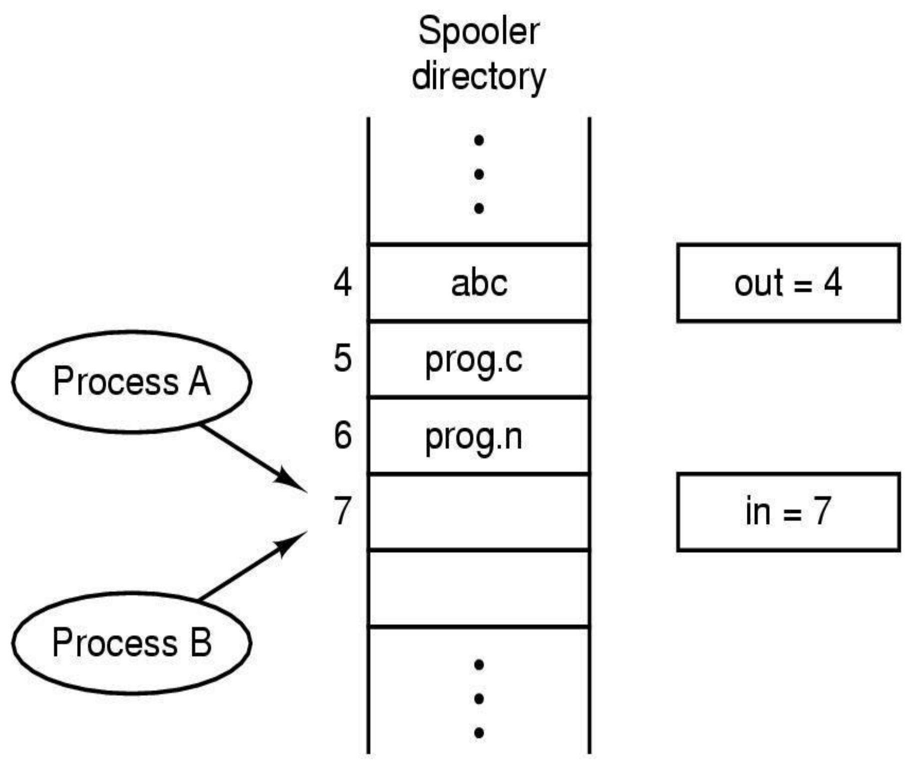
Race Condition

- **A** runs again starting from the place it left off
- It looks its local variable and finds 7 there and writes the file name in slot 7
- Then **A** sets in to 8
- As everything went fine, the printer daemon will not raise any error



Race Condition

- Process B never gets the chance
- Situations like this where two or more processes are reading or writing some shared data and the final result depends on who ran precisely are called **Race Conditions**



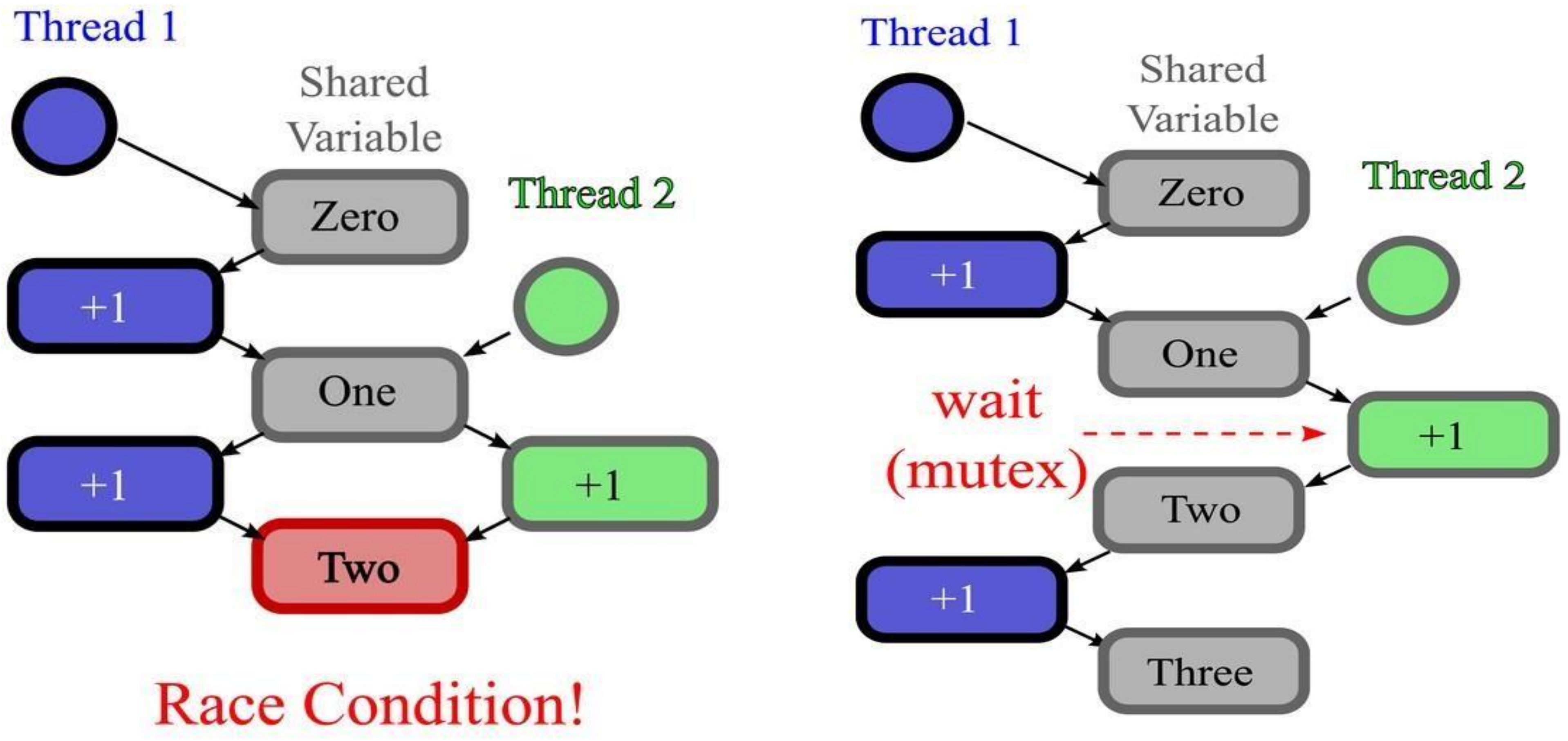
Example-2 Race Condition

A race condition may occur if commands to read and write a large amount of data are received at almost the same instant, and the machine attempts to overwrite some or all of the old data while that old data is still being read.

The result may be one or more of the following:

- notification and shutdown of the program
- errors reading the old data or errors writing the new data.

Race Condition

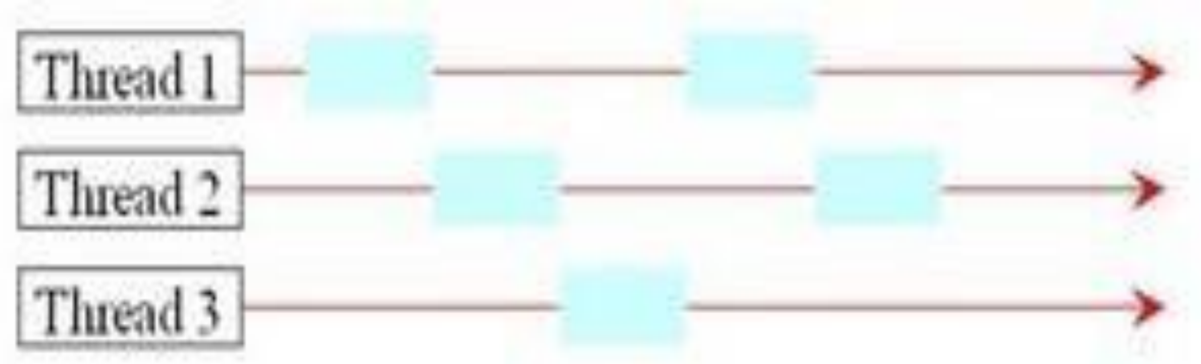


Marwadi University

Synchronization

Synchronization

To Prevent Race Condition, **Concurrent Process must be Synchronized.**



What is Concurrency?

- *When multiple processes are working simultaneously in system, is called concurrency or process synchronization.*
- When two or more process want to access same resource than , synchronization is required between the processes.

When Synchronization required ?

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 arises in the case of Cooperative process because resources are shared in Cooperative processes.

Synchronization

Examples where process synchronization is required:

- In following example process P1 and process P2 are changing a shared variable X at a time.

Suppose value of x is 50.

Process P1:	Process P2:
Read X;	Read X;
X=X+100;	X=X-200;
Write X;	Write X;

If these processes will not execute in proper order than output can be wrong.

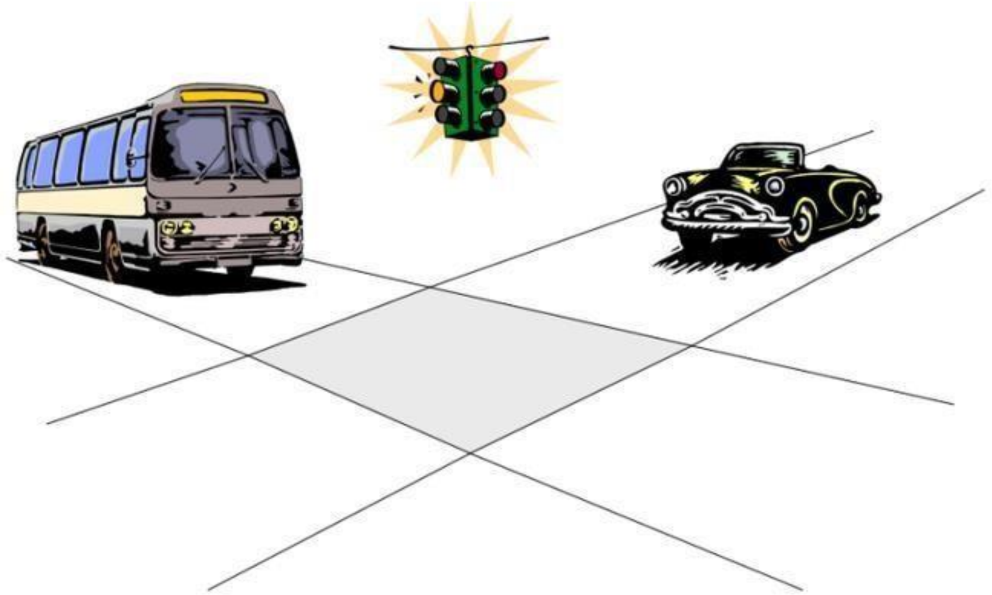
- After execution of process P1 and P2, **value of X should be -50.**

Critical Section Problem

Critical Section

Critical Resource:

When 2 or more processes want to access same resource, then this resource is called critical resource.



Critical Section

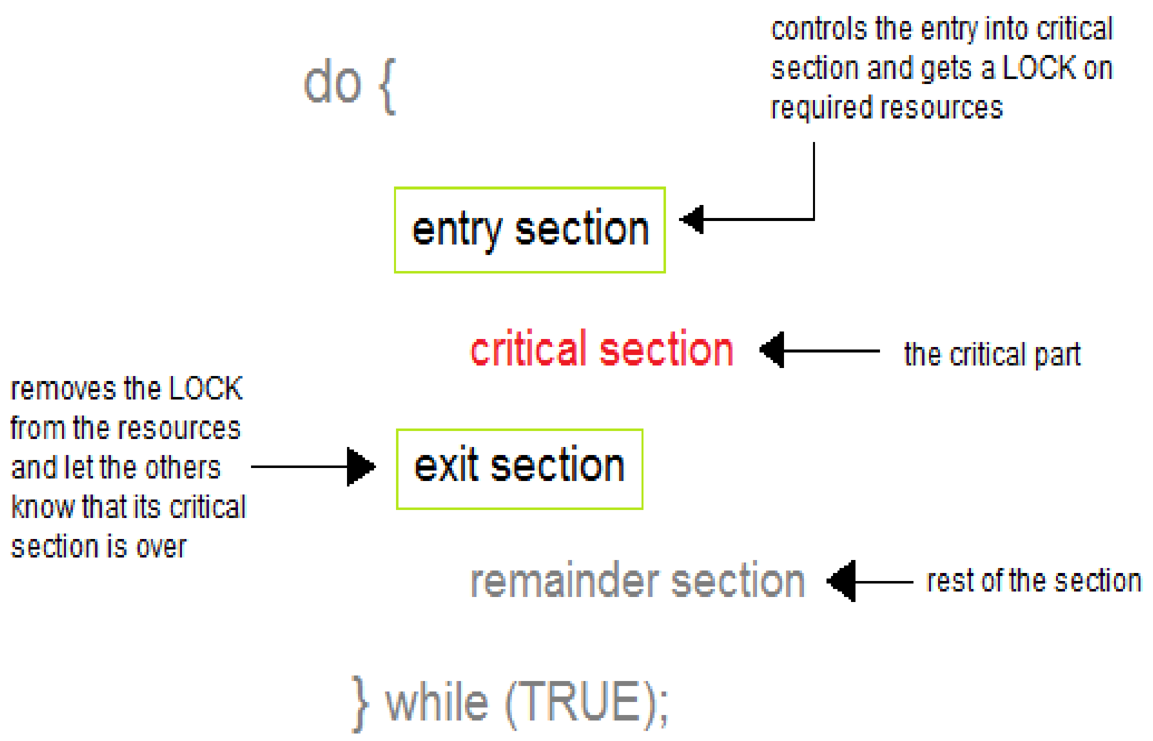
Critical Section / Critical Region

- It is Region, where process will change the value of shared variable or shared resource will be accessed.
- *Only one process should be allowed to execute its critical section.*

- For process synchronization ,Operating system will allowed only one process to execute its critical section.

Critical Section Problem

- At a given point of time, only one process must be executing its critical section.
- If any other process also want to execute its critical section, it must wait until the first one finishes


controls the entry into critical section and gets a LOCK on required resources
do {
entry section
critical section
the critical part
removes the LOCK from the resources and let the others know that its critical section is over
exit section
remainder section
rest of the section
} while (TRUE);

Mutual Exclusion

Solution to Critical Section Problem

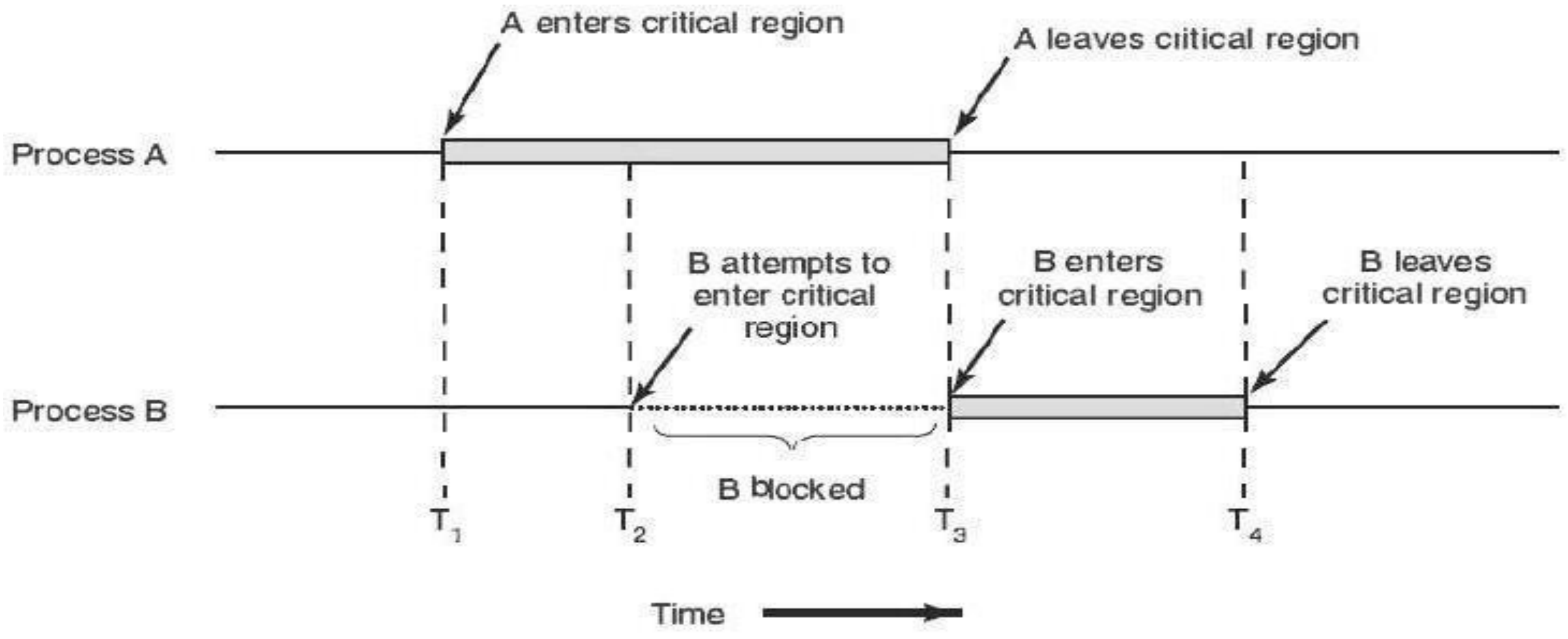
Mutual Exclusion : Out of a group of cooperating processes, only one process can be in its critical section at a given point of time.

Progress : If no process is in its critical section, and if one or more threads want to execute their critical section then any one of these threads must be allowed to get into its critical section.

Bounded Waiting : Bounded waiting means that each process must have a limited waiting time. It should not wait endlessly to access the critical section.

Mutual Exclusion

- During the concurrent process, system allowed only process to access the resource or execute critical section, this procedure is called mutual exclusion.



Mutual Exclusion

Different Proposal to Achieve Mutual Exclusion

- Disabling Interrupts
- Lock Variables
- Strict Alteration
- Petersen’s Solution
- The TSL Instruction

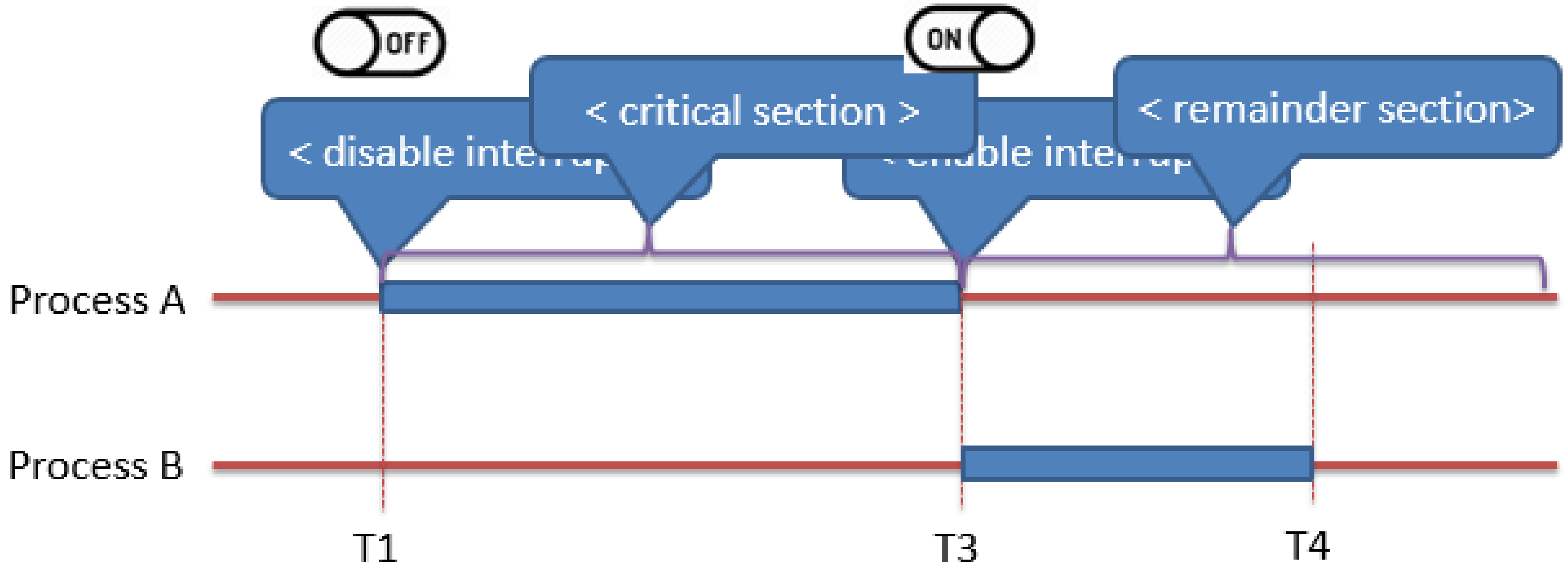
Mutual Exclusion

Different Proposal to Achieve Mutual Exclusion

- Disabling Interrupts
- Lock Variables
- Strict Alteration
- Petersen’s Solution
- The TSL Instruction

Disabling Interrupts

- Disable all interrupts
- With interrupts disabled, no clock interrupts can occur.
- Thus, once a process has disabled interrupts, it can examine and update the shared memory without fear that any other process will intervene.



Disabling Interrupts

Disadvantages

- Unattractive to give user processes the power to turn off interrupts

  What if one of them did it (disable interrupt) and never turned them on (enable interrupt) again ? That could be end of system.
- If the system is a multiprocessor , with two or more CPUs, disabling interrupts affects only the CPU that executed the disable instruction. The other ones will continue running and can access the shared memory

Mutual Exclusion

Different Proposal to Achieve Mutual Exclusion

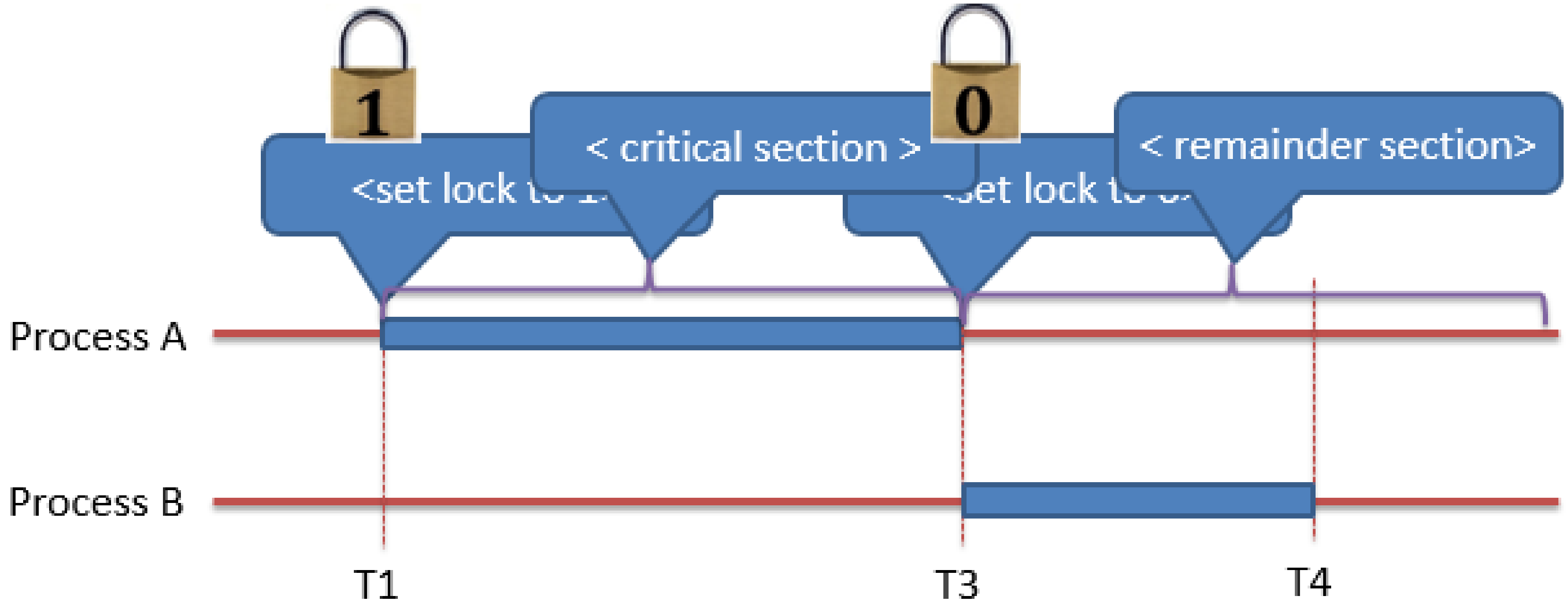
- Disabling Interrupts
- Lock Variables
- Strict Alteration
- Petersen’s Solution
- The TSL Instruction

Lock Variables

- Consider having a single, shared (lock) variable, initially 0. When a process wants to enter its critical region, it first tests the lock.

- If the lock is 0, the process sets it to 1 and enters the critical region. If the lock is already 1, the process just waits until it becomes 0. Thus, a 0 means that no process is in its critical region, and a 1 means that some process is in its critical region.

Lock Variables



Lock Variables

Disadvantage

- If process P0 sees the value of lock variable 0 and before it can set it to 1, context switch occurs. (OS preempt CPU)
- Now process P1 runs and finds value of lock variable 0, so it sets value to 1, enters critical region.
- At some point of time P0 resumes, sets the value of lock variable to 1, enters critical region.
- Now two processes are in their critical regions accessing the same shared memory, which violates the mutual exclusion condition.

Mutual Exclusion

Different Proposal to Achieve Mutual Exclusion

- Disabling Interrupts
- Lock Variables
- Strict Alteration
- Petersen’s Solution
- The TSL Instruction

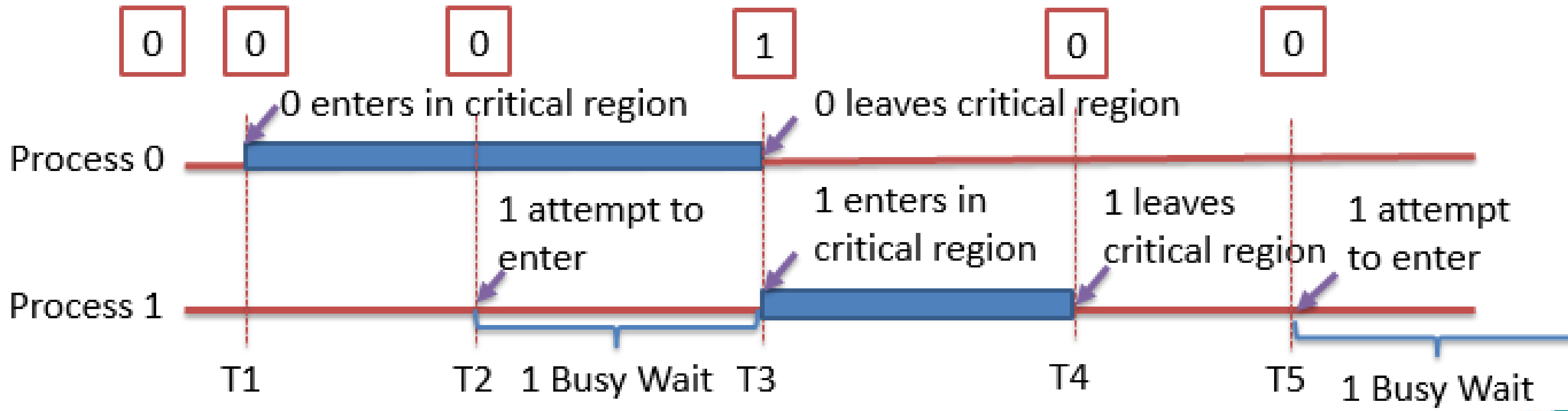
Strict Alteration

- The integer variable **Flag,** initially 0, keeps track of whose turn it is to enter the critical region and examine or update the shared memory.

- Initially process 0 inspects Flag, finds it to be 0, and enters its critical region. Process 1 also finds it to be 0 and therefore sits in a tight loop continually testing Flag to see when it becomes 1.

- Continuously testing a variable until some value appears is called **busy waiting**. It should usually be avoided, since it wastes CPU time.

Strict Alteration



Strict Alteration

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

(a)
```

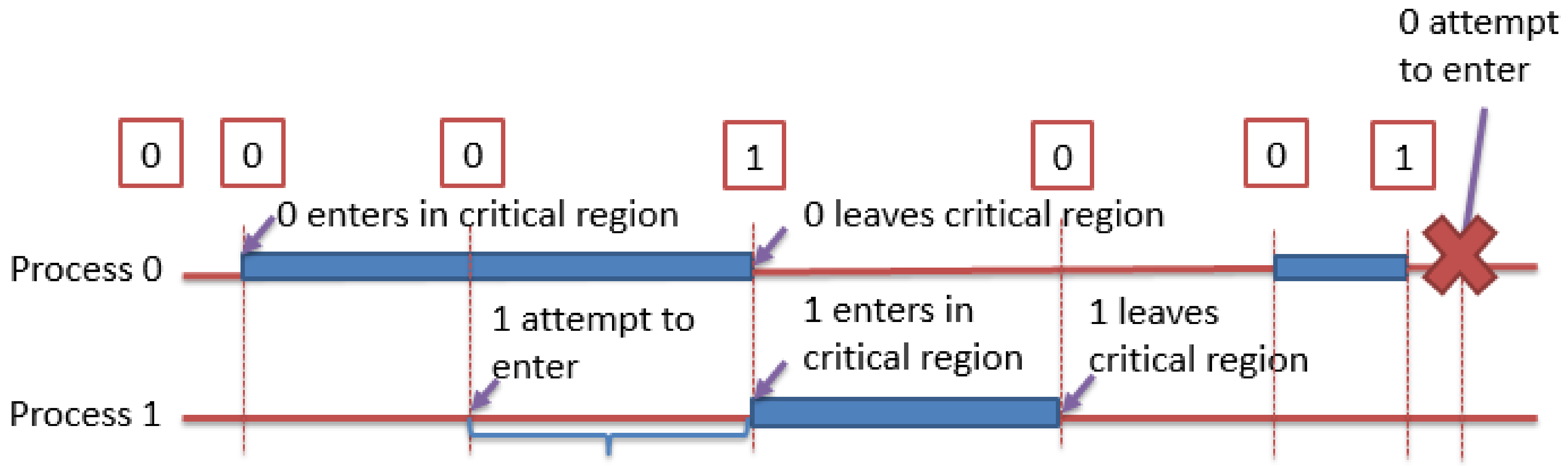
```
while (TRUE) {
while (Flag != 1); /* loop */;
critical_region();
Flag = 0;
noncritical_region();
}

(b)
```

(a) Process 0. (b) Process 1

Strict Alteration

Disadvantage:

- It is limited to 2 processes
- Busy Waiting
- Progress is not achieved ➔ Process 0 is blocked by Process 1



Mutual Exclusion

Different Proposal to Achieve Mutual Exclusion

- Disabling Interrupts
- Lock Variables
- Strict Alteration
- Petersen’s Solution
- The TSL Instruction

Peterson’s Solution

- Peterson’s Solution is a classical software based solution to the critical section problem.
- In Peterson’s solution, we have two shared variables:

  boolean INTERESTED : Initialized to FALSE, initially no one is interested in entering the critical section

  int TURN : The process whose turn is to enter the critical section.

Peterson's Solution

```
#define N 2
#define TRUE 1
#define FALSE 0
boolean INTERESTED[N] = FALSE
int TURN;
void Entry_Section(int process1)
{
    int process2;
    INTERESTED[process1] = TRUE;
    TURN  = process1;
    while(INTERESTED[process2] == TRUE && TURN = process1);
}
void Exit_Section(int process1)
{
    INTERESTED[process1] = FALSE;
}
```

Peterson's Solution

- 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 blocks other processes from entering the critical section.
  - **Bounded Waiting** is preserved as every process gets a fair chance.

Peterson’s Solution

Disadvantages of Peterson’s Solution

- It involves Busy waiting
- It is limited to 2 processes.

Mutual Exclusion

Different Proposal to Achieve Mutual Exclusion

- Disabling Interrupts
- Lock Variables
- Strict Alteration
- Petersen’s Solution
- The TSL Instruction

The TSL Instruction **(Test and Set)**

- Test and Set is a hardware solution to the synchronization problem.
- In Test And Set, shared lock variable which can take either of the two values, 0 or 1.
- Before entering into the critical section, a process inquires about the lock.
- If it is locked, it keeps on waiting till it become free and if it is not locked, it takes the lock and executes the critical section.

The TSL Instruction

```
enter_region:
tsl register, flag ;       copy flag to register and set flag to 1
cmp register, #0 ;         was flag zero?
jnz enter_region ;         if flag was non zero, lock was set , so loop
ret ;                      return     (enter critical region)

leave_region:
mov flag, #0 ;             store zero in flag
ret ;                      return
```

Thank you!!