Operating system

GATEWAY CLASSES

ONE SHOT REVISION

Unit-2

Concurrent Processes

CS / IT / CS Allied / MCA

By Dr. Kapil Kumar Sir

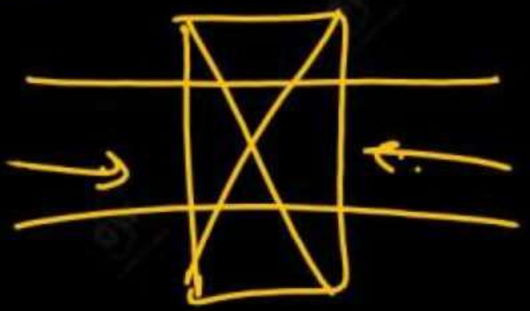
OPERATING SYSTEM

UNIT 2 – Concurrent Processes
SYLLABUS

Concurrent Processes: Process Concept, Principle of Concurrency, Producer/Consumer Problem, Mutual Exclusion, Critical Section Problem, Dekker's solution, Peterson's solution, Semaphores, Test and Set operation; Classical Problem in Concurrency- Dining Philosopher Problem, Sleeping Barber Problem; Inter Process Communication models and Schemes, Process generation.

Process Synchronization

- Processes need to communicate other processes for execution.
- For example the printout of a word document is needed. While working with Word process, a print dialog box (another process: printer driver process) will appear to facilitate the printout of the document. This communication will be provided by the Operating system.
- **Problems without synchronization:**
  - Inconsistency (Data inconsistency)
  - Loss of data
  - Deadlock etc.



- Process synchronization is required during the communication among the processes.

Serial and Parallel Mode of Execution

- Processes may be executed in serial mode or parallel mode.

- **Serial Mode Execution -**

- Serial mode means processes will be executed one by one.
- Process will start its execution and after completion of execution of one process, the execution of another process will start.
- Like use of ATM machine by the user or the costumer.
- In this processing, running of one process will not affect the running of another process.

- **Parallel Mode Execution -**

- In parallel execution, number of processes may be executed at a time, simultaneously in multiprogramming environment.

GW
GATEWAY CLASSES

Principle of Concurrency

- Concurrency is the execution of multiple instruction sequences at the same time.
- It happens in the operating system when there are several process threads running in parallel.
- Concurrency results in the sharing of resources resulting in problems like deadlocks and resource starvation.
- There are several motivations for allowing concurrent execution -

- **Physical Resource Sharing:** Multiuser environment since hardware resources are limited.
- **Logical Resource Sharing:** Shared file (same piece of information).
- **Computation Speedup:** Parallel execution.

- The principle of concurrency involves the idea that the operating system can manage and execute multiple tasks concurrently, allowing for efficient resource utilization and improved system responsiveness.
- The principle aims to maximize system throughput and performance by allowing tasks to execute independently and concurrently whenever possible.

Relationship Between Processes of Operating System

The processes executing in the operating system is one of the following two types:

1. Independent Processes:

- These processes do not communicate with other process, i.e. runs independently.
- Independent process do not share anything then no affect on another process.
- The result of execution depends only on the input state.
- The result of the execution will always be the same for the same input.
- The termination of the independent process will not terminate any other.
- Suppose one process in being executed on SBI- SERVER and another on HDFC-SERVER, then they are independent processes and will not affect to each other.

2. Cooperative Processes (coordinating or communicating processes):

- Its state is shared along other processes.
- The result of the execution depends on relative execution sequence and cannot be predicted in advanced(Non-deterministic).
- The result of the execution will not always be the same for the same input.
- The termination of the cooperating process may affect other process.
- A cooperative process is one that can affect or be affected by other process executing in the system, because they share something like variable, memory (buffer), code, resources (CPU, Printer, Scanner etc.)
- The sharing of anything may affect the execution of processes.
- Concurrent access to shared data may result in data inconsistency.

- Therefore, cooperative processes are need to synchronize in proper way.
- There are various mechanisms to ensure the orderly execution of cooperating processes that share a logical address space so that data consistency is maintained because concurrent access to shared data may result in data inconsistency.
- Consider the following example-

int shared = 5 // Global Variable

Process P1	Process P2
I1: int x = shared ;	I1: int y = shared ;
I2: X + + ;	I2: y − − ;
I3: Sleep (1) ;	I3: Sleep (1) ;
I4: shared = x ;	I4: shared = y ;

- Say P1 starts its execution first:

int shared = 5 // Global Variable

Process P1	Process P2
I1 : int x = shared ;	I1: int y = shared ;
I2 : X + + ;	I2: y − −;
I3 : Sleep (1) ;	I3 : Sleep (1) ;
I4 : shared = x ;	I4 : shared = y ;

- Sleep means it will be paused (preempted) for one unit of time, means now other process will be executed, context switch will take place.
- After execution, shared = 4, which is wrong, it should be 5, P1 is incrementing by 1 and P2 is decrementing by 1, i.e. it should remain 5.
- It shows that processes are not being executed in synchronized manner.

- Say now P2 starts its execution first:

int shared = 5 // Global Variable

Process P1	*Process P2*
I1 : int x = shared ;	I1: int y = shared ;
I2 : X + + ;	I2: y − −;
I3 : Sleep (1) ;	I3 : Sleep (1) ;
I4 : shared = x ;	I4 : shared = y ;

- After execution, shared = 6, which is wrong, it should be 5, P1 is incrementing by 1 and P2 is decrementing by 1, i.e. should remain 5. It shows that processes are not being executed in synchronized manner.

Imp

Race condition

- An undesirable situation where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a race condition. (as observed in the previous example)
- Consider the following another example and say Process P1 starts its execution first:

int X = 5 // Global Variable

Process P1		Process P2
R1 = 5	R1 = X	R2 = 5
R1 = 5 + 2 = 7	R1 = R1 + 2	R2 = 5 + 3 = 8
Interrupt		Interrupt
X = 7	X = R1	X = 8
		R2 = X
		R2 = 5

- Say now P2 starts its execution first:

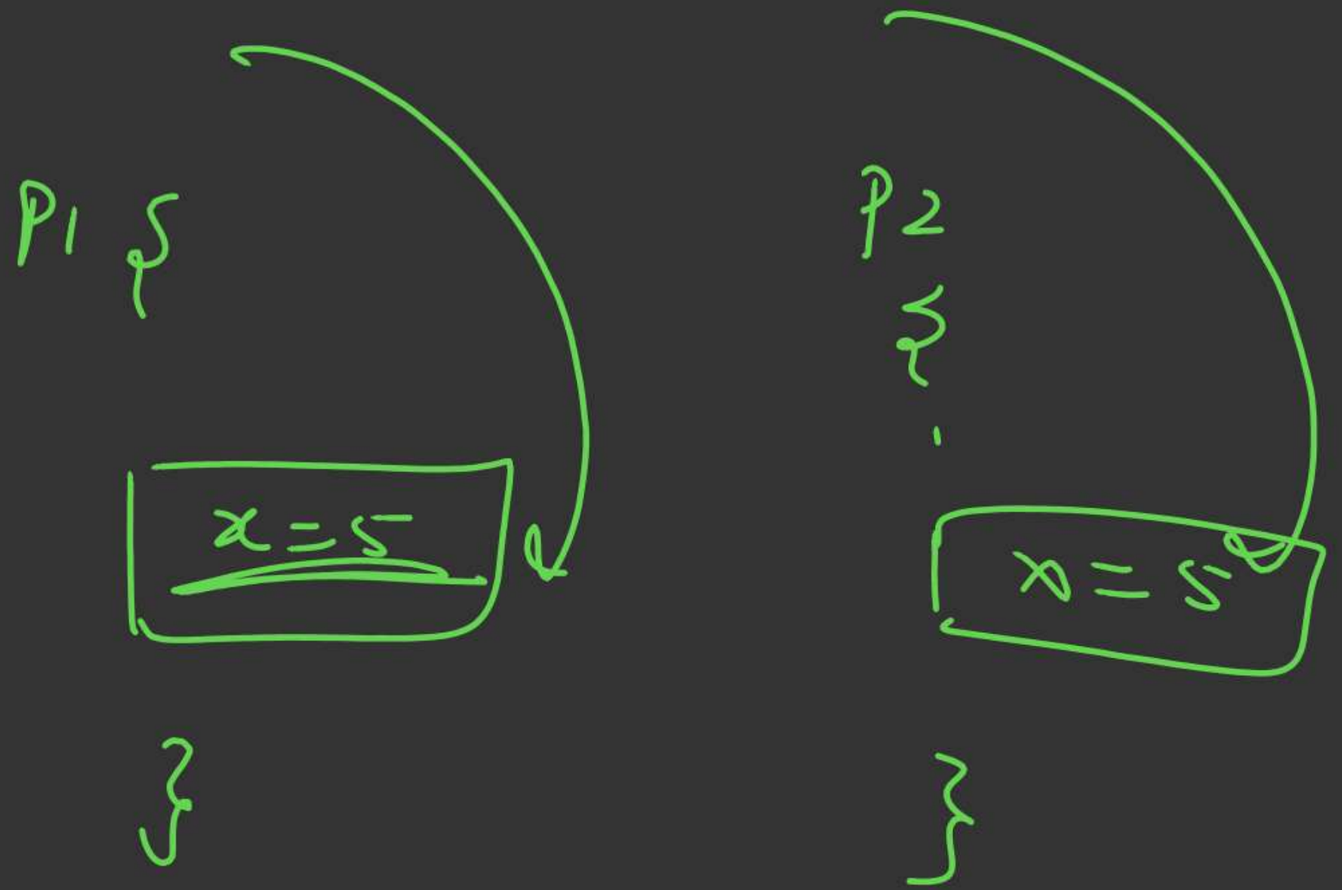
int X = 5 // Global Variable

Process P1		Process P2	
R1 = 5	R1 = X	R2 = 5	R2 = X
R1 = 5 + 2 = 7	R1 = R1 + 2	R2 = 5 + 3 = 8	R2 = R2 + 3
Interrupt		Interrupt	
X = 7	X = R1	X = 8	X = R2

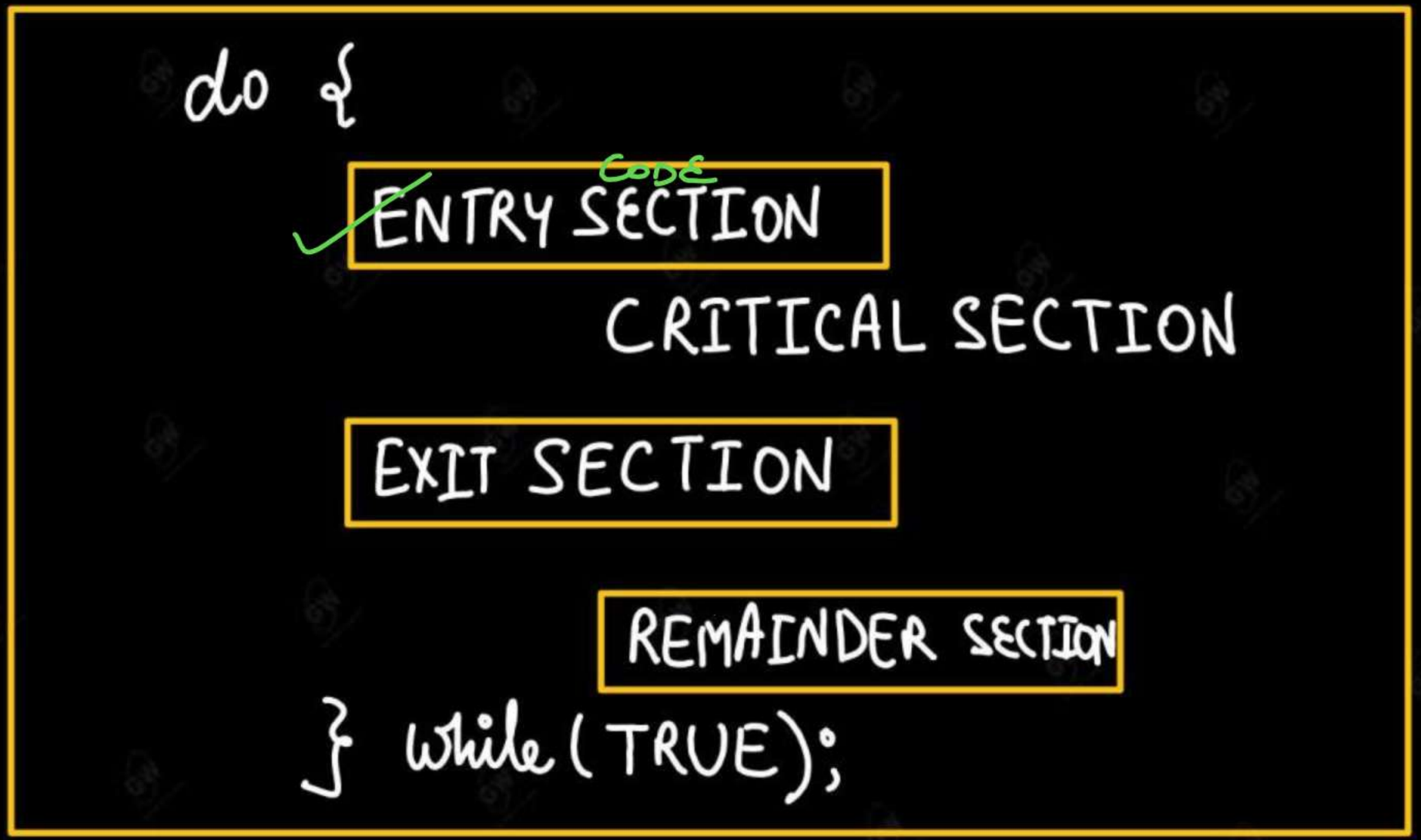
- To guard against the race condition, we need to ensure that only one process at a time can be manipulating the shared variable.
- To make such a guarantee, we require that the processes be synchronized in some way.
- Several algorithms can be used to achieve the synchronization among the cooperative processes.

The Critical Section Problem

- Consider a system consisting of n processes $\{P_0, P_1 \ldots \ldots P_{n-1}\}$.
- Each process has a segment of code called a critical section, in which the process may be changing common variables, updating a table, writing a file and so on.
- The critical section is a code segment where the shared variables can be accessed and this section needs synchronization.
- The important feature of the system is that, when a process is executing in its critical section, no other process is to be allowed to execute its critical section.
- That is no two processes are executing in their critical section at the same time.
- The critical section problem is to design a protocol that the processes can use to cooperate.
- Each process must request permission to enter the critical section.
- The section of code implementing this request in the "entry section".
- The critical section may be followed by an "exit section".
- The remaining code is the "remainder section".


P1 {
x = 5
}
P2 {
x = 5
}

The general structure of a process P_i is shown as follows:



Requirements must be satisfied by the solution to the Critical Section Problem

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

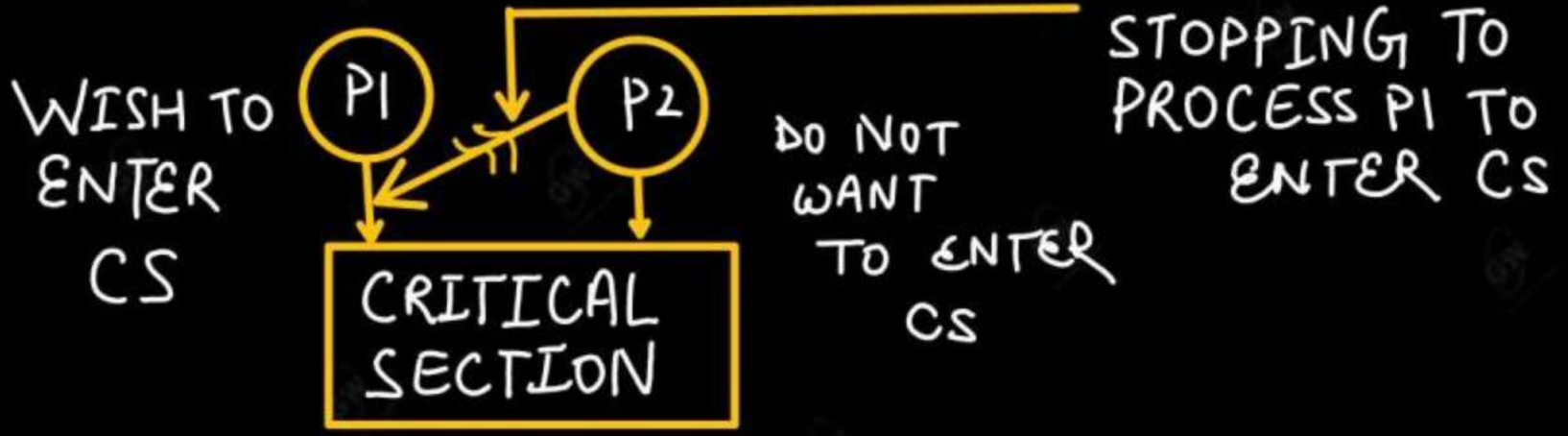
PR

1. Mutual Exclusion -

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

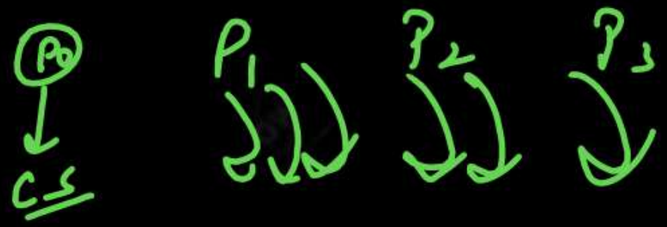
2. Progress -

- If no process is executing in its critical section and some any other process wish to enter their critical section, then the process should be allowed to enter the critical section.



- Consider the following example:
- There are two processes P_1 & P_2 and no process in its critical section.
- P_1 wish to enter in its critical section & P_2 do not want to enter the critical section but creating some problem/situation (on the basis of code written in its entry section) for process P_1 so that process P_1 could not enter its critical section, it means no progress.

- In other words, P_1 wish to enter but the entry code of P_2 is stopping P_1 to enter its Critical Section or P_2 wish to enter but the entry code of P_1 is stopping P_2 to enter its Critical Section, it means no progress.



3. Bounded Waiting (Secondary Condition):

- If a process is executing in Critical Section and other process is waiting to enter the critical section then the first process should not reenter into critical section again so that other processes waiting may be bounded.

4. No assumption related to the hardware and speed (Secondary Condition, Architectural Neutrality) :

- Suppose solution is provided for synchronization achievement, that solution must not be hardware dependent, that must be hardware independent.
- Solution must not be dependent or work for only one operating system. Solution must be universal.

AKTU PYQs

Q.1. Explain the principle of concurrency. 2014-15, 5 Marks

Q.2. Differentiate concurrent execution and parallel execution. 2015-16, 2 Marks

Q.3. What do you understand by critical section? 2015-16, 2 Marks

Q.4. What is Critical Section? 2016-17, 10 Marks

Q.5. Give the principles, mutual exclusion in critical section problem. Also discus how well these principles are followed in Dekker's solution. 2016-17, 5 Marks

Q.6. State the Critical Section problem. Illustrate the software based solution to the Critical Section problem. 2017-18, 7 Marks

Q.7. What is a Critical Section problem? Give the conditions that a solution to the critical section problem must satisfy. 2018-19, 7 Marks

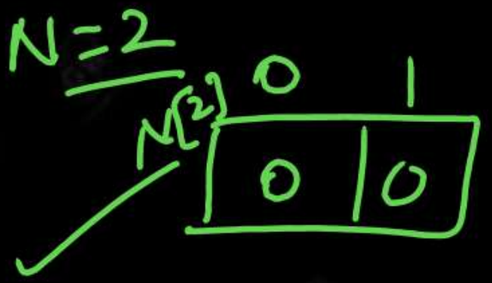
Q.8. Explain in detail about the Mutual Exclusion and Critical Section problem. 2021-22, 10 Marks

Peterson's Solution

- It is a **software based solution** to the critical – section problem. (USER MODE)
- Peterson's solution is restricted to **two processes.**
- It is a **busy – waiting type of solution** i.e. one process will be in critical section and other which is not in critical section will ask for completion of task in critical section.
- It uses two data structure (i) turn (global/shared variable) & (ii) interested (Boolean). { Array of size 2 0, 1 }

Turn = 0 (P_0)
Turn = 1 (P_1)

- The turn variable indicates whose turn is to enter its critical section i.e. if turn = i, then process P_i is allowed to execute in its critical section.
- For example, if turn = 0 means $P_0's$ turn, if turn = 1, means P_1's turn. The interested array is used to indicate if a process is ready/interested to enter its critical section. For example, if interested [i] is true, this value indicates that P_i is ready/interested to enter its critical section.




```
// Peterson' Solution
Non Critical Section
# define N 2
# define TRUE 1
# define FALSE  0
int Interested [ N ] = False ;
int turn;
//Entry_Section
void entry_section (int process)
{
int other ;
other = 1 – process ;
I1 :    Interested [ process ] = True ;
I2 :  turn  = process ;
I3 :  while (interested [ other ] == True & & turn
== Process);
}
I4: Critical Section
//Exit_Section
void exit_section ( int process )
{
I5 :    Interested [ process ] = False;
}
```

```
//Non Critical Section
# define N 2
# define TRUE 1
# define FALSE  0
int Interested [ N ] = False ;
int turn;
//Entry_Section
void entry_section (int process)
{
int other ;
other = 1 – process ;
I1 :  Interested [ process ] = True ;
I2 :  turn  = process ;
I3 :  while (interested [ other ] == True && turn ==
Process);
}
I4: Critical_Section
//Exit_Section
Void exit_section ( int process )
{
I5 :   Interested [ process  ] = False;
}
```

N=2

- Taken Interested [2] /*N=2, array of 2 elements because it is only for 2 processes, it will have only 2 values */

Index : 0 1

Say : P_0 P_1

Interested:

(interested[2]=0)

F	F
0	0

/* Initially false means both donot want to enter the critical section */

```
//Non Critical Section
# define N 2
# define TRUE 1
# define FALSE  0
int Interested [ N ] = False;
int turn;
//Entry_Section
void entry_section (int process)
{
int other ;
other = 1 – process ;
I1 :  Interested [ process ] = True ;
I2 :  turn  = process ;
I3 :  while (interested [ other ] == True && turn ==
Process);
}
I4: Critical_Section
//Exit_Section
Void exit_section ( int process )
{
I5 :   Interested [ process  ] = False;
}
```

- Global/shared variable : turn, it will be used by both the processes P_0 *or* P_1 i.e. synchronization is required.

Turn
- if 0 = P_0's turn
- if 1 = P_1's turn

- Initially its value is 0 means P_0's turn.
- Interest [N] = False ; i.e. initially no any processes is interested to enter critical section.
- Now Entry_section will be executed -

void entry_section (int process)

Process which want to execute the critical section.

- If process P_0 will be interested then process = 0 i.e. P_0
- If process P_1 will be interested then process : = 1 i.e. P_1

```
//Non Critical Section
# define N 2
# define TRUE 1
# define FALSE  0
int Interested [ N ] = False;
int turn;
//Entry_Section
void entry_section (int process)
{
int other ;
other = 1 – process ;
I1 :  Interested [ process ] = True ;
I2 :  turn  = process ;
I3 :  while (interested [ other ] == True && turn ==
Process);
}
I4: Critical_Section
//Exit_Section
Void exit_section ( int process )
{
I5 :   Interested [ process  ] = False;
}
```

- int other; the process which comes for critical section, other then that process is other -
  - If P_0 comes. other is P_1 and if P_1 comes, other is P_0.
- Other : = 1 - process.
  - If process is 0, other is 1 - process = 1 – 0 = 1 (P_1)
  - If process is 1, other is 1 – process = 1 - 1 = 0 (P_0)
- I1: Interested [Process] = true;
  - if P_0 is interested then Interested [0] = True;
  - if P_1 is interested then interested [1] = True;
- I2: Turn = Process ;
  - If Process P_0 is interested, then turn=0 means turn is of Process P_0.
  - If Process P_1 is interested, then turn=1 means turn is of Process P_1.
- Which process will execute this instruction; first- Turn = Process ;
  - Only that process will enter the critical section first.

```
//Non Critical Section
# define N 2
# define TRUE 1
# define FALSE  0
int Interested [ N ] = False;
int turn;
//Entry_Section
void entry_section (int process)
{
int other ;
other = 1 – process ;
I1 :  Interested [ process ] = True ;
I2 :  turn  = process ;
I3:  while (interested [other] == True && turn==
Process);
}
I4: Critical_Section
//Exit_Section
void exit_section ( int process )
{
I5 :   Interested [ process  ] = False;
}
```

- **I3: while (Interested [other] == True && Turn == Process);**
  - if P_0 is process and other is P_1 and that is also interested, mean P_0 and P_1 both want to enter the critical section (which is not possible) && Turn == Process; is also true.
  - Then both conditions are true, mean overall condition is true then it will be executed again and again in while loop but if any condition out of these two conditions is false then overall while condition will be false and one process will enter the critical section.
- **Exit Section:**
  - Interested [Process] = false ;
  - The process which has executed the critical section, will false its Interest and will come out of the critical section.

```
//Non Critical Section
# define N 2
# define TRUE 1
# define FALSE  0
int Interested [ N ] = False;
int turn;
//Entry_Section
void entry_section (int process)
{
int other ;
other = 1 – process ;
I1 :  Interested [ process ] = True ;
I2 :  turn  = process ;
I3 :   while (interested [ other ] == True && turn ==
Process);
}
I4: Critical_Section
//Exit_Section
void exit_section ( int process )
{
I5 :    Interested [ process  ] = False;
}
```

1. Mutual Exclusion
2. Progress
3. Bounded waiting

- Now see how the algorithm is executed.
- We have to check for 3 necessary requirements.
- Ensure Mutual Exclusion – The mutual exclusion can be checked by the following-
  - First, come with the first process say P_0 process in the critical section.
  - Then preempt that process i.e. take CPU (resource) and give to another process & then see another process can be executed in the critical section or not while first process is in critical section.
  - If another process can be executed in the critical section, it means mutual exclusion not achieved otherwise mutual extension is achieved.

```
//Non Critical Section
# define N 2
# define TRUE 1
# define FALSE  0
int Interested [ N ] = False ;
int turn;
//Entry_Section
void entry_section (int process)
{
int other ;
other = 1 – process ;
I1 :  Interested [ process ] = True ;
I2 :  turn  = process ;
I3 :  while (interested [ other ] == True && turn ==
Process);
}
I4: Critical_Section
//Exit_Section
Void exit_section ( int process )
{
I5 :    Interested [ process  ] = False;
}
```

- Say P_0 comes, it will execute instruction number 1 i.e. interested [process] = True; i.e. interested [P_0] = 1; i.e. P_0 is interested for the critical section.
- Then, it will execute the instruction 2 i.e. turn = process; i.e. turn = 0;
- Now instruction 3 will be executed:
  - while (interested [other] == True && turn == process) ;
  - i.e. interested [1] == true && 0 == 0 i.e. false && true i.e. overall condition is false.
  - i.e. it will execute instruction 4 i.e. process P_0 will enter the critical section.

```
//Non Critical Section
# define N 2
# define TRUE 1
# define FALSE  0
int Interested [ N ] = False ;
int turn;
//Entry_Section
void entry_section (int process)
{
int other ;
other = 1 – process;
I1: Interested [process] = True ;
I2: turn  = process;
I3 :  while  (interested  [other]==True  &&  turn==
Process);
}
I4: Critical_Section
//Exit_Section
void exit_section ( int process )
{
I5 :Interested [process] = False;
}
```

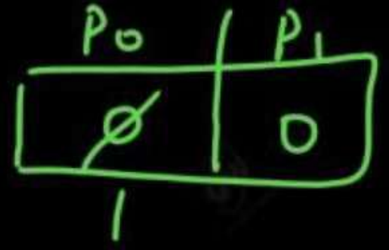
- Now Preempt process P_0 and CPU will be given to P_1.
- Now P_1 will starts its execution form instruction 1 i.e.
- Interested [process] = true ; i.e. interested [1] = true; i.e. P_1 is now interested to enter the critical section.
- Now instruction 2 will be executed. i.e. Turn = process i.e. Turn = 1.
- Now instruction 3 will be executed-
  - while (interested [other] == true && turn ==process);
  - i.e. interested [0]==1 && Turn == 1; i.e. true && true = True

```
//Non Critical Section
# define N 2
# define TRUE 1
# define FALSE  0
int Interested [ N ] = False ;
int turn;
//Entry_Section
void entry_section (int process)
{
int other ;
other = 1 – process ;
I1 :  Interested [ process ] = True ;
I2 :  turn  = process ;
I3 :  while (interested [ other ] == True && turn ==
Process);
}
I4: Critical_Section
//Exit_Section
Void exit_section ( int process )
{
I5 :   Interested [ process  ] = False;
}
```

- Both conditions are true i.e. Overall while condition is true.
- i.e. it will execute this loop again and again i.e. busy waiting loop and will not execute instruction - 4 i.e. will not enter the critical section.

❑ i.e. Mutual exclusion is ensured.

```
//Non Critical Section
# define N 2
# define TRUE 1
# define FALSE  0
int Interested [ N ] = False ;
int turn;
//Entry_Section
void entry_section (int process)
{
int other ;
other = 1 – process ;
I1 :  Interested [ process ] = True ;
I2 :  turn  = process ;
I3 :  while (interested [ other ] == True && turn ==
Process);
}
I4: Critical_Section
//Exit_Section
Void exit_section ( int process )
{
I5 :    Interested [ process  ] = False;
}
```

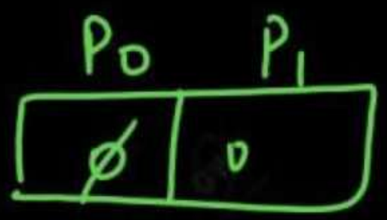


- Now check for **progress.** Progress means a process which want to enter the critical section is not being stopped by another process which do not want to enter the critical section.
- When P_0 comes
- It will execute instruction–1 and will show the interest by setting, interested [0] = 1.
- Now it will executed instruction–2 setting turn = 0
- Now it will execute instruction - 3 i.e.
  - while (interested [other] == True) that is checking that other process is interested or not, that is false && (Turn == Process) i.e. True and false and true is false.

```
//Non Critical Section
# define N 2
# define TRUE 1
# define FALSE  0
int Interested [ N ] = False ;
int turn;
//Entry_Section
void entry_section (int process)
{
int other ;
other = 1 – process ;
I1 :  Interested [ process ] = True ;
I2 :  turn  = process ;
I3 :  while (interested [ other ] == True && turn ==
Process);
}
I4: Critical_Section
//Exit_Section
Void exit_section ( int process )
{
I5 :    Interested [ process  ] = False;
}
```

- i.e. Is Process P_1 (other process) is stopping Process P_0 to enter the critical section?– No because false && True = False (NO BUSY WAIT LOOP)
- i.e. P_0 can enter the critical section.
- i.e. not interested process is not stopping the interested process to enter the critical section.
- i.e. Progress is ensured.

```
//Non Critical Section
# define N 2
# define TRUE 1
# define FALSE  0
int Interested [ N ] = False ;
int turn;
//Entry_Section
void entry_section (int process)
{
int other ;
other = 1 – process ;
I1 :  Interested [ process ] = True ;
I2 :  turn  = process ;
I3 :  while (interested [ other ] == True && turn ==
Process);
}
I4: Critical_Section
//Exit_Section
Void exit_section ( int process )
{
I5 :    Interested [ process  ] = False;
}
```



- Now check for "Bounded waiting"
- We know which process will execute the instruction–2 first, will be executed first i.e. which is coming first will be executed first.
- Say P_0 comes.
- Instruction 1: interested [P_0] = True = 1.
- Instruction 2: Turn = Process i.e. Turn = 0.
- Instruction 3: Check for P_1 is interested or not through while condition. [F && T = False]
  - P_1 is not interested.
- P_0 will enter the critical section. If process P_0 wants again chance of critical section?

```
//Non Critical Section
# define N 2
# define TRUE 1
# define FALSE  0
int Interested [ N ] = False ;
int turn;
//Entry_Section
void entry_section (int process)
{
int other ;
other = 1 – process ;
I1 :  Interested [ process ] = True ;
I2 :  turn  = process ;
I3 :   while (interested [ other ] == True && turn ==
Process);
}
I4: Critical_Section
//Exit_Section
Void exit_section ( int process )
{
I5 :    Interested [ process  ] = False;
}
```

- When P_0 in the critical section.
- Preemption i.e. CPU will be taken and will be given to another process P_1.
- Now P_1 will execute first instruction i.e. interested [process] = True, interested [P_1] = 1. i.e. now P_1 is interested.
- Now it will execute Instruction 2 & will set Turn = 1.
- *Now Instruction 3 will be executed*:

  while (interested [other i.e. P0] == True && $Turn == Process$)

  True &&

  True = True (i.e. busy wait loop)

```
//Non Critical Section
# define N 2
# define TRUE 1
# define FALSE  0
int Interested [ N ] = False ;
int turn;
//Entry_Section
void entry_section (int process)
{
int other ;
other = 1 – process ;
I1 :  Interested [ process ] = True ;
I2 :  turn  = process ;
I3 :  while (interested [ other ] == True && turn ==
Process);
}
I4: Critical_Section
//Exit_Section
Void exit_section ( int process )
{
I5 :   Interested [ process  ] = False;
}
```

- P_1 can not enter the critical section.
- P_0 is already in critical section.
- Now P_0 will execute again and will come out of critical section.
- Now P_0 will execute Instruction 5 (exit section)
- Instruction 1: interested [$Process$] = True

  i.e. interested [0] = 1. Now I2 will be executed
  - will set Turn = 0,
  - I3:- interested (other-P_1) == True && turn Process
    True — Turn = 0, True
    - i.e. busy wait loop.
    - i.e. P_0 can not enter the cs again
  - Now P1 will come, it is on 3rd instruction
  - I3 will be executed. (it will be false)

```
//Non Critical Section
# define N 2
# define TRUE 1
# define FALSE  0
int Interested [ N ] = False ;
int turn;
//Entry_Section
void entry_section (int process)
{
int other ;
other = 1 – process ;
I1 :  Interested [ process ] = True ;
I2 :  turn  = process ;
I3 :  while (interested [ other ] == True && turn ==
Process);
}
I4: Critical_Section
//Exit_Section
Void exit_section ( int process )
{
I5 :    Interested [ process  ] = False;
}
```

- i.e. Process P_1 can enter the critical section.
- It means once P_0 has entered the critical section & due to preemption P_1 (other Process) has set its turn, then P_0 after executing the critical section, again can not enter the critical section.
- i.e. **Bounded waiting is achieved/ensured.**

AKTU PYQs

Q.1. Describe Peterson solution. 2012-13, 5 marks

Solution of Critical Section Problem using lock variable

- This is a software mechanism implemented in User mode.
- This is a busy waiting solution which can be used for more than two processes.
- In this mechanism, a variable lock is used.
- There are two possible values of lock - 0 or 1.
- Lock value 0 means that the critical section is vacant while the lock value 1 means that it is occupied.
- A process which wants to get into the critical section first check the value of the lock variable.
- If it is 0 then it sets the value of lock as 1 and enter to the critical section, otherwise it waits.
- Race conditions are prevented by requiring that critical section be protected by locks.
- That is, a process must acquire a lock before entering a critical section; it releases the lock when it exits the critical section.
- It is a solution for multi-processes.

- If we look at the Pseudo Code, we find that there are three sections in the code. Entry Section, Critical Section and the exit section.
- Critical section solution using lock variable:-

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

```
do {
    ES  ACQUIRE LOCK
        CRITICAL SECTION
    Exit Sec  RELEASE LOCK
        REMAINDER SECTION
} while (TRUE);
```

- The pseudo code of the mechanism looks like following:

```
I1: while (lock==1);
I2: lock =1;
I3: Critical Section;
I4: lock=0;
```

FIG - SOLUTION TO THE CRITICAL SECTION PROBLEM USING LOCK

```
I1: while (lock == 1);
I2: lock = 1;
I3: Critical Section;
I4: lock=0;
```

ENTRY CODE

CS

EXIT CODE

- Say two processes are P_1 & P_2 (solution of multi-processes).
- Initially lock = 0 (say)
- lock = 0 shows Critical Section is vacant that is no process is in its critical section.
- Say P_1 comes and run the entry code, execute

  I1: i.e. while (lock == 1); i.e. while (0 == 1);

  i.e. condition is false and it will come out of while loop.

- Now it will execute I2: i.e. Set Lock = 1 and then will execute I3 and P_1 will enter the Critical Section.
- Now let suppose P_2 also want to enter the critical section
- The instruction –1 will be executed i.e. while (lock == 1); i.e. while (1 == 1); i.e. condition is true i.e. infinite loop. i.e. P_2 can not enter the Critical Section.
- When P1 will complete its work then it will come out of Critical Section and will execute instruction 4 i.e. set lock = 0.

```
I1: while (lock==1);
I2: lock =1;
I3: Critical Section;
I4: lock=0;
```

- Now suppose P_2 says, I wish to execute the Critical Section:

  i.e. while (lock == 1);

  i.e. while (0 == 1);

  i.e. condition is false, it will come out of while loop & then will execute I2 and set lock = 1 & will enter the Critical Section.

- In this way processes can be executed using lock and release tools.

```
I1: while (lock==1);
I2: lock =1;
I3: Critical Section;
I4: lock=0;
```

- Now check for **Mutual Exclusion** requirement.
- There are two processes say P_1 & P_2 & Lock = 0 (i.e. no one is in Critical Section)
- Say P_1 comes and execute $I1$:

  i.e. while (lock == 1);

  i.e. while (0 == 1) **False**

  i.e. Out of while loop.
- Now suppose after executing $I1$, process P_1 is **Preempted.**
- Now P_2 comes and P_2 execute I1 i.e.

  while (lock == 1); i.e. while (0 == 1) **False**
- Now P_2 will execute $I2$ i.e. set lock = 1 & now P_2 enters the critical section.
- Now say P_1 comes (Back)
- Now it will not be restarted because it has already executed $I1$ before preemption.
- Now P_1 will set **lock = 1** and will enter the critical section.
- **It means two processes P_1 & P_2 are in critical section.**
- i.e. do not satisfy the Mutual Exclusion condition.
- Since this method failed at the basic step; hence, there is no need to talk about the other conditions to be fulfilled.

Synchronization using Hardware : Test_and_Set instruction

- **Limitation of lock variable:**
- We have seen that in lock variable say there are two processes P_1 and P_2 and initially the value of lock variable is 0 i.e. no process is in the Critical Section.
- Say process P_1 comes first.
- First the instruction I_1 i.e. while (lock == 1); will be executed and this condition will be false as initially lock = 0 i.e. it will come out of while loop.
- Say after execution of I_1 there an interrupt is encountered.
- Now P_2 will execute the I_1 i.e. while (lock == 1); will be executed.
- As lock is still 0, this condition will become false and it will come out of while condition.
- Then I_2 will be executed i.e. lock will be set by value 1. Now P_2 will enter the critical section.

- Now let suppose P_1 comes again. As P_1 has already executed I_1 , directly it will execute I_2 & will enter the critical section.
- In this way more than one processes will enter the critical section. This has happened because of an interrupt occurred between I_1 & I_2.
- In test and set instruction, the instruction I_1 & I_2 of lock tool has been combined in one i.e. made both the instructions atomic by the following condition:

```
while(test_and_set(& lock));
```

- This instruction is equivalent to - I_1 : while (lock = = 1); & I_2 : lock = 1 ;

❑ Compare both the functions:-

Lock Variable

```
I1 : while (lock = = 1);
I2 : lock = 1 ;
I3 : Critical Section ;
I4 : lock = 0 ;
```

Test_and_Set Instruction

```
I1 : while (Test_and_set
(& lock));
I2 : critical Section ;
I3 : lock = 0 ;
```

Call by ref^n

```
/* function Definition of test_and_set */
Test_and_set ( boolean * target)
{
boolean r =* target;
*target = true;
return r;
}
```

TEST & SET INSTRUCTION :-

```
I1 : while (Test_and_Set (& lock));
I2 : Critical Section ;
I3 : lock = 0 ;
/* function Definition of test_and_set */
Test_and_Set ( boolean * target)
{
boolean r =* target;
*target = true;
return r;
}
```

P1

- Initially lock = 0 ;
- Say process P_1 comes.
- Now while condition will be executed (I1)

  i.e. while (Test_and _Set (& lock)) ;
- This instruction will call function Test_and _Set ( ) in call by reference manner as the address of lock variable is being passed to the function definition i.e. say

int a = 5 ;

a | 5 | 1000

logical name ← LOCK

1	∅	∅	1

actual name / Ref / Add 1000 (say)

```
I1 : while (Test_and_Set (& lock));
I2 : Critical Section ;
I3 : lock = 0 ;
/*function Definition of test_and_set */
Test_and_set ( boolean *target)
{
boolean r =* target;
*target = true;
return r;
}
```

- The address of the lock say 1000 will be passed and will be stored in pointer variable target i.e.

Target (1004)

- i.e. target will point to lock.
- Now boolean r = * target; will be executed i.e.

r = *(1000)

r = 0

2000

*target
*(&lock)
*(1000)

TEST

- Now *target = true; i.e. * (1000) = 1;
- i.e.

*(1000) = 1

Lock 1000 → 1

SET

```
I1 : while (Test_and_Set (& lock));
I2 : Critical Section ;
I3 : lock = 0 ;
/*function Definition of test_and_set */
Test_and_set ( boolean *target)
{
boolean r =* target;
*target = true;
return r;
}
```

- Then return r i.e. will return 0.
- i.e. while condition is false.
- Then P_1 will enter the critical section.
- Let suppose P_2 comes.
- Then while condition will be checked i.e. will call Test_and_Set( ) and will pass the address of lock to the function definition. i.e.

Target | 1000

- Then in function definition –

boolean r = * target

i.e. r = * (1000)

i.e. r = 1 → r | 1

```
I1 : while (Test_and_Set (& lock));
I2 : Critical Section ;
I3 : lock = 0 ;
/*function Definition of test_and_set */
Test_and_set ( boolean *target)
{
boolean r =* target;
*target = true;
return r;
}
```

1

* target = true

i.e. * (1000) = 1

i.e.

1

lock

- It will return 1.
- Now while condition will execute again and again & P_2 can not enter the critical section.
- i.e. Mutual Exclusion is achieved.

AKTU PYQs

Q.1. Discuss Mutual-exclusion implementation with test and set ( ) instruction.

2017-18, 7 Marks

Q.2. Give the solution to Critical Section Problem using Hardware Instructions.

2017-18, 2 Marks

Solution of Critical Section Problem using Turn variable (Strict Alteration)

- We have already discussed – Synchronization using Lock variable and Synchronization using test_and_set () method.
- This is also a two process solution i.e. works for only two processes.
- It works in the user mode without support of kernel.
- Let see its pseudo code-

	Process P0	Process P1
Entry Code	While (turn ! = 0);	While (turn != 1);
	Critical Section	Critical Section
Exit Code	Turn = 1;	Turn = 0;

- Let suppose the initial value of turn is 0 i.e.

  turn = 0.

{ turn = 1 may also be taken, if turn = 0 , i.e., process P0 will run first, if turn = 1, i.e. process P1 will run first}

- So let suppose turn = 0.
- Now see the pseudo code of Process P0 -

```
Process P0
While ( turn != 0 );
Critical Section
Turn = 1;
```

0 != 0 F

P0

- The first instruction, while (turn ! = 0); will be executed.
- i.e. while (0 ! = 0); i.e. false and process P0 will enter the Critical Section.
- P0 is already in Critical Section, check if P1 can enter the Critical Section.
- Now see the pseudo code of Process P1.

```
Process P1
While ( turn != 1 );
CS
Turn = 0;
```

Es (0 != 1) T

Busy wait

- Now see its Entry Section code -
- i.e. while (turn ! = 1);
- i.e. while (0 ! = 1) ; i.e. the condition is true and it will execute this while loop again and again and will not be able to enter the Critical Section.
- Mutual Exclusion is achieved.

- Let suppose the initial value of turn is 1 i.e.

  turn = 1.

- if turn = 1, i.e. process P1 will run first.
- Now see the pseudo code of Process P1-

```
Process P1
While ( turn != 1 );
Critical Section
Turn = 0;
```

- The first instruction, while (turn ! = 1); will be executed.
- i.e. while (1 ! = 1); i.e. the condition is false and process P1 will enter the Critical Section.
- P1 is already in Critical Section, check if P0 can enter the Critical Section.
- Now see the pseudo code of Process P0.

```
Process P0
While ( turn != 0 );
CS
Turn = 1;
```

- Now see its Entry Section Code -
- i.e. while (turn ! = 0);
- i.e. while (1 ! = 0) ; i.e. the condition is true and it will execute this while loop again and again and will not be able to enter the Critical Section.
- i.e. Mutual Exclusion is achieved.

- **Now check for the Progress condition:-**
- Progress says if no process is in the Critical Section i.e. Critical Section is empty and any process say P0 wish to enter the Critical Section and if any other process say process P1 that do not wish to enter the Critical Section, should not stop process P0 to enter the Critical Section that wish to enter the Critical Section. If it stops it, i.e. no progress is achieved.
- Lets check it-
- Let suppose initially turn = 0 and Critical Section is empty, suppose process P1 wish to enter the Critical Section.
- i.e. it will execute its Entry Section –
- while (turn ! = 1);
- i.e. while (1 ! = 0); i.e. condition is true and it will execute this while loop again and again and will not be able to enter the Critical Section even the Critical Section is empty.
- Let suppose initially turn = 1 and Critical Section is empty, suppose process P0 wish to enter the Critical Section i.e. it will execute its Entry Section Code –

Busy wait loop

- while (turn ! = 0);
- i.e. while (1 ! = 0); i.e. condition is true and it will execute this while loop again and again and will not be able to enter the Critical Section even the Critical Section is empty.
- i.e. No process is able to enter the Critical Section even it is empty i.e. Progress is not achieved i.e. No Progress.
- In this solution of Critical Section Problem, first Process P0 will enter the Critical Section and after executing in the Critical Section, it will set the turn value by 1. Then Process P1 will be able to enter the Critical Section. i.e. named as Strict Alteration.

- **Now check for Bounded Waiting-**
- Let suppose turn = 0 and process P0 wish to enter the Critical Section.
- First P0 will execute its Entry Section code i.e. while (turn ! = 0); i.e. while (0 ! = 0); i.e. condition is false and P0 will enter the Critical Section.
- After executing the Critical Section, the process P0 will set turn = 1; i.e. now process P1 can enter the Critical Section.
- **Check if P0 again can enter the Critical Section-**
- P0 first will execute the Entry Section code i.e. while (turn != 0); i.e. while (1 != 0); i.e. condition is true and it will execute the while loop again and again and will not be able to enter the Critical Section.
- Let suppose turn = 1 and Process P1 wish to enter the Critical Section.
- P1 will execute the Entry Section Code i.e. while (turn != 1); i.e. while (1 ! = 1); i.e. condition is false and Process P1 can enter the critical section. After executing the Critical Section it will set turn = 0.

- Check if P1 again can enter the Critical Section-
- P1 first will execute the Entry Section code i.e. while (turn != 1); i.e. while (0 != 1); i.e. condition is true and it will execute the while loop again and again and Process P1 will not be able to enter the critical section again.
- i.e. Bounded Waiting is achieved.

- The fourth condition is – the solution should be hardware independent.
- This solution is hardware independent and can be executed on any platform.

Synchronization Tool - Semaphore

- There are two types of solutions for synchronization – software solution and hardware solution.
- Another way is to provide the tool for synchronization to the programmer for the solution of critical section problem and to achieve the synchronization in concurrent environment.
- Semaphore is such a synchronization tool provided to programmer to write code for solution of critical section problem.
- It is an integer value which can not be accessed directly, its value can only be accessed by two functions –
  - wait () or P() function — down ( )
  - signal () or V() function — up ( )
- These both functions are atomic functions.

* If the value of S is greater than 0, the wait() function decrease the value of S by 1. The definition of wait() operation is as follows:

```
wait(S)
{ while(S<=0); /* Busy wait loop */
    S=S-1;
}
```

* The signal() function increase the value of S by 1.

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

- **Purpose of wait ( ) :-**
- This S variable is going to be shared between processes.
- So when one process let say want to access a resource and want to enter the critical section that time this variable S that is sheared between the processes will take care of who should access the resource or who should enter the critical section.
- So if the value of S is less than or equal to 0 that means that some process is already making use of the shared resource or is already there is the critical section. So at that time no other process should be allowed to enter the critical section or it should not be allowed of use the resource that is being shared.
- So that is why when the process wants to access the shared resource or want to enter the critical section it will check this condition and while checking this condition if this condition is true then it will be stuck here and will not be allowed to enter the critical section or it will not allowed to use the shared resources.

- If S is not less or equal to 0 it will enter the critical section and will decrement the value of S by 1 to inform other processes that S has been decremented by one and on the basis of updated value of S other process can enter or not enter the critical section.
- This condition is basically testing for availability of resource or process can enter the critical section or not.

signal()

- **Purpose of V operation :-**
- It increments the value of S.
- This signal operation will be called when the process that was making use of the S to enter critical section or wants using a resource completes its operation and comes out of it.

release

- Hence denoting that it has relend the semaphore.
- All the modifications to the integer value of the semaphore in the wait ( ) and signal ( ) must be executed indivisibly.
- That is, when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value.

- **Binary Semaphore :-** (0, 1)
- Binary semaphore are also knowns mutex locks as they are locks that provide mutual exclusion.
- If S value is 0 that means that shared resource is being used by some other process and the currently requesting process has to wait
- In terms of critical section, if S = 0 that means that some process is already executing in critical section and the requesting process has to wait and if the value of S is 1 means that is the shared resource is free and can be used by ~~curetting~~ currently requesting process.

Critical Section Problem Solution With Semaphore

Mutual Exclusion is achieved

- Suppose there are two processes P0 and P1.
- Suppose P0 wish to enter the critical section. First it will execute the wait(s).
- Wait(s) means, it will check the while condition, i.e. while (s < = 0); , no it is 1 i.e. while condition is false and it will decrease the value of s by 1 i.e. now s = 0 and process P0 will enter the critical section.
- Suppose now P1 wish to enter the critical section. Now it will execute wait (s) i.e. it will decease the value of s by 1.

- The value of s is already 0, i.e. it will stuck in while loop and will not come out of the while loop i.e. can not enter the critical section.
- P1 will be able to enter the critical section when process P0 will complete its execution in Critical Section and will execute its signal(), i.e. signal() will increase the value of semaphore by 1.
- The value of s will indicate that how many processes can enter the critical section. If the value of counting semaphore is 2 , i.e. 2 processes can enter the critical section at a time. Situation where we want that more that one process should enter the critical section, counting semaphore is used.
- How many processes can enter the critical section , it will depend on the value of the counting semaphore.

Question - Consider a semaphore S, initialized with value 10. what should be the value of S after executing 6 times P() and 8 times V() function on S? ***(GATE Question)***

Answer – Given S = 10, i.e. counting Semaphore , 10 – 6 + 8 = 12

Counting Semaphore

- Its value can range over an unrestricted domain i.e. it can have multiple values. (it can be 1, 2, 3.....}.
- It is used to control access to resources that has multiple instances.
- Let suppose there are 3 processes- P1, P2, and P3.
- Let say the resource is having multiple instances R1 and R2 that means, it is not just one recourse, but the same recourse is having multiple instances so multiple processes can make use of it.
- So P1 can use resource R1 and P2 can use resource R2. To make synchronized use of resources counting semaphore is used.
- The value of counting semaphore is 2 i.e. S = 2;
- First Process P1 comes and execute wait()-

wait (s)

{

while (s <=0);

i.e. while (2<=0); condition is false.

- i.e. it will decrement S by 1 and S will become 1 and process P1 can use the recourse 1 or can enter the critical section.
- Let suppose process P2 arrives and want to use the second resource instance R2.
- It will again execute the wait().

i.e. wait (s)

{

while (S < = 0); i.e. while (2 < = 0); i.e. condition is false and it will come out of the while loop.

- Now it will decrement S by 1 , now S = 0; and use the resource type R2 or enter the Critical Section.
- Now both instances R1 and R2 are used by process P1 and P2.
- Now let suppose another process P3 arrives.
- It will again execute wait().

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

- i.e. while (0 < = 0); i.e. condition is true and it will struck in the while loop and will not be able to use any resource or will not be able to enter the Critical Section.
- Now let suppose any process P1 or P2 complete its Critical Section execution and execute signal (S)-

i.e. signal (S) { S ++; i.e. S = 0 + 1 = 1, which shows that resource has been released and available for other process to use.

- Now P3 which was stuck in the while condition , now condition will become false as S = 1 and it will execute S -- i.e. S = 1 – 1 = 0 and can use the recourse or can enter the critical section.

Disadvantages of Semaphore

- It requires " Busy Waiting".
- What is Busy Waiting? – While a process is in Critical Section, any other process that tries to enter the Critical Section must loop continuously in the entry code.
- Busy waiting wastes CPU cycles that some other process might be able to use productively.
- This type of semaphore is also called a Spinlock because the process "spins" while waiting for the lock.

Solution –

- To overcome the need for busy waiting, we can modify the definition of the wait and signal semaphore operations.
- When a process executes the wait() operation and finds that the semaphore value is not positive, it must wait.

(i) However, rather then engaging in the busy waiting, the process can block itself.

GW GATEWAY CLASSES

(ii) The block operation places a process into a waiting queue associated with the semaphore, and the state of the process is switched to the waiting state.

- Then control is transferred to the CPU scheduler, which selects another process to execute.
- Hence we are not going to waste our precious CPU time by letting the processor / CPU busy in busy wait loop.
- By applying this scenario we can observe deadlocks and starvation.
- This implementation of a semaphore with a waiting queue may result in a situation where two or more processes are waiting indefinitely for an event that can be caused only one of the waiting process. For example-

P0	P1
Wait (S); Wait (Q); . . Signal(S); Signal(Q);	Wait (Q); Wait (S); . . Signal(Q); Signal(S);

S Q

- The processes P0 and P1 are going to use semaphore S and Q.
- Process P0 performs wait(S) and wait (Q) that means it want to use S and Q and no other process can use S and Q.
- Process P0 then perform signal(S) and signal (Q) that means release S and Q so that other process can use S and Q.
- Similarly for process P1.
- When P0 perform wait on S at the same time P1 perform wait on Q, no problem.
- Now P0 can not perform wait on Q and P1 can not perform wait operation on S. This is a deadlocked condition. It is similar with the signal() operation.

AKTU PYQs

Q.1. Define Busy Waiting ? How to overcome busy waiting using Semaphore operations.

2016-17, 2 Marks

Q.2. Define Binary Semaphores and Mutex. 2017-18, 2 Marks

Q.3. Explain what semaphores are, their usage, implementation given to avoid busy waiting and binary semaphores.

2018-19, 7 Marks

Classic Problems of Synchronization

Bounded – Buffer Problem or Producer – Consumer Problem

- The bounded buffer problem, also known as the producer-consumer problem, is a classic synchronization problem in operating systems.
- It involves multiple processes sharing a fixed-size buffer of n slots where one process, the producer, generates data and puts it into the buffer, and another process, the consumer, retrieves data from the buffer.

❑ The code for Producer process is as follow: -

```
/*Producer*/
int count = 0; // shared variable
void producer (void)
{
int itemp ;
while ( true)
{
produce_item ( itemp );
while ( count == n ); //overflow
buffer [ in ] = itemp ;
in = ( in + 1) mod n;
count = count + 1; /* shared variable,
                      how many items are in buffer*/
}
```

$R_p \leftarrow m(count)$
$R_p = R_p + 1$
$m[count] = R_p$

Register

```
load Rp, m[count];
incr Rp;
store m[count], Rp;
```

Size of Buffer

Say $n = 8$

Buffer [0 ... n-1]

0	
1	
2	
3	
4	
5	
6	
7	

Shared buffer: Shared by both the processes

COUNT

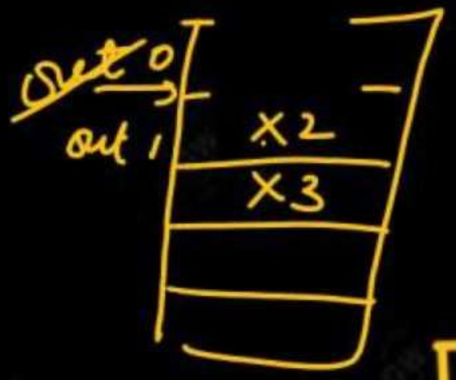
Shared Variable: Shared by both processes

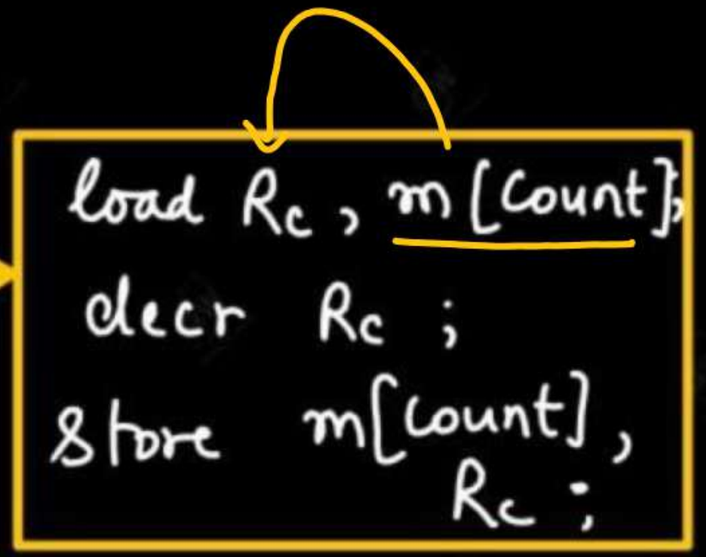
in 0 Say

out 0 Say

❑ The code for Consumer process is as follow: -

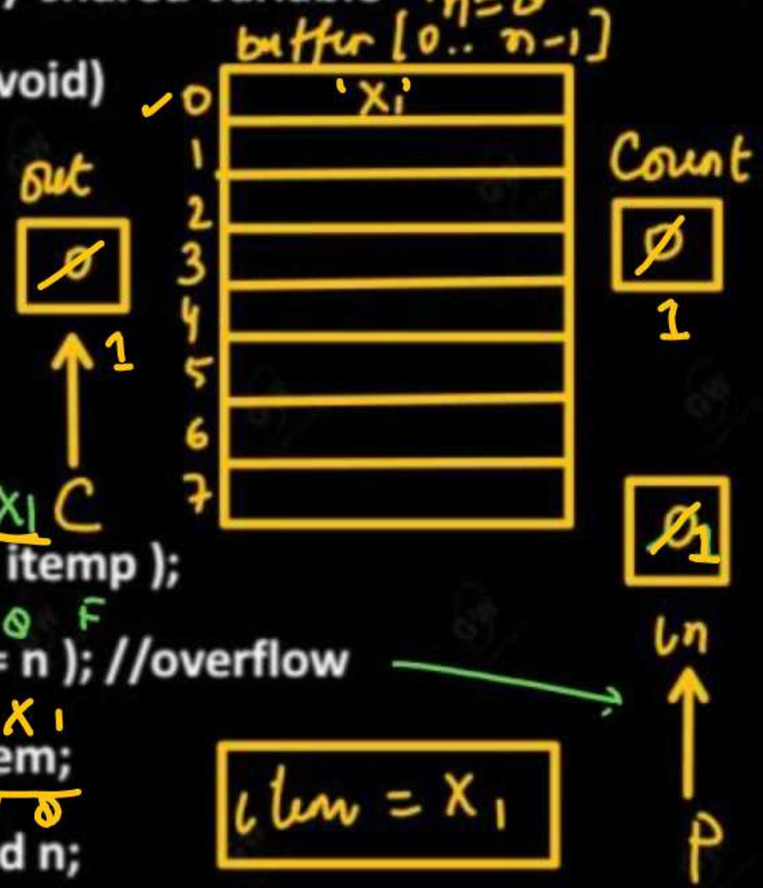
```
/*Consumer*/
void consumer(void)
{
int itemc;
while (true)
{
while ( count == 0 ); // underflow or empty
itemc = buffer ( out );
out = ( out + 1 ) mod n;
count = count - 1; /* shared variable,
          how many items are in buffer*/
Process_item ( itemc );
}
```





- If an item is produced by the Producer, then that item will be placed in the Buffer and the count will be incremented by 1 which says that how many items are in the Buffer.
- If an item is consumed by the Consumer, then item will be processed and the value of count will be decreased by 1.
- If count is equal to n, the maximum size of Buffer, item can not be produced by the Producer since no space is in Buffer.
- If count is equal to 0, item cannot be consumed by the Consumer since no item is in the buffer.

```
/*Producer*/
int count = 0; // shared variable
void producer (void)
{
int itemp ;
while ( true)
{
produce_item ( itemp );
while ( count == n ); //overflow
buffer [ in ] = item;
in = ( in + 1) mod n;
count = count + 1; /* shared variable,
how many items are in buffer*/
}
```



- **Case 1:** Producer will produce the item and Consumer will consume the item from the Buffer.
- Let suppose Producer comes and produce one item say X1.
- int count =0 and buffer both are global variables.
- out = 0 (say) means no item is consumed by the Consumer. This is used by the Consumer.
- In is used by the Producer.
- while (count == n); i.e. if count is equal to the size of buffer that means buffer is full and Producer can not produce the item i.e. buffer overflow, the Producer will stuck in the infinite loop, here (0 == 8) False

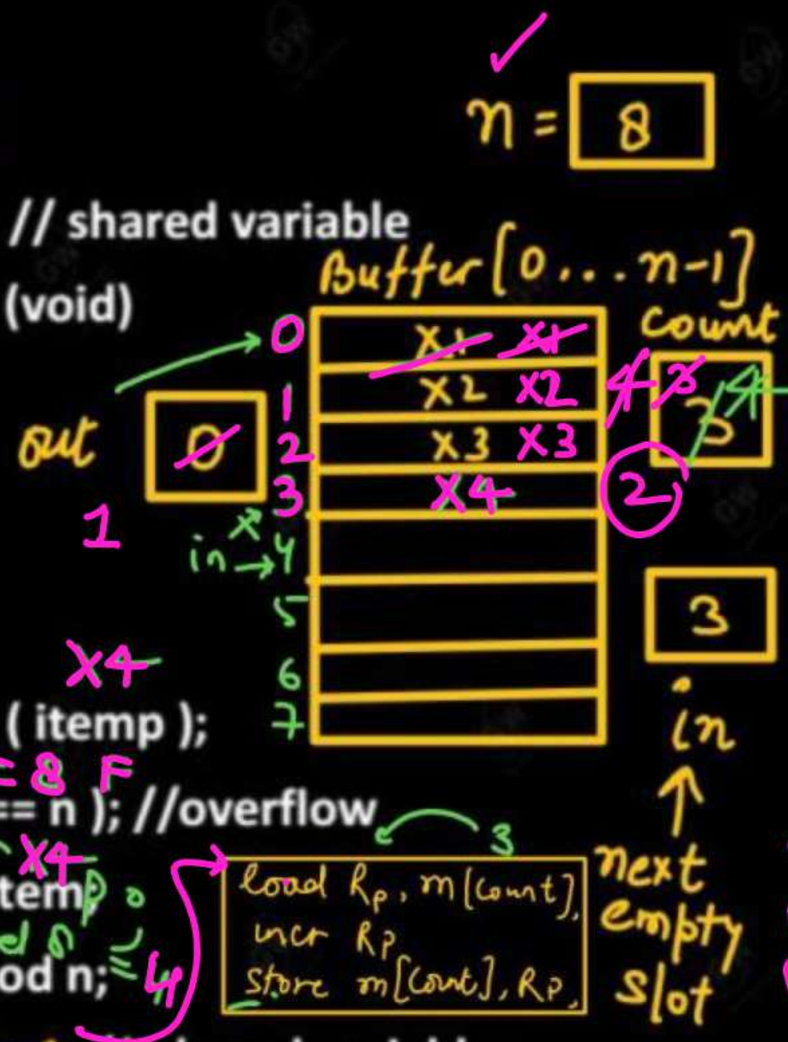
```
/*Producer*/
int count  = 0;  // shared variable
void producer (void)
{
int itemp ;
while ( true)
{
produce_item ( itemp );
while ( count == n ); //overflow
buffer [ in ] = itemp ;
in = ( in + 1) mod n;
count = count + 1; /* shared variable,
          how many items are in buffer*/
}
```

- Buffer [in] = itemp; i.e. Buffer [0] = itemp; Let item is X1. i.e. Buffer [0] = X1.
- in = (in + 1) mod n = (0 + 1) mod 8 = 1 (i.e. the next empty slot in buffer)
- Now count = count + 1 = 0 + 1 = 0; //count will be incremented by 1. How CPU will execute this instruction. CPU execute this instruction using microinstructions-
- $load\ R_p, m\ [count]\ i.e.\ R_p \leftarrow 0$
- $incr\ R_p \quad i.e.\ R_p \leftarrow R_p + 1 = 1$
- Store m[count], R_p i.e. m[count] ← 0.
- In this way Producer has produced an item.
- Now see how the Consumer consumes the item from the buffer.

```
/*Consumer*/
void consumer(void)
{
int itemc;
while (true)
{
while ( count == 0 ); // underflow or empty
itec = buffer [ out ];
out = ( out + 1 ) mod n;
count = count - 1; /* shared variable, how many
items are in buffer*/
Process_item(itemc);
}
```

- Now Consumer will consume the item from the same buffer.
- Let suppose Consumer comes.
- while (count == 0); i.e. if count is equal to 0 that means buffer is empty and Consumer can not consume the item i.e. buffer is underflow, the Consumer will stuck in the infinite loop, here (1 == 0) False i.e. Consumer can consume the item.
- Now itemc = buffer [out] i.e. itemc = buffer [0] = X1.
- Now out = (out + 1) mod n = (0 + 1) mod 8 = 1 mod 8 = 1. i.e. location of next item to consume.
- count = count - 1 = 1 – 1 = 0, How CPU will execute this instruction? CPU execute this instruction using microinstructions-
- load $R_c, m\ [count]$ $i.e.\ R_c = 1$
- Decr R_c i.e. $R_c = 0$
- Store m[count], R_c i.e. m [count] = 0.
- i.e. no item is in the buffer i.e. process the item.

```
/*Producer*/
int count  = 0;  // shared variable
void producer (void)
{
int itemp ;
while ( true)
{
produce_item ( itemp );
while ( count == n ); //overflow
buffer [ in ] = itemp;
in = ( in + 1) mod n;
count = count + 1; /* shared variable,
                      how many items are in buffer*/
}
```



- **Case 2: -**
- Let suppose Producer has produced three items say X1, X2, X3. count will b 3. i.e. in = 3 (the next empty slot in buffer)
- out = 0 (say) means no item is consumed by the Consumer.
- Say Producer is ready to produce one more item. (already produced 3 items). → X4
- while (count == n); i.e. (3 ==0) False
- Buffer [in] = itemp; i.e. Buffer [3] = itemp; Let item is X4. i.e. Buffer [3] = X4.
- in will become 4. (the next empty slot in buffer)
- count = count + 1; //count will be incremented by 1. How it will be incremented-
  - $load\ R_p, m\ [count]\ i.e.\ R_p \leftarrow 3$
  - $incr\ R_p\ \ i.e.\ R_p = R_p + 1 = 4$
- Let suppose now Producer is preempted and store m[count], R_p could not be executed.
- Now due to preemption of Producer process CPU will see for another process for execution.

```
/*Consumer*/
void consumer(void)
{
int itemc;
while (true)
{
while (count == 0 ); // underflow or empty
itemc = buffer [ out ];
out = ( out + 1 ) mod n;
count = count - 1; /* shared variable, how many
items are in buffer*/
Process_item ( itemc);
}
```

3==0 F

0+1 mod 8 = 1

load R_c, m[count];
decr R_c; 2
store m[count], R_c

- Now Consumer process will be executed.
- In consumer process:
- $while\ (count == 0); i.e.\ (3 == 0) false$
- $itemc = Buffer\ (out);\quad out\ is\ 0.\ (say)$
- $itemc = X1$
- One item is $Consumed$
- $out = (out + 1)\ mod\ n = 1\ mod\ 8 = 1$
- Remove item X1 from buffer.
- count = count – 1, how it will be executed -
- load $R_c, m\ [count]\quad i.e.\ R_c = 3$
- Decr R_c i.e. $R_c = 2$
- Suppose again the system is preempted and store m [count], R_c is not executed.

```
/*Producer*/
int count  = 0;  // shared variable
void producer (void)
{
int itemp ;
while ( true)
{
produce_item ( itemp );
while ( count == n ); //overflow
buffer [ in ] = item;
in = ( in + 1) mod n;
count = count + 1; /* shared variable,
          how many items are in buffer*/
 }
```

- Now due to preemption of Consumer process CPU will execute another process.
- Now Producer will resume (not started) and execute $store\ m[Count], R_p$ i.e. $R_p = 4$
- count will be updated now & count = 4
- Producer terminated.
- Now Consumer will resumes its execution and will executed third instruction i.e. $store\ m\ [Count];\ R_c$ i.e. $R_c = 2$
- Which will decrement the value of count and count will become, count = 2
- Now Consumer process will also terminated.
- Now observe the value of count which is 2, which shows 2 items in the buffer.
- But there are three items in the buffer. But count value is showing wrongly.
- It is a Race Condition i.e. data inconsistency.
- Process synchronization is not achieved in this case.
- This problem can be solved another synchronization tool such as Semaphore.

Solution of Bounder Buffer Problem or Producer – Consumer Problem using Semaphore

- In this solution we use two counting semaphore empty and full and one binary semaphore S:

1. S = a binary semaphore with initial value 1 which is used to acquire and release the lock
2. empty = a counting semaphore whose initial value is the number of slots in the buffer, since, all slots are empty.
3. full = a counting semaphore whose initial value is 0.

```
wait(S)
{
  while(S<=0);
    S=S-1;
}
```

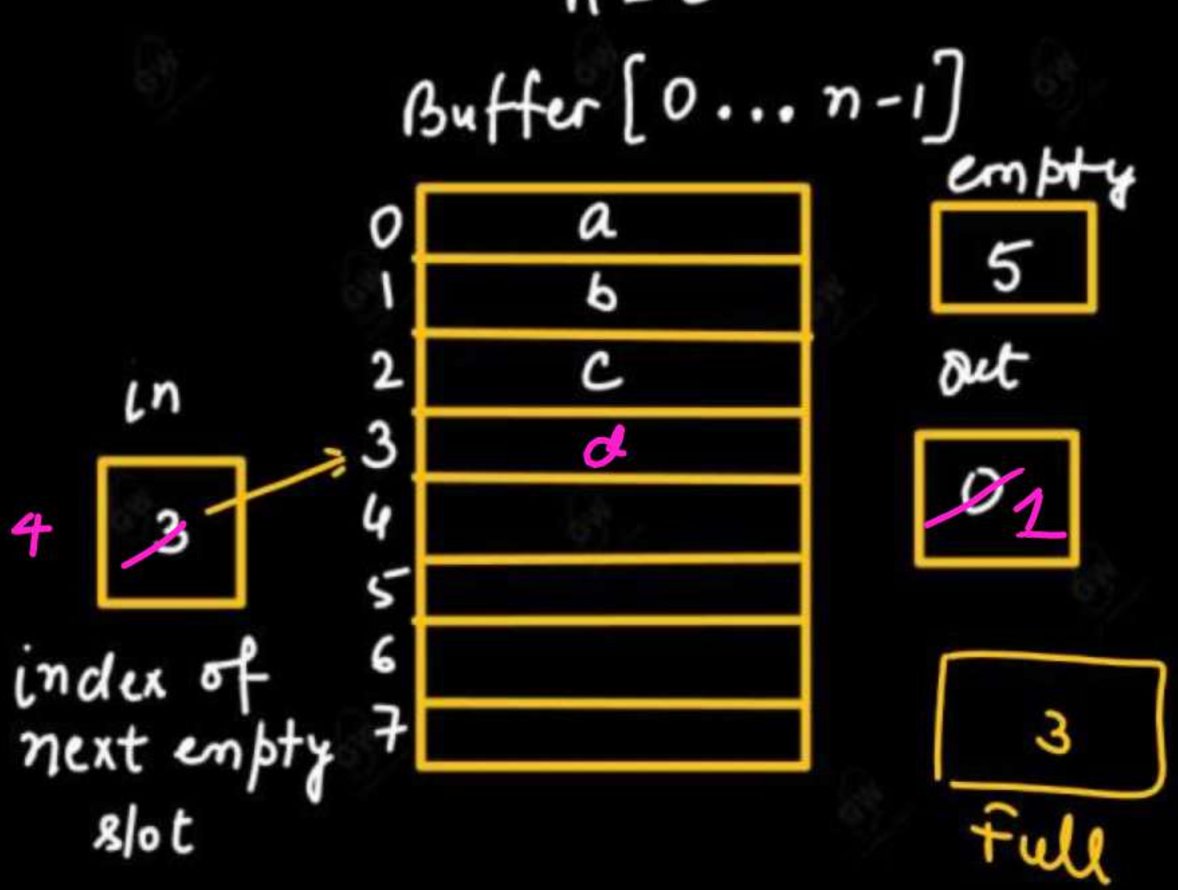
```
signal(S)
{
   S=S+1;
}
```

```
/* Producer Code*/
do
{
Produce_item ( itemp )
wait ( empty );
wait ( s ); // acquire lock;
/* add data to the buffer */
Buffer [ in ] = itemp;
in = ( in + 1 ) mod n;
signal ( s ); // release lock
signal ( full ); // increment full
}while ( TRUE );
```

Entry Section

CS

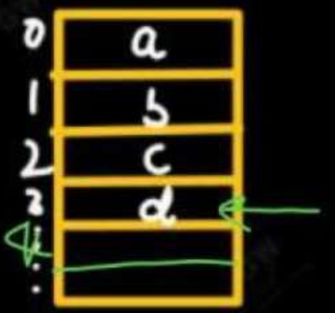
Exit section



```
/* Producer Code*/
do
{
Produce_item ( itemp )
wait ( empty );
wait ( s ); // acquire lock;
/* add data to the buffer */
Buffer [ in ] = itemp;
in = ( in + 1 ) mod n;
signal ( s ); // release lock
signal ( full ); // increment full
}while ( TRUE );
```

CS

- Here empty = 5, total number of slots in buffer n = 8, full = 3
- Let suppose Producer comes and produce an item. d (say)
- In the Producer code, perform the wait operation on the empty semaphore i.e. empty = 5 – 1 = 4
- Then it will execute wait (s) i.e. s = s – 1 = 1 - 1 = 0
- i.e. it will enter the critical section and put buffer [3] = itemp = d (say)



```
/* Producer Code*/
do
{
Produce_item ( itemp )
wait ( empty );
wait ( s ); // acquire lock;
/* add data to the buffer */
Buffer [ in ] = itemp;
in = ( in + 1 ) mod n;
signal ( s ); // release lock
signal ( full ); // increment full
}while ( TRUE );
```

- Then in = (3 + 1) mod 8 = 4 mod 8 = 4 i.e. next empty location in the buffer
- Then signal (s) i.e. s = s + 1 = 0 + 1 = 1 which says that other process now can enter the Critical Section.
- i.e. we have to achieve the progress i.e. Producer process have executed the critical section and now Consumer can execute in its critical section.
- Signal (full) i.e. full = full + 1 = 3 + 1 = 4 i.e. now buffer contains 4 items.
- Now let suppose Consumer process comes.

signal (S) { S = S+1 = 0+1 = 1 }

signal(full) { full = 3+1 = 4 }

signal(full) { full = full+1 = 3+1 = 4

```
/* Consumer Code*/
do
{
wait ( full );
wait ( s ); // acquire lock;
/*remove data from the buffer */
Itemc = buffer [ out ];
out = ( out + 1 ) mod n;
signal ( s ); // release lock
signal ( empty ); // increment empty
}while (TRUE);
```

Entry section

- First it will execute wait (full) i.e. full = full -1 = 4-1=3.
- Then wait(s) i.e. s = s – 1 = 1 – 1 = 0
- Then it will enter the critical section.
- Itemc = buffer [out] i.e. itemc = buffer [0] = a
- out = (0 + 1) mod 8 = 1
- Then signal (s) i.e. s = s + 1 = 0 + 1 = 1
- Then signal (empty) i.e. empty = empty + 1 = 4 + 1 = 5
- i.e. one empty slot has been increased.
- i.e. without any preemption, Producer and Consumer both are executing perfectly in the synchronized manner.

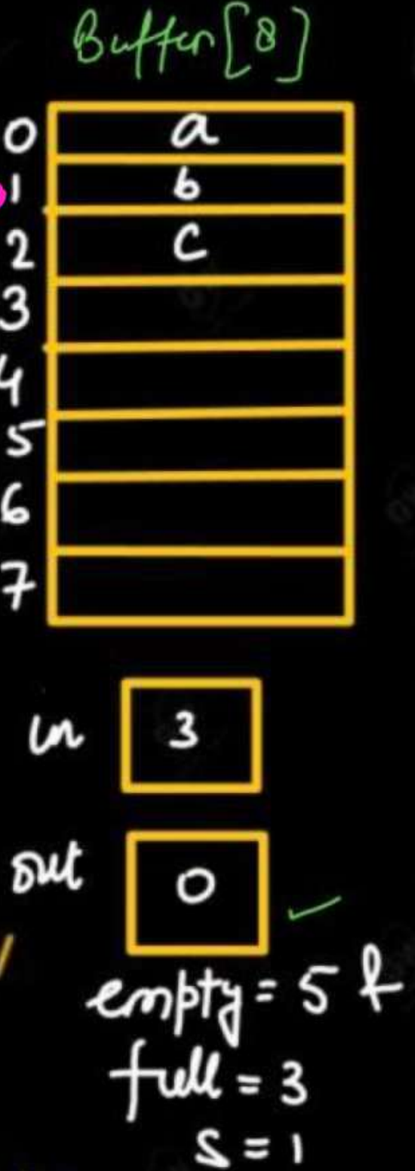
- Now check the solution with preemption.

```
/* Producer Code*/
do
{
Produce_item(itemp)
wait(empty);
wait(s); // acquire lock;
/* add data to the buffer */
Buffer[in]=itemp;
In=(in+1) mod n;
signal (s); // release lock
signal(full); // increment full
}while (TRUE);
```

} Entry Section

```
/* Consumer Code*/
do
{
wait(full);
wait(s); // acquire lock;
/* remove data from the buffer */
Itemc=buffer[out];
Out=(out+1) mod n;
signal (s); // release lock
signal(empty); // increment empty
}while (TRUE);
```

```
/* Producer Code*/
do
{
Produce_item ( itemp )
wait ( empty );
wait ( s ); // acquire lock;
/* add data to the buffer */
Buffer [ in ] = itemp;
in = ( in + 1 ) mod n;
signal ( s ); // release lock
signal ( full ); // increment full
} while ( TRUE );
/* Consumer Code*/
do
{
wait ( full );
wait(s); // acquire lock;
/* remove data from the buffer */
itemc = buffer [ out ];
out = ( out + 1 ) mod n;
signal ( s ); // release lock
signal ( empty ); // increment empty
}while ( TRUE );
```



- wait(empty) i.e. empty = empty - 1 = 5 – 1 = 4
- Now suppose preemption occurs.
- Now Consumer will be executed
- wait (full) i.e. full = full – 1 = 3 – 1 = 2
- wait (s), s = s – 1 = 1 – 1 = 0
- Will enter the critical section i.e. itemc = buffer [0] = 'a'
- out = (0 + 1) mod 8 = 1 mod 8 = 1
- Now suppose again preemption.
- Producer comes.
- Now Producer will execute (restored)
- wait (s) i.e. ~~s = s – 1 = 0 – 1 not possible~~, will be stuck in the loop. i.e. can not enter the critical section
- Again Consumer will be executed
- It will execute signal (s) i.e. s = s + 1 = 0 + 1 = 1
- signal (empty) i.e. empty = empty + 1 = 1 + 5 = 6
- i.e. **mutual exclusion is achieved**

AKTU PYQs

Q.1. State the Producer-consumer problem. Given a solution to the solution using semaphores.

2016-17, 5 Marks, 2017-18, 10 Marks, 2022-23, 10 Marks

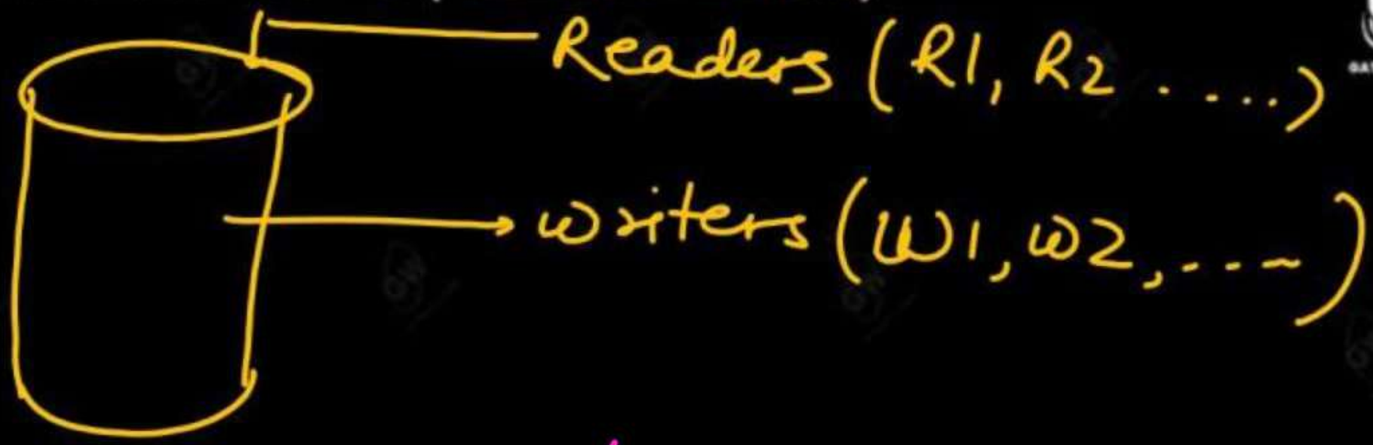
Q.2. What is producer Consumer problem? How it can illustrate the classical problem of synchronization? Explain.

2018-19, 7 Marks

The Readers – Writers Problem

- Suppose that a database is to be shared among several concurrent processes.
- Some of these processes may want only to read the database, where as others may want to update the database (i.e. to read and write both).
- We distinguish between these two type of processes by referring to the former as **Readers** and to the later as **Writers**.
- Obviously, if two Readers access the shared data simultaneously, no adverse effects will result.
- However, if a Writer and some other process (either a Reader or a Writer) access the database simultaneously, chaos may ensure.
- To ensure that these difficulties do not arise, we require that the Writers have exclusive access to the shared database while writing to the database.
- This synchronization problem is referred to as the **Readers - Writers problem.**

- Multiple users can use the database at the same time. (Readers or Writers)



- Possible Cases -

Case 1: One Reader is already using the database and Writer comes, that is a problem. (on same database). R – W problem.

Case 2: One Writer is already using the database and Reader comes. W – R problem.

Case 3: One Writer is updating the database and at the same time another Writer comes and try to update the same database. W-W problem.

Case 4: One Reader is reading the database and another Reader is coming and reading the same database. R – R, this is allowed, no problem.

GW GATEWAY CLASSES

① ② ③ ④ RR

- Due to R – W, W – R and W – W problem there may be data inconsistency and even data may be lost.
- To over come this problem we have to use synchronization tools such as Semaphore, Monitor, Locks etc.
- The code must be written so that R-W, W-R and W-W problem may be avoided.
- Readers and writers may be synchronized by Semaphore.

```
/* Readers – Writers Problem Solution using
Binary Semaphore*/
int rc = 0 /* read count, how many Readers are
there in the database or Buffer i.e. how many
Readers are using the database*/
semaphore mutex = 1;
semaphore db = 1;
/*Reader Code*/
void Reader ( void )
{
while ( true )
{
wait ( mutex ); /*down*/
rc = rc + 1;
if ( rc == 1 ) then wait (db);
signal ( mutex ); /*up*/
DB; /* Critical Section */
wait ( mutex );
rc = rc – 1;
if ( rc == 0 ) then signal ( db );
signal ( mutex );
process_data;
}
}
/*Writer Code*/
void Writer ( void )
{
while ( true );
{
wait ( db );
DB; /*Critical Section*/
signal ( db );
}
}
```

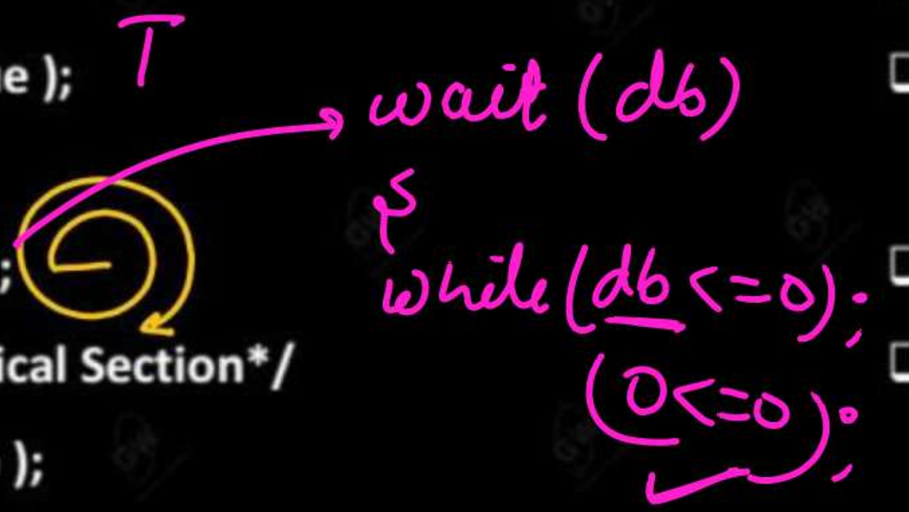
CS

wait (mutex)
{
if (mutex <= 0);
mutex = mutex
-1;
}
signal (mutex)
{ mutex = mutex+1}

```
int rc = 0;
semaphore mutex = 1;
semaphore db = 1;
void Reader ( void )
{
while ( true )
{
wait ( mutex ); /*down*/
rc = rc + 1; /* readerCount */
if ( rc == 1 ) then wait (db);
signal ( mutex ); /*up*/
DB; /*Critical Section */
wait ( mutex );
rc = rc – 1;
if ( rc == 0 ) then signal ( db );
signal ( mutex );
process_data;
}
}
```

- **Case – 1:**
- R - W (Read – Write), one Reader process and one Writer process.
- let suppose Reader R1 comes first .
- The initial value of rc = 0 (i.e. Reader count = 0), mutex = 1 and db = 1;
- while loop – True
- wait (mutex) i.e. mutex = mutex – 1 = 1 – 1 = 0
- rc = rc + 1 = 0 + 1 = 1
- if (rc == 1) i.e. 1 = 1 i.e. true, then
- wait (db) i.e. db = db – 1 = 1 – 1 = 0
- signal (mutex) i.e. mutex = mutex + 1 = 0 + 1 = 1
- Reader R1 will enter the DB and can read the same.

```
/*Writer Code*/
void Writer ( void )
{
while ( true );
{
wait ( db );
DB; /*Critical Section*/
signal ( db );
}
}
```



- Now suppose Writer comes.
- Writer Code will be executed.
- while loop – True
- wait (db) i.e. if (db <= 0); i.e. (0 <= 0); it will be True stuck in while loop.
- In this case the Writer will be blocked.
- i.e. if one Reader is reading the database, the Writer should not use the database.
- First case is established.

```
/*Writer Code*/
void Writer ( void )
{
while ( true );
{
wait ( db );
DB; /*Critical Section*/
signal ( db );
}
}
```

rc=0
mutex = 1
db = 1

```
wait (db)
{ while (db<=0);
```
(1<=0) : F
```
db=db-1;
}
```

W R

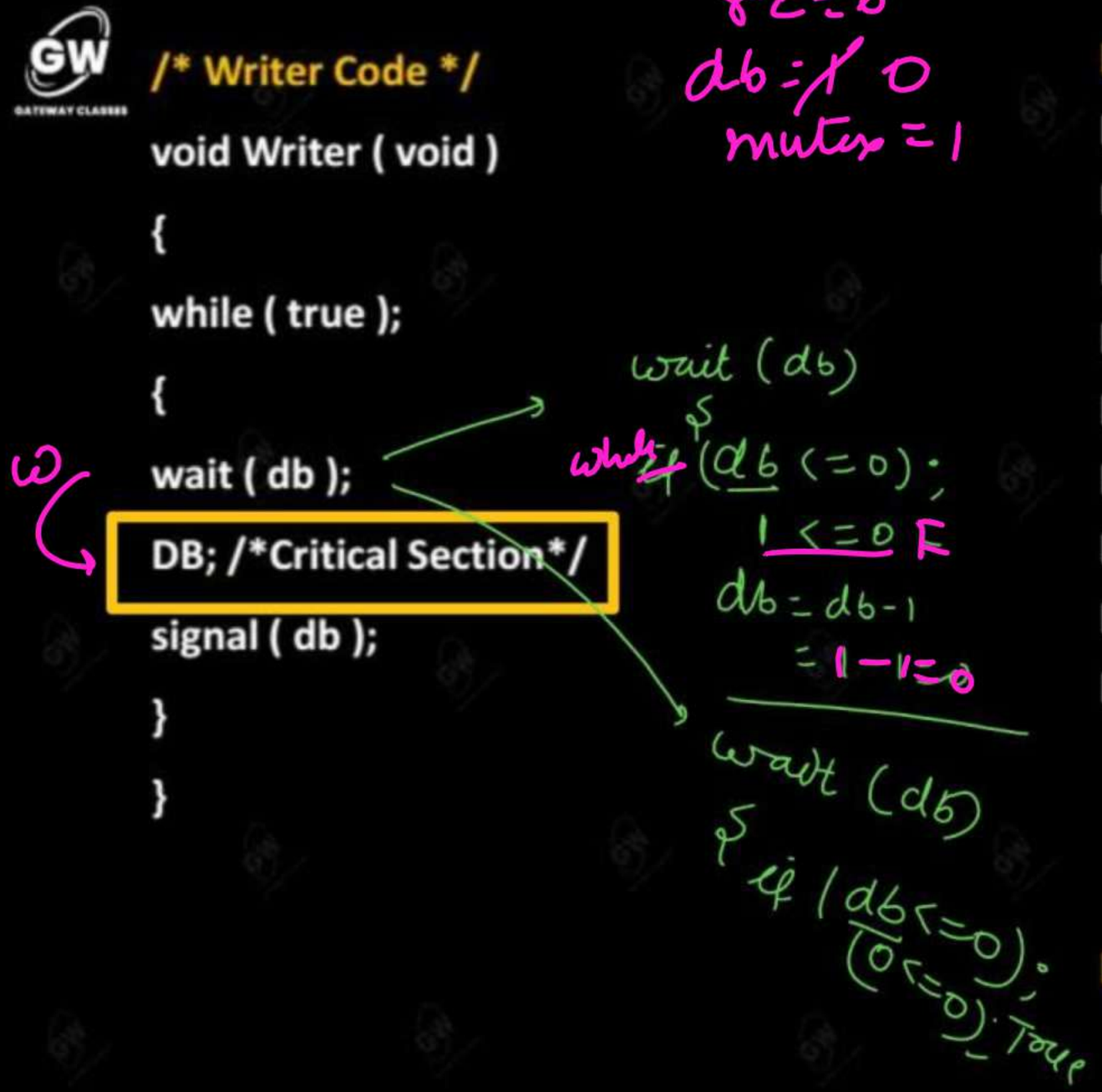
- Case – 2 - Write – Read Problem
- Let suppose Writer W1 comes first.
- Writer code will be executed.
- While (true) – True
- Wait (db) i.e. db = db – 1 = 1 – 1 = 0.
- Writer W1 will enter the DB. or CS

```
wait ( db)
{ if (db<=0);
```
1<=0 False
db=db-1 = 1-1=0.

```
int rc = 0;
semaphore mutex = 1;
semaphore db = 1;
void Reader ( void )
{
while ( true )
{
wait ( mutex ); /*down*/
rc = rc + 1;
if ( rc == 1 ) then wait (db);
signal ( mutex ); /*up*/
DB; /*Critical Section */
wait ( mutex );
rc = rc – 1;
if ( rc == 0 ) then signal ( db );
signal ( mutex );
process_data;
}
}
```

- Suppose Reader R1 comes.
- Reader code will be executed.
- while (true) is True
- wait (mutex), mutex = mutex – 1 = 1 – 1 = 0
- rc = rc + 1 = 0 + 1 = 1
- It (rc = = 1) then wait (db) i.e. if (db < = 0); ie. (0 < = 0); true, i,.e Reader will be stuck in while loop and the Reader will be blocked.
- Case 2 is also established i.e. when one Writer is writing the database, another Reader can not read the database.

```
/* Writer Code */
void Writer ( void )
{
while ( true );
{
wait ( db );
DB; /*Critical Section*/
signal ( db );
}
}
```



- Case – 3 - Write – Write Problem
- Let suppose a Writer W1 comes.
- Writer code will be executed.
- while (true) – True
- wait (db) i.e. db = db – 1 = 1 – 1 = 0.
- Writer will enter DB (successfully).
- Now suppose another Writer W2 comes.
- Writer code will be executed.
- while (true) = True
- wait (db) i.e. if (db < = 0); i.e. (0 < = 0); i.e. the condition is true and the Writer W2 will struck in the while loop and W2 will be blocked.
- i.e. Case – 3 is also established i.e. when one writer is using the DB another Writer can not enter the DB.

```
int rc = 0;
semaphore mutex = 1;
semaphore db = 1;
void Reader ( void )
{
while ( true )
{
wait ( mutex ); /*down*/
rc = rc + 1;
if ( rc == 1 ) then wait (db);
signal ( mutex ); /*up*/
DB; /*Critical Section */
wait ( mutex );
rc = rc – 1;
if ( rc == 0 ) then signal ( db );
signal ( mutex );
process_data;
}
}
```

- Case – 4:
- R - R (Read – Read)
- let suppose Reader R1 comes first .
- The initial value of rc = 0 (Reader count = 0), mutex = 1 and db = 1;
- while loop – True
- wait (mutex) i.e. mutex = mutex – 1 = 1 – 1 = 0
- rc = rc + 1 = 0 + 1 = 1
- if (rc == 1) i.e. 1 = 1 i.e. true, then
- wait (db) i.e. db = db – 1 = 1 – 1 = 0
- signal (mutex) i.e. mutex = mutex + 1 = 0 + 1 = 1
- Reader R1 will enter the DB and can read the same.

sc=φ 1

mutex=1

db=1

wait (mutex)

{ while (mutex <=0);

(1<=0); F

mutex = mutex - 1

= 1-1=0

---

wait (db)

{ while (db<=0);

1<=0 F

db=db-1

=1-1=0

Signal (mutex)

{ mutex = mutex+1

= 0+1=1

wait (mutex)

{ while (mutex <=0);

(1<=0) F

mutex = 0

sc=2

2==1 F

Signal (mutex)

mutex=1

```
int rc = 0;
semaphore mutex = 1;
semaphore db = 1;
void Reader ( void )
{
while ( true )
{
wait ( mutex ); /*down*/
rc = rc + 1;
if ( rc == 1 ) then wait (db);
signal ( mutex ); /*up*/
DB; /*Critical Section */
wait ( mutex );
rc = rc – 1;
if ( rc == 0 ) then signal ( db );
signal ( mutex );
process_data;
}
}
```

- let suppose another Reader R2 comes.
- while loop – True
- wait (mutex) i.e. mutex = mutex – 1 = 1 – 1 = 0
- rc = rc + 1 = 1 + 1 = 2 false
- if (rc == 1) i.e. 2 = 1 i.e. ~~true~~, then
- wait (db) i.e. db = db – 1 = 1 – 1 = 0
- signal (mutex) i.e. mutex = mutex + 1 = 0 + 1 = 1
- Reader R2 will enter the DB and can read the same.
- i.e. if one Reader is reading the DB, another Readers can also enter the DB.

GW
GATEWAY CLASSES

AKTU PYQs

Q.1. Give the solution of Readers – Writers problem by using the concept of Semaphore ?

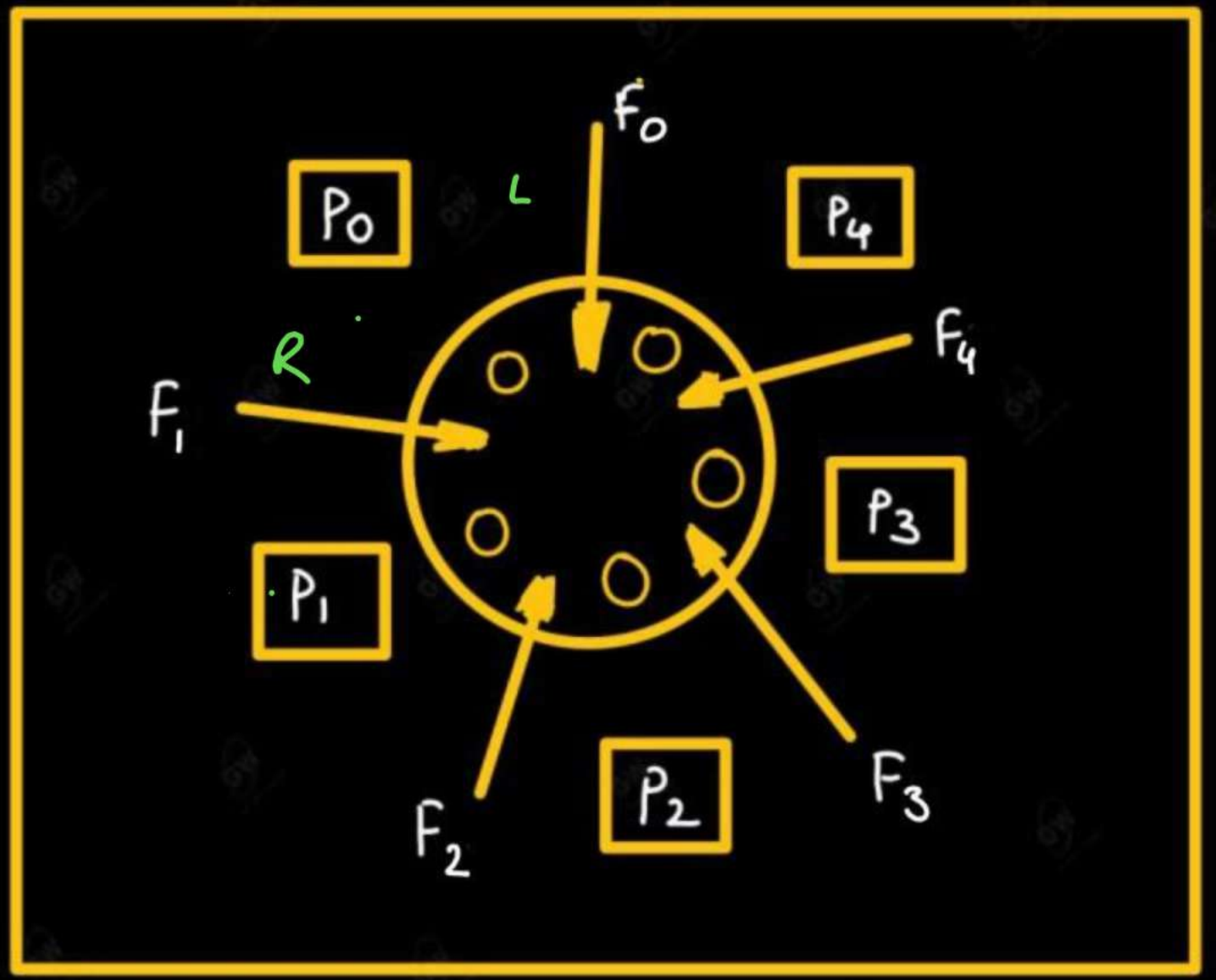
2015-16, 5 Marks

The Dining – Philosophers Problem

- The Dining Philosophers problem is a classic synchronization problem in computer science.
- It was introduced by Dijkstra in 1965 to illustrate the challenges of resource allocation and deadlock avoidance in concurrent systems.
- In the Dining - Philosophers problem, there are five philosophers say P_0, P_1, P_2, P_3, P_4 sitting around a dining table.
- Each philosopher alternates between thinking and eating.
- In the center of the table is a bowl of rice and the table is laid with five single chopsticks/forks. $(F_0, F_1, F_2, F_3, F_4)$
- **Philosopher States : -**

  1. Thinking
  2. Eating

THE SITUATION OF DINING PHILOSOPHERS



5 -

- PHILOSOPHERS :- $\{P_0, P_1, P_2, P_3, P_4\}$
- FORKS – 5 $\{f_0, f_1, f_2, f_3, f_4\}$

- A philosopher may pick up only one chopstick at a time.
- Obviously a philosopher can not pick up a chopstick that is already in the hand of a neighbor.
- When a hungry philosopher has both chopstick at the same time he / she eats without releasing his/her chopsticks.
- When the philosopher is finished eating he / she put down both chopsticks and starts thinking again.

/ * Philosopher Code : Every Philosopher will follow this code */

// N = Number of forks $N = 5$

```
void Philosopher ( void )
{                                   P0
while ( true )
{
thinking ();
take_fork ( i ); // left fork                    } ENTRY
take_fork ( ( i + 1 ) mod N ); // right fork     } CODE
eat(); // Critical Section                       CS
put_fork ( i ); // left fork                    } EXIT
put_fork ( ( i + 1 ) mod N ); // right fork      } CODE
}
}
```

f0 f1 f0 f1

Conditions:

1. Pick left fork
2. Pick right fork
3. Eating
4. Put left fork
5. Put right fork

```
/ * Philosopher Code : Every Philosopher will
follow this code */
// N = Number of forks
void Philosopher ( void )
{
while ( true )
{
thinking ();
take_fork ( i ); // left fork
take_fork ( ( i + 1 ) mod N ); // right fork
eat(); // Critical Section
put_fork ( i ); // left fork
put_fork ( ( i + 1 ) mod N ); // right fork
}
}
```

- Case – 1:
- Say P0 comes
- thinking ( )
- take ith fork i.e. 0th fork // f0, left fork
- take (ith + 1) mod N fork i.e. f [(0 + 1) mod N] /* N is number of forks*/
- i.e. 1 mod 5 = f1 fork
- Pick f1 fork i.e. right fork
- Start eating / * critical section */
- Put left fork i.e. f0
- Put right fork i.e. f1
- Work done

```
/ * Philosopher Code : Every Philosopher will
follow this code */
// N = Number of forks
void Philosopher ( void )
{
while ( true )
{
thinking ();
take_fork ( i ); // left fork
take_fork ( ( i + 1 ) mod N ); // right fork
eat(); // Critical Section
put_fork ( i ); // left fork
put_fork ( ( i + 1 ) mod N ); // right fork
}
}
```

- Say P1 comes
- thinking ( )
- take ith fork i.e. f1 fork (left)
- Take right fork i.e. f2 fork f [(1 + 1) mod 2] = 2 /* n is number of forks*/
- Start eating / * critical section */
- Put left fork i.e. f1 fork (ith)
- Put right fork i.e. f2 fork (i + 1) mod N
- Work done
- This is the normal sequence.
- In this philosophers are coming in the normal sequence.
- Then this code will work perfectly.

P0 P1
f0 f1 f2

```
/ * Philosopher Code : Every Philosopher will
follow this code */
// N = Number of forks
void Philosopher ( void )
{
while ( true )
{
thinking ();
take_fork ( i ); // left fork
take_fork ( ( i + 1 ) mod N ); // right fork
eat(); // Critical Section
put_fork ( i ); // left fork
put_fork ( ( i + 1 ) mod N ); // right fork
}
}
```

- **Case 2:-**
- Let suppose P_0 comes. (i = 0)
- Thinking
- Take left fork i.e. f0
- Before taking the right fork say P_1 comes
- Now P_1 will start the execution of the same code.
- Take left fork i.e. f1 ($i = 1$)
- And P0 is waiting for fork f1, right fork of P0
- F_1 will be provided to P_0 , when F_1 & F_2 both will be provided to P_1 and P_1 will eat then it will put both fork F_1 & F_2 .
- Then f1 will be provided to P_0.
- Means 2 or more philosophers are coming simultaneously at a time then, there may be a problem of Race Condition / Data Inconsistency
- This problem can be solved by using Binary Semaphore.

Solution to the Dining – Philosopher Problem using Semaphore

- In this solution array of semaphore will be used – S[i]
- There are five forks i.e. 5 semaphores will be used
- i.e. array of 5 S, S[i]; S[5] –
- Let suppose all are initialized by –
- Now the modified code for Dining – Philosopher will be as follows-

S[5]

S[0]	S[1]	S[2]	S[3]	S[4]

S_0	S_1	S_2	S_3	S_4
0	0	1	1	1

```
/ * Dining Philosopher Problem Solution using
Binary Semaphore */
void philosopher ( void)
{
while (true)
{
thinking();
wait ( take_fork ( Si ) );
wait ( take_fork ( ( Si + 1 ) mod N );
eat();
signal ( put_fork ( i ); // left fork
signal ( put_fork ( ( i + 1 ) mod N );
}
}
```

ENTRY CODE

EXIT CODE

- Let suppose P0 comes
- P0 want to take both forks f0 and f1
- It means it needs 2 semaphores: S0 and S1 to execute.
- Let suppose P1 comes
- It needs two semaphores S1 and S2
- When P2 comes, it will need S2 and S3
- When P3 comes, it will need S3 and S4
- If P4 comes, it will need S4 and S0

	L	R
P0	S0	S1
P1	S1	S2
P2	S2	S3
P3	S3	S4
P4	S4	S0

Types of Semaphore

- There are two types of semaphore –

(i) Binary and { Semaphore var S: 0, 1 }

(ii) Counting Semaphore { ——— S: 0, 1, 2, 3 ... Positive int }

- The Binary semaphore can have 2 values – 0 and 1
- The counting semaphore can have any value i.e. any positive integer from 0 to n.
- The binary semaphore is used to implement the solution of critical section problems with multiple processes.
- The binary semaphore is used to control access to a recourse that has multiple instances.
- It is used in Reader - Writer Problem, Bounded - Buffer Problem etc.

```
/ * Dining Philosopher Problem Solution using
Binary Semaphore */
void philosopher ( void)
{
while (true)
{
thinking();
wait ( take_fork ( Si ) );
wait ( take_fork ( ( Si + 1 ) mod N );
eat();
signal ( put_fork ( i ); // left fork
signal ( put_fork ( ( i + 1 ) mod N );
}
}
```

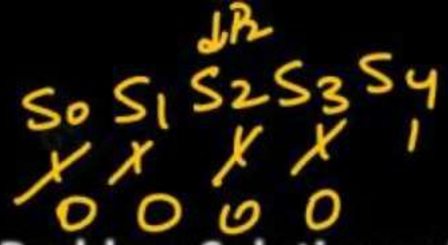
P0

wait (S0)
{ if (S0<=0); F
S0 = S0-1;
}

S0 f0 S1 f1

- Then only they will go in eat() i.e Critical section
- /* Let suppose all the semaphores are initialized by 1, if it will be 0, it will be blocked because the wait will go in infinite loop*/
- Case 1:
- Let suppose P0 comes
- wait of S0 (initial 1, will be 0)
- wait of S1 (initial 1, will be 0)

} ENTRY CODE




```
/ * Dining Philosopher Problem Solution using
Binary Semaphore */
void philosopher ( void)
{
while (true)
{
thinking();
wait ( take_fork ( Si ) );
wait ( take_fork ( ( Si + 1 ) mod N );
eat();
signal ( put_fork ( i ); // left fork
signal ( put_fork ( ( i + 1 ) mod N );
}
}
```

- P0 will enter eat() state i.e. critical section
- Again S0 will become 1 from 0
- And S1 will become 1 from 0

} EXIT CODE

- **Take another case:**
- Suppose P0 comes first (initial S0, S1, S2, S3, S4 all values are 1)
- S0 value will become 0 (wait /down)
- S1 value will become 0 (wait /down)

} ENTRY CODE

- P0 will enter Eat() // CS
- At the same time if P1 comes
- P1 will also wait the S1 value that is already 0 - it means P1 will be blocked
- Now if P2 comes
- P2 will change the value of S2 to 0 and S3 to 0
- And can enter to eat state
- **Now in this way there are two processes P0 and P2 both are in Critical Section**

```
/ * Dining Philosopher Problem Solution using
Binary Semaphore */
void philosopher ( void)
{
while (true)
{
thinking();
wait ( take_fork ( Si ) );
wait ( take_fork ( ( Si + 1 ) mod N );
eat();
signal ( put_fork ( i ); // left fork
signal ( put_fork ( ( i + 1 ) mod N );
}
}
```

- Mutual exclusion means there will be only one process in the critical section at a time and there are two processes P0 and P2 are in the Critical section but it is to be noted that P0 and P2 both are independent processes, they are non cooperative processes.
- It is special case in Dining – Philosophers problem where there are two processes in the Critical Section but both are independent processes because both are using different forks. (resources are different) (they are independent in terms of forks)

AKTU PYQs

Q.1. Explain in detail about the Dining Philosopher Problem.

2021-22, 10 Marks

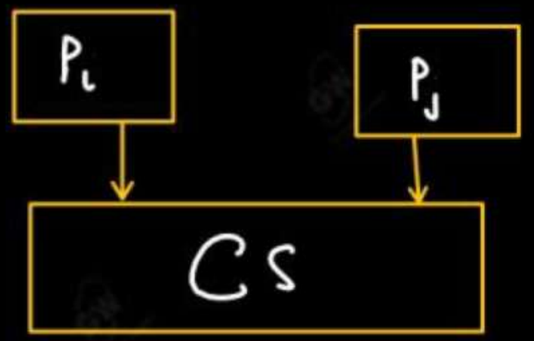
Q.2. Explain dinning philosopher problem and its solution using semaphore.

2022-23, 10 Marks

The Dekker's Algorithm in Synchronization

- To obtain such a mutual exclusion, bounded waiting, and progress there have been several algorithms implemented, one of which is Dekker's Algorithm.
- Dekker's algorithm is the first solution of critical section problem.
- This solution satisfy the mutual exclusion, progress and bounded waiting.
- This algorithm is for two processes.
- Let us see the code of both the processes Pi and Pj-

* Mutual Exclusion
* Progress
* Bounded wait



if Both wish to enter the Critical section

Process Pi

```
initially flag [ i ] = false;
do
{
flag [ i ] = true;
while ( flag [ j ] )
{
if ( turn == j )
{
flag [ i ] = false;
while ( turn == j );
flag [ i ] = true;
}
}
Critical Section
turn = j;
flag [ i ] = false;
remainder section
} while ( true );
```

Process Pj

```
initially flag [ j ] = false;
do
{
flag [ j ] = true;
while ( flag [ i ] )
{
if ( turn == i )
{
flag [ j ] = false;
while ( turn == i );
flag [ j ] = true;
}
}
Critical Section
turn = i;
flag [ j ] = false;
remainder section
} while ( true );
```

- Initially both processes will false their flag value i.e. no process want to enter its critical section.
- Suppose both want to enter the critical section.
- Suppose Pi wish to enter the critical section then first it will true its flag value.
- Suppose Pj wish to enter the critical section then first it will true its flag value.

Process Pi

```
initially flag [ i ] = false;
do
{
flag [ i ] = true;
while ( flag [ j ] )
{
if ( turn == j )
{
flag [ i ] = false;
while ( turn == j );
flag [ i ] = true;
}
}
Critical Section
turn = j;
flag [ i ] = false;
remainder section
} while ( true );
```

Process Pj

```
initially flag [ j ] = false;
do
{
flag [ j ] = true;
while ( flag [ i ] )
{
if ( turn == i )
{
flag [ j ] = false;
while ( turn == i );
flag [ j ] = true;
}
}
Critical Section
turn = i;
flag [ j ] = false;
remainder section
} while ( true );
```

- That is both want to enter the critical section and make their flag value true.
- Now Process Pi will check the while condition –

while (flag [j]) i.e. till the value of flag [j] is true and

- Then in while loop it will execute – if (turn == j)
- i.e. if turn is of process Pj i.e. each process is checking if flag of other process is true and turn is also of other process,

Process Pi

```
initially flag [ i ] = false;
do
{
flag [ i ] = true;
while ( flag [ j ] )
{
if ( turn == j )
{
flag [ i ] = false;
while ( turn == j );
flag [ i ] = true;
}
}
Critical Section
turn = j;
flag [ i ] = false;
remainder section
} while ( true );
```

Process Pj

```
initially flag [ j ] = false;
do
{
flag [ j ] = true;
while ( flag [ i ] )
{
if ( turn == i )
{
flag [ j ] = false;
while ( turn == i );
flag [ j ] = true;
}
}
Critical Section
turn = i;
flag [ j ] = false;
remainder section
} while ( true );
```

- If condition is true, then process Pi will false its flag by executing the next instruction – flag [i] = false;
- i.e. every process check flag and turn of other process if flag of other process is true and the turn is of other process, it false its own flag.
- Then next instruction –

  while (turn == j);

  will be executed,

Process Pi

```
initially flag [ i ] = false;
do
{
flag [ i ] = true;
while ( flag [ j ] )
{
if ( turn == j )
{
flag [ i ] = false;
while ( turn == j );
flag [ i ] = true;
}
}
Critical Section
turn = j;
flag [ i ] = false;
remainder section
} while ( true );
```

Process Pj

```
initially flag [ j ] = false;
do
{
flag [ j ] = true;
while ( flag [ i ] )
{
if ( turn == i )
{
flag [ j ] = false;
while ( turn == i );
flag [ j ] = true;
}
}
Critical Section
turn = i;
flag [ j ] = false;
remainder section
} while ( true );
```

- i.e. if the turn is of other process then process Pi will be blocked and will not enter the critical section.
- In this situation other process Pj will be executed in their critical section.
- When this condition –

  while (turn == j);

will be false, i.e. the turn is not of other process, i.e. process Pj is not executing in its critical section.

Process Pi

```
initially flag [ i ] = false;
do
{
flag [ i ] = true;
while ( flag [ j ] )
{
if ( turn == j )
{
flag [ i ] = false;
while ( turn == j );
flag [ i ] = true;
}
}
Critical Section
turn = j;
flag [ i ] = false;
remainder section
} while ( true );
```

Process Pj

```
initially flag [ j ] = false;
do
{
flag [ j ] = true;
while ( flag [ i ] )
{
if ( turn == i )
{
flag [ j ] = false;
while ( turn == i );
flag [ j ] = true;
}
}
Critical Section
turn = i;
flag [ j ] = false;
remainder section
} while ( true );
```

- Then next instruction –

  flag [i] = true;

  will be executed
- i.e. process Pi will true its flag and will enter the critical section.
- And after executing in the critical section, process Pi will set the turn of another process Pj by executing the next instruction –

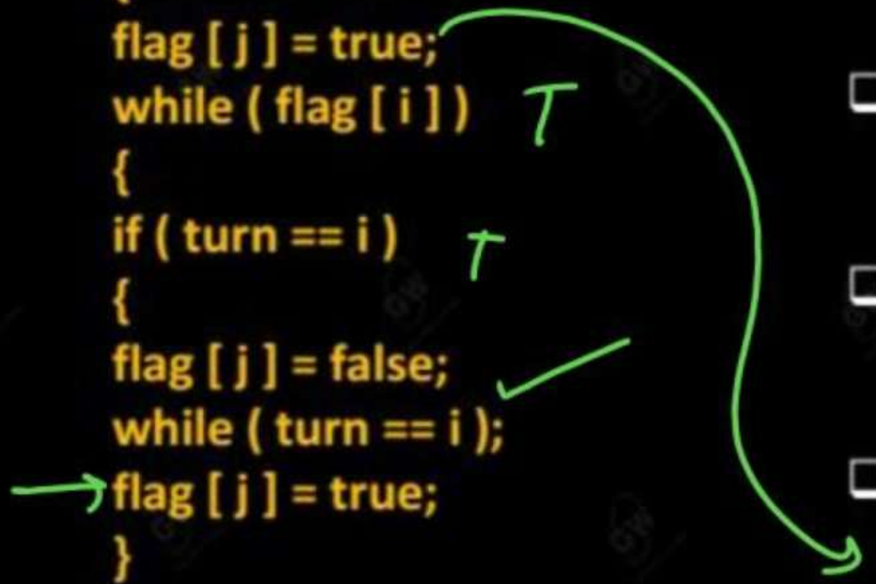
  turn = j;

Process Pi

```
initially flag [ i ] = false;
do
{
flag [ i ] = true;
while ( flag [ j ] )
{
if ( turn == j )
{
flag [ i ] = false;
while ( turn == j );
flag [ i ] = true;
}
}
Critical Section
turn = j;
flag [ i ] = false;
remainder section
} while ( true );
```

Process Pj

```
initially flag [ j ] = false;
do
{
flag [ j ] = true;
while ( flag [ i ] )
{
if ( turn == i )
{
flag [ j ] = false;
while ( turn == i );
flag [ j ] = true;
}
}
Critical Section
turn = i;
flag [ j ] = false;
remainder section
} while ( true );
```



- And will false its flag value by executing the next instruction- flag [i] = false;
- Then will execute the remainder section.
- In the same manner process Pj will be executed.
- Suppose Pj wish to enter the critical section then first it will true its flag value.

Process Pi

```
initially flag [ i ] = false;
do
{
flag [ i ] = true;
while ( flag [ j ] )
{
if ( turn == j )
{
flag [ i ] = false;
while ( turn == j );
flag [ i ] = true;
}
}
Critical Section
turn = j;
flag [ i ] = false;
remainder section
} while ( true );
```

Process Pj

```
initially flag [ j ] = false;
do
{
flag [ j ] = true;
while ( flag [ i ] )
{
if ( turn == i )
{
flag [ j ] = false;
while ( turn == i );
flag [ j ] = true;
}
}
Critical Section
turn = i;
flag [ j ] = false;
remainder section
} while ( true );
```

- Now Process Pj will check the while condition –

  while (flag [i])

  i.e. till the value of flag [i] is true and
- Then in while loop it will execute – if (turn == i)
- i.e. if turn is of process Pi
- i.e. each process is checking if flag of other process is true and turn is also of other process,

Process Pi

```
initially flag [ i ] = false;
do
{
flag [ i ] = true;
while ( flag [ j ] )
{
if ( turn == j )
{
flag [ i ] = false;
while ( turn == j );
flag [ i ] = true;
}
}
Critical Section
turn = j;
flag [ i ] = false;
remainder section
} while ( true );
```

Process Pj

```
initially flag [ j ] = false;
do
{
flag [ j ] = true;
while ( flag [ i ] )
{
if ( turn == i )
{
flag [ j ] = false;
while ( turn == i );
flag [ j ] = true;
}
}
Critical Section
turn = i;
flag [ j ] = false;
remainder section
} while ( true );
```

- If condition is true, then process Pj will false its flag by executing the next instruction - flag [j] = false;
- i.e. every process check flag and turn of other process if flag of other process is ~~turn~~ true and the turn is of other process, it false its own flag.

Process Pi

```
initially flag [ i ] = false;
do
{
flag [ i ] = true;
while ( flag [ j ] )
{
if ( turn == j )
{
flag [ i ] = false;
while ( turn == j );
flag [ i ] = true;
}
}
Critical Section
turn = j;
flag [ i ] = false;
remainder section
} while ( true );
```

Process Pj

```
initially flag [ j ] = false;
do
{
flag [ j ] = true;
while ( flag [ i ] )
{
if ( turn == i )
{
flag [ j ] = false;
while ( turn == i );
flag [ j ] = true;
}
}
Critical Section
turn = i;
flag [ j ] = false;
remainder section
} while ( true );
```

- Then next instruction – while (turn == i); will be executed, i.e. if the turn is of other process then process Pj will be blocked and will not enter the critical section. In this situation other process Pi will be executed in their critical section.
- When this condition – while (turn == i); will be false, i.e. the turn is not of other process Pi, i.e. process Pj ~~is not executing~~ will not execute in its critical section.

Process Pi

```
initially flag [ i ] = false;
do
{
flag [ i ] = true;
while ( flag [ j ] )
{
if ( turn == j )
{
flag [ i ] = false;
while ( turn == j );
flag [ i ] = true;
}
}
Critical Section
turn = j;
flag [ i ] = false;
remainder section
} while ( true );
```

Process Pj

```
initially flag [ j ] = false;
do
{
flag [ j ] = true;
while ( flag [ i ] )
{
if ( turn == i )
{
flag [ j ] = false;
while ( turn == i );
flag [ j ] = true;
}
}
Critical Section
turn = i;
flag [ j ] = false;
remainder section
} while ( true );
```

- Then next instruction – flag [j] = true; will be executed i.e. process Pj will true its flag and will enter the critical section. And after executing in the critical section, process Pj will set the turn of another process Pi by executing the next instruction – turn = i;
- And will false its flag value by executing the next instruction –flag [j] = false;
- Then will execute the remainder section.

Process Pi

```
initially flag [ i ] = false;
do
{
flag [ i ] = true;
while ( flag [ j ] )
{
if ( turn == j )
{
flag [ i ] = false;
while ( turn == j );
flag [ i ] = true;
}
}
Critical Section
turn = j;
flag [ i ] = false;
remainder section
} while ( true );
```

Process Pj

```
initially flag [ j ] = false;
do
{
flag [ j ] = true;
while ( flag [ i ] )
{
if ( turn == i )
{
flag [ j ] = false;
while ( turn == i );
flag [ j ] = true;
}
}
Critical Section
turn = i;
flag [ j ] = false;
remainder section
} while ( true );
```

- Consider a case, if Process Pi has executed its instruction –

  flag [i] = true; and executed the next instruction – while (flag [j]) before executing the instruction – flag [j] = true by process Pj i.e. the instruction of process Pi – while (flag [j]) will become false then it will directly enter the critical section without entering the while loop where it first check if the turn is of process Pj and enter the if condition.

- The same scenario will be applicable for another process Pj.

GW
GATEWAY CLASSES

AKTU PYQs

Q.1. Explain the following terms briefly.

(i) Dekker's Solution

(ii) Busy Waiting

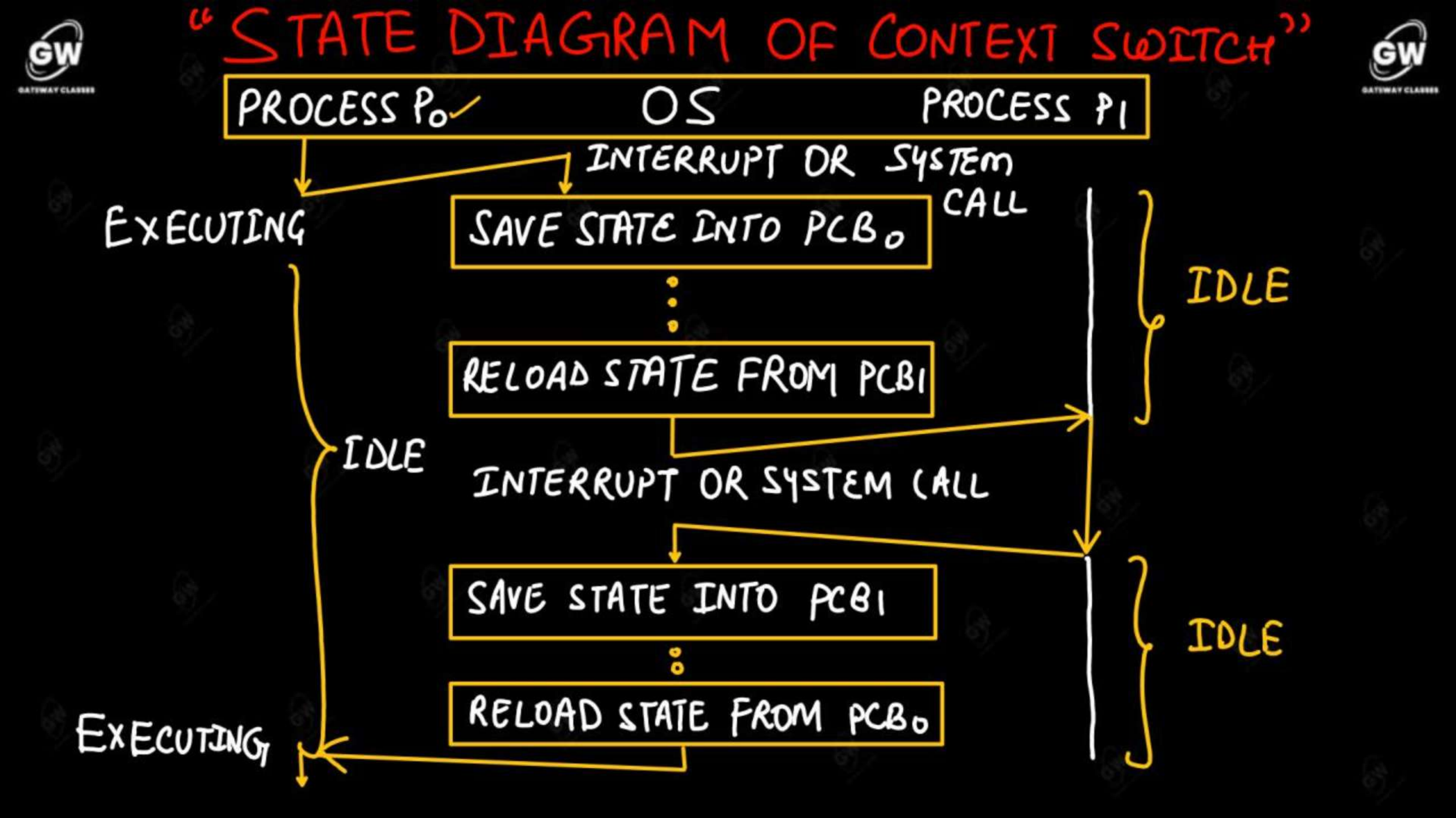
2014 - 15, 10 Marks

Q.2. Give the principles, mutual exclusion in critical section problem. Also discus how well these principles are followed in Dekker's solution.

2016 -17, 5 Marks

Context Switch

- Interrupts cause the Operating System to change a CPU from its current task and to run a kernel routine.
- When an interrupt occurs the system needs to save the current context of the process running on the CPU so that it can restore that context when its processing is done.
- The context is represented in the Process Control Block or Task Control Block of the process; it includes the value of the CPU registers, the process state and memory management information etc.
- Switching the CPU to another process requires performing a state-save of the current process and a state-restore of a different process. This task is known as context switch.
- When a context switch occurs the kernel saves the context of the old process in its PCB and loads the saved context of the new process scheduled to run.
- Context-switch time is overhead, because the system does no useful work while switching.


"STATE DIAGRAM OF CONTEXT SWITCH"
PROCESS P0
OS
PROCESS P1
INTERRUPT OR SYSTEM CALL
EXECUTING
SAVE STATE INTO PCB0
IDLE
RELOAD STATE FROM PCB1
IDLE
INTERRUPT OR SYSTEM CALL
SAVE STATE INTO PCB1
IDLE
RELOAD STATE FROM PCB0
EXECUTING

Operations on Processes

❑ Process Creation:

- ✓ A process may create several new processes via a create-process system call, during the course of execution.
- ✓ The creating process is called a parent process, and the new processes are called the children of that process.
- ✓ Each of these new processes may in turn create other processes, forming a tree of processes.
- ✓ Most operating system including the Unix and the Windows family of operating systems identify processes according to a new process identifier (pid), which is typically an integer number.

Pid

- When a process creates a new process, two possibilities exist in terms of execution:

1. **Forking:**

- Forking is a method where the parent process creates a copy of itself to form the child process with different Process IDs (PIDs).
- The child process usually inherits certain characteristics from the parent process, such as memory space and open file descriptors.

2. **Spawning or Creating a new process:**

- In this method, the parent process explicitly requests the operating system to create a new process.
- Unlike forking, the new process is not a copy of the parent process, but rather a new process entirely.

❑ Process Termination:

- A process terminates when it finishes executing its final statement and ask the operating system to delete it by using exit system call.
- All the resources of the process including physical and virtual memory, files, I/O buffers are de-allocated by the operating system.

Inter process Communication

IPC

- IPC means the way in which processes communicate with each other.
- There may be numbers of processes running in our system. They may need to communicate to each other for execution.
- Based on their behavior and interaction, processes executing concurrently in the operating system may be either:
  - (i) Independent processes and
  - (ii) Cooperating processes.
- **Independent Processes:**
  - ✓ A process is independent if it cannot affect or be affected by the other processes executing in the system.
  - ✓ Independent processes are those that do not share resources or communicate with other processes.
  - ✓ Each independent process executes its task without relying on or affecting the behavior of other processes.

- ✓ These processes are typically self-contained and do not need to coordinate with other processes to complete their tasks.
- ✓ Examples of independent processes include simple utility programs or standalone applications that run without needing to interact with other processes.

❑ **Cooperating Processes:**

- ✓ Any process that does not share data with any other process is independent whereas Cooperating processes are those that share resources, communicate with each other, or coordinate their activities to achieve a common goal.
- ✓ They may exchange data, synchronize their execution, or coordinate their activities through inter-process communication (IPC) mechanisms provided by the operating system.
- ✓ Cooperating processes can work together to accomplish tasks that are too complex for a single process to handle alone.
- ✓ Examples of cooperating processes include multi-threaded applications, client-server systems, and distributed computing systems where different processes collaborate to perform tasks efficiently.

Models of Inter Process Communications

- Inter-process communication is a mechanism that allows processes to communicate with each other and synchronize their actions.
- The communication between these processes can be seen as a method of co-operation between them.
- Processes can communicate with each other through:

1. Message Passing and
2. Shared Memory

- These are the two fundamental models of inter-process communication.

GW
GATEWAY CLASSES

1. Message Passing Model:

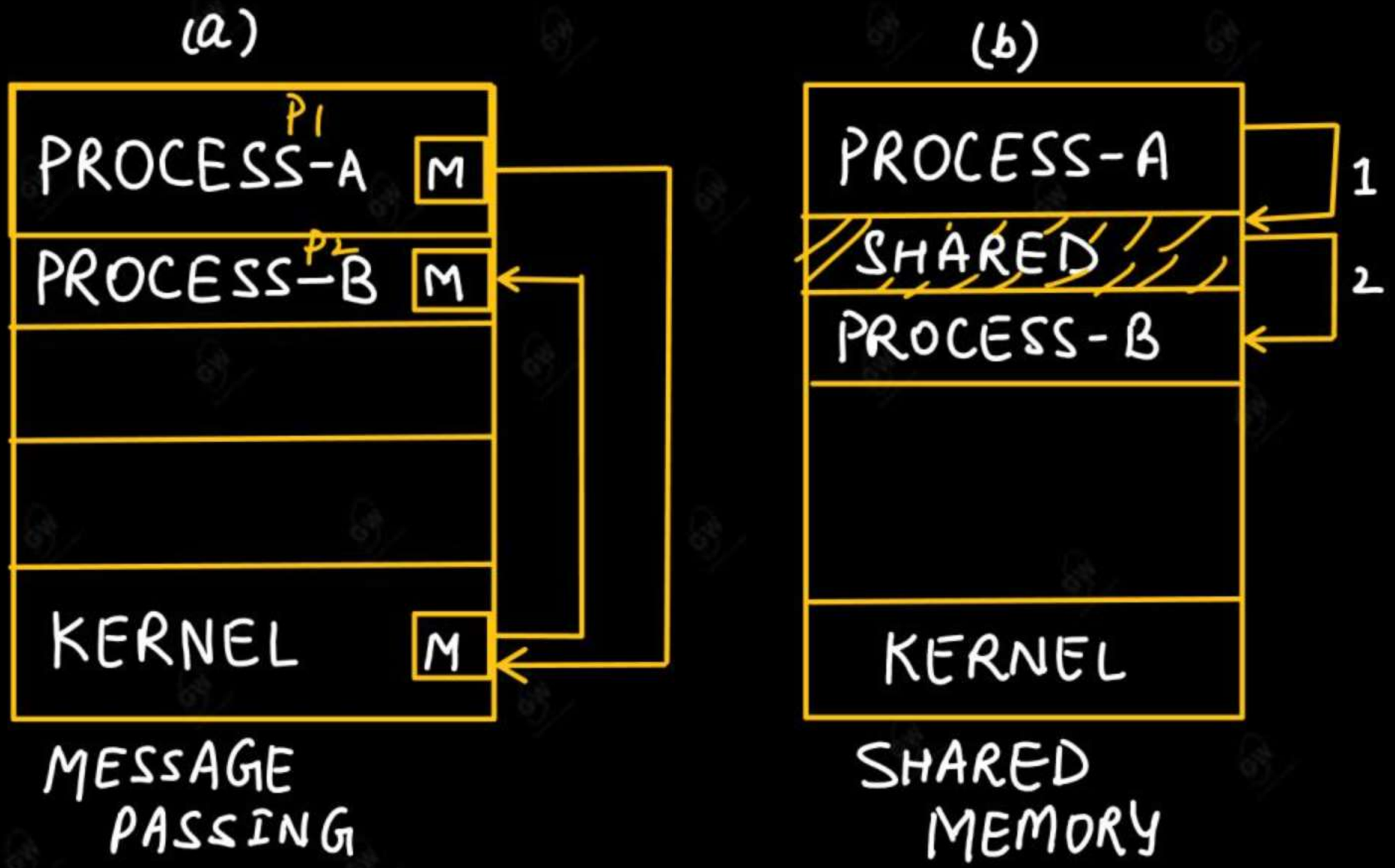
- In this model, processes communicate by sending and receiving messages.
- If two processes P1 and P2 want to communicate with each other, they proceed as follows:
  - Establish a communication link (if a link already exists, no need to establish it again).
  - Start exchanging messages using basic primitives.

  We need at least two primitives:

P1 – send (message, destination) or send (message)

P2 – receive (message, host) or receive (message)

- The sender process constructs a message and explicitly sends it to the receiving process.
- This communication can be either synchronous or asynchronous.
- In synchronous communication, the sender waits until the receiver acknowledges receipt of the message, while in asynchronous communication, the sender continues its execution without waiting for a response from the receiver.


(a)
P1
PROCESS-A
M
P2
PROCESS-B
M
KERNEL
M
MESSAGE PASSING
(b)
PROCESS-A
SHARED
PROCESS-B
KERNEL
1
2
SHARED MEMORY

2. Shared Memory Model:

- In this model, processes communicate by accessing shared memory regions.
- Processes can read from and write to these shared memory locations.
- This model allows processes to communicate by simply reading from and writing to memory locations, which can provide very efficient communication between processes running on the same system.
- However, it requires careful synchronization mechanisms to ensure that processes do not access shared memory in an inconsistent or conflicting manner.
- Techniques such as semaphores, locks, and monitors are commonly used to manage access to shared memory in a synchronized manner.
- In a shared-memory model a region of memory that is shared by cooperating processes is established.
- Processes can then exchange information by reading and writing data to the shared region.

- Shared memory is faster than message passing as message passing system are typically implemented using system calls and then require the more time consuming task of kernel intervention.
- In contrast in shared-memory systems, system calls are required only to establish shared – memory regions.
- Once shared memory is established all accesses are treated as routine memory access and no assistance from the kernel is required.

Advantages of IPC

- Enables processes to communicate with each other and share resources, leading to increased efficiency and flexibility.
- Facilitates coordination between multiple processes, leading to better overall system performance.
- Allows for the creation of distributed systems that can span multiple computers or networks.
- Can be used to implement various synchronization and communication protocols, such as semaphores, pipes, and sockets.

GW
GATEWAY CLASSES

Disadvantages of IPC

- Increases system complexity, making it harder to design, implement, and debug.
- Can introduce security vulnerabilities, as processes may be able to access or modify data belonging to other processes.
- Requires careful management of system resources, such as memory and CPU time, to ensure that IPC operations do not degrade overall system performance.
- Can lead to data inconsistencies if multiple processes try to access or modify the same data at the same time.

AKTU PYQs

Q.1. What is the use of inter process communication and context switching? 2017-18, 2 Marks

Q.2. Explain Inter-process Communication. 2017-18, 2 Marks

Q.3. Critically evaluate the method of massage passing as a means of Inter-process communication. 2017-18, 10 Marks

Q.4. Write a short note on inter-process communication. 2018-19, 2 Marks

Q.5. Explain in detail about the Inter Process Communication models and Schemes. 2021-22, 10 Marks

Download Gateway Classes Application
From Google Play store
Link in Description

Thank You