PURBANCHAL UNIVERSITY

SCHOOL OF ENGINEERING (PUSOE)

Operating System

Submitted by: **Barsha Karn**

Roll no : 13 'A'

Faculty: Computer Engineering

Batch Year : 2020

Year/ Semester: 3rd/5th

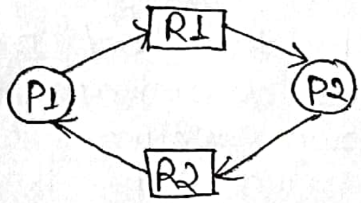
College: PUSOE, Brt

Submitted to:

Er. Deeoranjan Dongol sir

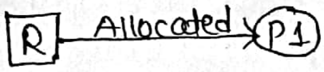
1. What is deadlock? Explain various conditions for deadlock.

→ Deadlock is a situation where two or more processes are waiting for each other. For eg - Traffic Jam : cars are all taking up space on the road while attempting to move the same direction, preventing anyone from making forward progress. In context of OS, Let us assume two process P1 and P2, each waiting for resource R2 and R1 respectively. The process depend on each other to release the resource but none taking an action, thus creating a infinite waiting and no work is done. This is called deadlock.

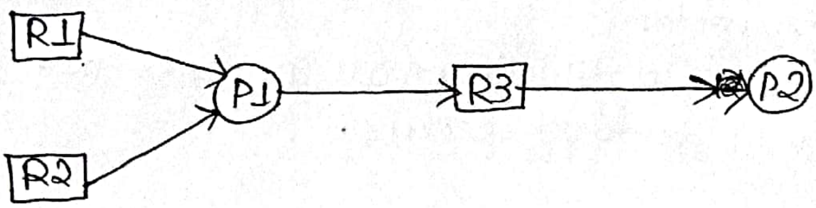


Various Conditions for deadlock are :-

i) Mutual Exclusion :- A resource can be held only by one process at a time. In other words if a process P1 is using some resources R at a particular instant of time, then some other process P2 can't hold or use the same resources R at that particular instant of time. The process P2 can make a request for that resource R but it can't use that resource simultaneously with process P1.

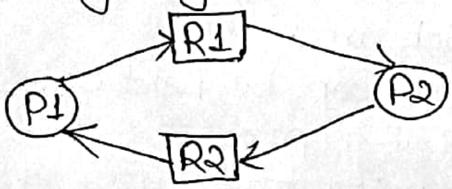


ii) Hold and wait :- A process can hold a number of resources at a time and at the same time, it can request for other resources that are being held by some other process. For eg: a process P1 can hold two resources R1 and R2 at the same time, it can request some resources R3 that is currently held by process P2.



iii) No Preemption :- A resource can't be preempted from the process by another process forcefully. For eg:- if a process P1 is using some resource R, then some other process P2 can't forcefully take that resource. If does so, then what's the need for various scheduling algorithm. The process P2 can request for the resource R and can wait for that resource to be freed by the process P1.

iv) Circular Wait :- It is a condition when the first process is waiting for the resource held by the second process, the second process is waiting for the resource held by the third process, and so on. At last, the last process is waiting for the resource held by the first process. So, every process is waiting for each other to release the resource and no one is releasing their own resource. Everyone is waiting here for getting the resource. This is called a circular wait.



---

2. Discuss the Banker's algorithm of multiple resources for avoidance of deadlock with suitable example.

⇒ Banker's algorithm is a resource allocation and deadlock avoidance algorithm used in operating systems to prevent deadlock situations in a system with multiple resources and multiple processes. Here are steps for Banker's algorithm for multiple resources :

i) Look for row, R, whose unmet resources need are all ≤ A. If no such row exists, system will eventually deadlock since no process can run to completion.

ii) Assume process of row chosen requests all resources it needs and finishes. Mark process as terminated, add all its resources to the A vector.

iii) Repeat steps (i) & (ii) until either all process are marked as terminated or until deadlock occurs.

Eg: A system has four process P1, P2, P3, P4 and three resources R1, R2 and R3 with existing resources E = (15, 9, 5). After allocating resources to all the processes available resources becomes A = (3, 2, 0).

Process↓	Allocation			Maximum			Need		
	R1	R2	R3	R1	R2	R3	R1	R2	R3
P1	3	0	1	3	2	2	0	2	1
P2	5	4	1	6	8	2	1	4	1
P3	2	2	0	3	2	4	1	0	4
P4	2	1	3	4	2	3	2	1	0

Step1- With current available resource A = (3, 2, 0), P4 can be executed. Since need of P4 ≤ A i.e. (2, 1, 0) ≤ (3, 2, 0) so P4 is executed. After complete execution of P4, it releases the resources which is allocated by it. Now total current available resources of A becomes, A = (3, 2, 0) + (2, 1, 3) = (5, 3, 3)

Step 2- With current available resources A = (5, 3, 3), P1 can be executed. Since need of P1 ≤ A i.e, (0, 2, 1) ≤ (5, 3, 3) so P1 executes. After complete execution of P1 it releases the resource which is allocated by it. Now current available resources ~~of~~ A becomes, A = (5, 3, 3) + (3, 0, 1) = (8, 3, 4)

Step 3- With current available resources A = (8, 3, 4), P3 can be executed. Since need of P3 ≤ A i.e. (1, 0, 4) ≤ (8, 3, 4) so P3 executes. After complete execution of P3 it releases the resources which is allocated by it. Now total current available resources ~~of~~ A becomes, A = (8, 3, 4) + (2, 2, 0) = (10, 5, 4).

Step 4- With current available resources A = (10, 5, 4), P2 can be executed. Since need of P2 ≤ A i.e. (1, 4, 1) ≤ (10, 5, 4) so, P2 executes. After complete execution of P2 it releases the resources which is allocated by it. Now,

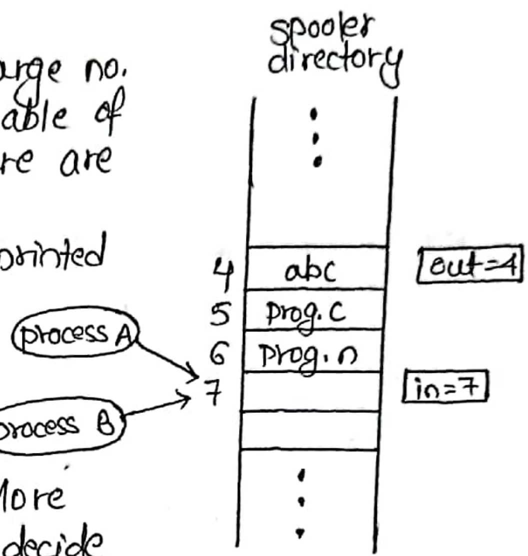
A = (10, 5, 4) + (5, 4, 1) = (15, 9, 5)

Thus, all the process runs, hence they are safe state.

Safe sequence :- P4 → P1 → P3 → P2

8. Explain race condition with four conditions to avoid race condition in detail.

→ A race condition is a concurrency-related issue in computer programming where the behaviour of a program depends on the relative timing of events or threads. It occurs when multiple threads or process access shared resources concurrently and the final outcome depends on the order of execution. This can result in unexpected or incorrect behaviour of the system.



Imagine that our spooler directory has a large no. of slots, numbered 0, 1, 2, ... each one capable of holding a file name. Also imagine that there are two shared variables.

out - which points to the next file to be printed
in - which points to the next free slot to the directory

At a certain instant, slots 0 to 3 are empty (the files that have already been printed) and slots 4 to 6 are full (with the names of files to be printed). More or less simultaneously, process A and B decide they want to queue a file for printing as shown in fig.

Process A reads in and stores the value 7, in a local variable called next-free-slot. Just then a clock interrupt occurs and the CPU decides that process A has run long enough so it switches to the process B. Process B also reads in, and also gets a 7, so it stores the name of its file in slot 7 and updates in to be an 8. Then it goes off and do other things.

Eventually, Process A runs again, starting from the place it left off last time. It looks at next-free-slot, finds a 7 there, and write its file name in slot 7, erasing the name that process B just put there. Then it computes next-free-slot +1, which is 8. The spooler directory is now internally consistent, so the printer daemon will not notice anything wrong, but process B will never receive any output. This is race condition.

Conditions to avoid race Condition :-

1. No two process may be simultaneously inside their critical region. (Mutual exclusion)
2. No process running outside its critical region may block other process.

3. No assumptions may be made about speeds or the no. of CPUs.
4. No process should have to wait forever to enter its critical region

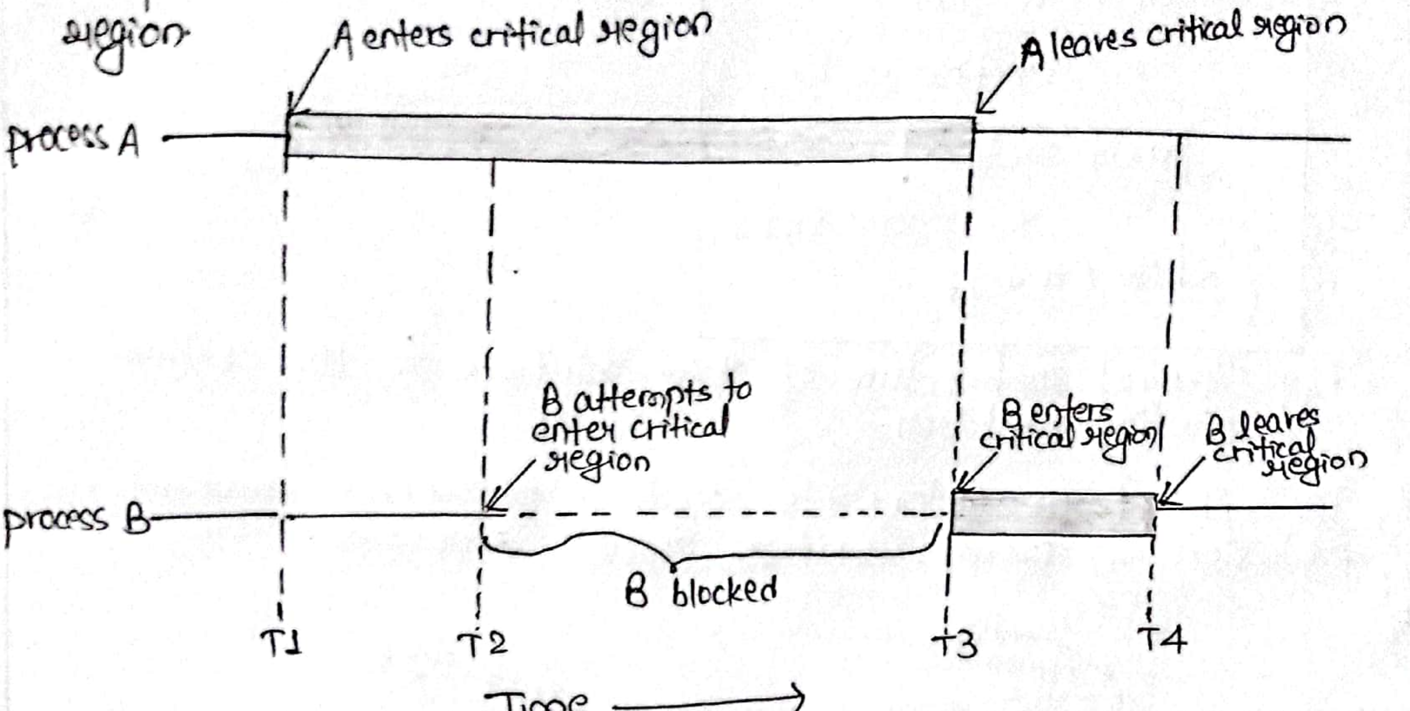


Fig: Mutual Exclusion using critical region.

---

4) Briefly explain critical section problem.

⇒ The critical section problem is one of the classic problem in OS. In OS, there are processes called cooperative processes that share and access a single resource. In these kinds of processes, the problem of synchronization occurs. The critical section problem is the problem that deals with this synchronization.

Any solution to critical section problem must satisfy the following requirements-

i) Mutual exclusion- when one or more process is executing in its critical section, no other process is allowed to execute in its critical section

ii) Progress- when no process is executing in its critical section, and there exists a process that wishes to enter its critical section, it should not have to wait indefinitely to enter it.

iii) Bounded waiting :- There must be on the no of times a process is allowed to execute in its critical section, after another process has requested to enter its critical section and before that request is accepted.

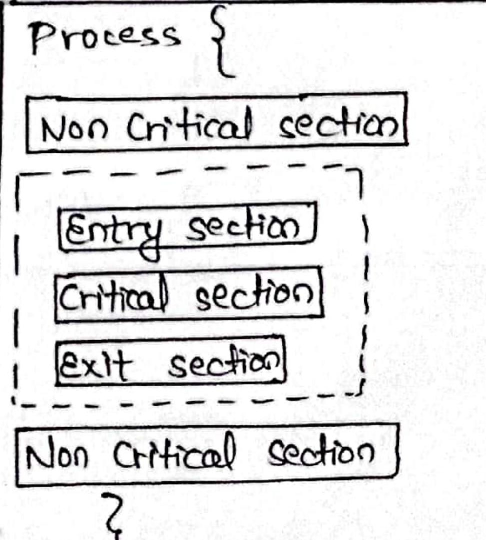


Fig: Typical structure followed by most process.

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

Fig: General architecture of these solutions to the critical section problem.

5) Describe how Peterson's solution preserves mutual exclusion in process/critical region/execution.

⇒

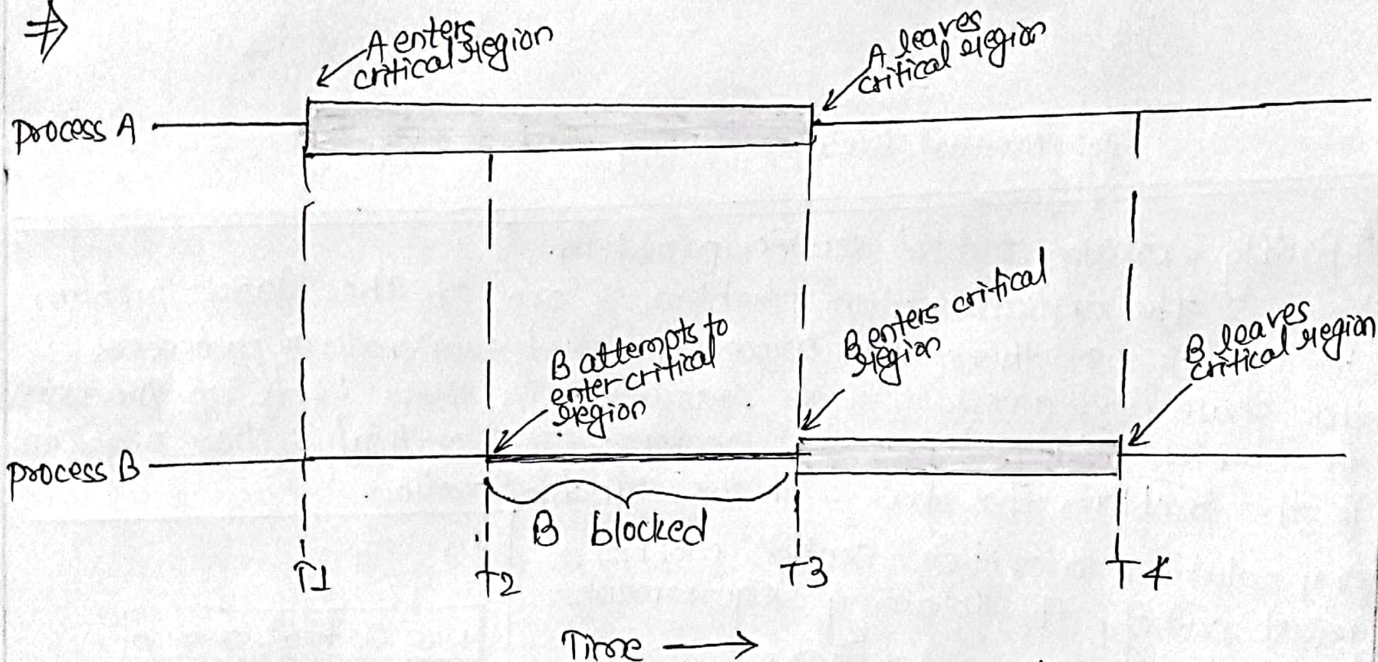


Fig: Mutual Exclusion using Critical region

5 Describe how Peterson's solution preserves mutual exclusion in process critical region execution.

⇒ The program for peterson's solution is -

```
#define N 2
# define TRUE 1
# define FALSE 0
int interested [N] = FALSE;
int turn;
void Entry_Section (int Process){
    int other;
    other = 1 - process;
    interested [process] = TRUE;
    turn = process;
    while (interested [other] == TRUE && turn = process);
}
void Exit_Section (int process)
{
    interested [Process] = FALSE;
}
```

In above code, Peterson's solution ensures mutual exclusion by using the 'turn' variable to coordinate which process can enter the critical section. If both processes try to enter simultaneously, onely one of them will succeed, as the 'turn' variable alter--nates between them.

The while loop in 'Entry_Section' prevents a process from entering the critical section if the other process is still interes--ted and it is not its turn. This waiting mechanism guara--ntees that only one process can be in the critical section at a time.

By setting and checking the 'interested' array and 'turn' variable, this code enforces mutual exclusion, allowing one process to execute its critical section before the other, ensuring that they do not interfere with each other's execution in its critical section.

⑥ Explain the use of semaphore in producer-consumer problem.

→ Code for producer-consumer problem using semaphore-

```
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;
void producer (void)
{ int item;
  while (TRUE){
    item = produce-item();
    down(&empty);
    down(&mutex);
    insert_item(item);
    up(&mutex);
    up(&full);
  }
}
void consumer(void)
{
  int item;
  while (TRUE){
    down(&full);
    down(&mutex);
    item = remove-item();
    up(&mutex);
    up(&empty);
    consume-item(item);
  }
}
```

The use of semaphore in above code is-

i) semaphore mutex = 1;

This semaphore is used as a mutex (short for mutual exclusion). It ensures that only one process (producer or consumer) can access the critical section at a time. This critical section is the block between 'down(&mutex)' and 'up(&mutex)' in both the producer and consumer ~~pro~~ functions.

ii) semaphore empty = N;

This semaphore represents the number of empty slots in a shared buffer or queue. It is used by the producer to check if there's space available in a buffer to insert a new item. The producer decreases this semaphore value using 'down(&empty)' when it inserts an item and increases it using 'up(&empty)' when it finishes.

iii) semaphore full = 0;

This semaphore represents the number of filled slots in the shared buffer or queue. It is used by the consumer to check if there are items available to consume. The consumer decreases this semaphore value using 'down(&full)' when it removes an item, and increases it using 'up(&full)' when it finishes.

The main purpose of these semaphores is to prevent race conditions and ensure that the producer-consumer do not interfere with each other when accessing the shared buffer.

---

(7) Explain classical IPC problem.

⇒ These problems are used for process synchronization.

(i) Producer-Consumer Problem (Bounded-Buffer problem)-

In this problem, two processes share a common, fixed size buffer. The producer puts information into the buffer and the consumer takes it out. Trouble arises when producer wants to put a new item in the buffer, but it is already full. The solution is the producer should go to sleep and to be awakened when the consumer has removed one or more items. Similarly, if the consumer wants to remove an item from the buffer and sees that the buffer is empty, it goes to sleep until the producer puts something in the buffer and wakes it up.

(ii) Sleeping Barber problem-

In this problem, there is a barber shop with one barber, one barber chair, and n chairs waiting for customers if there are any to sit on the chair.

- If there is no customer, then barber sleeps in his own chair.
- When a customer arrives, he has to wake up the barber.
- If there are many customers and the barber is cutting a customer's hair, then the remaining customers either wait if there are empty chairs ~~or~~ in waiting room or they leave if no chairs are empty.

The solution to this problem includes three semaphores. First is for the customer which counts no. of customers present in the waiting room. Second the barber, 0 or 1 is used to tell whether the barber is idle or working and third mutex is used to provide the mutual exclusion required for process to execute.
When the customer arrives, he acquires the mutex for entering critical region, if another customer enters thereafter, the second one will not be able to anything until the first one has released the mutex. If the chair is available then, the customer sits in the waiting room and the variable is incremented and also increases the customers semaphore. This wake up the barber if he is sleeping. At this point, customer and barber both are awake and the barber is ready to give that person a hair-cut. When the haircut is over, the customer exists the ~~produces~~ procedure and if there are no customers in the waiting room, the barber sleeps.

(iii) Dining Philoshopher Problem:

In this problem, five philoshopher sit around a circular table eating spaghett and discussing ~~phy~~ philosophy. The problem is that each philosopher needs 2 forks to eat, and there are only n forks one betwee 2 philosophers. At any time, philosopher either thinks or eats. And while eating, one require two forks. Now few problems like deadlock or starvation might arise in certain conditions.
In case, all philosophers take their left fork simultaneously right fork wont be available to anyone. Thus they have to wait for each other which might lead to condition of deadlock.
Again, if philosophers pick left fork and check the right fork, In case if one is unable to find right fork the left fork is put down. This might happen for a long duration which might lead to starvation.