Introduction to Dead locks

→ A lack of process synchronization can result in two extreme conditions are deadlock or starvation. Deadlock is the problem of multiprogrammed system.

→ Deadlock can be defined as the permanent blocking of a set of processes that either complete for system resources.

→ deadlock can occur on sharable resources such as files, printers, database, disks, tape drives, memory, cpu cycles etc.

→ A process is in deadlock state if it was waiting for particular event that will not occur. In a system deadlock, one or more processes are deadlocked.

Deadlock Example :-

→ System is collection of limited / finite no. of resources. These resources are distributed among a number of competing processes. Resources are of two types:

① Reusable resources.

② Consumable resources.

→ Reusable resource is used only by one process at a time. process can release resource after use. processors, I/O channel, I/O device, file, DB, primary & secondary memory, semaphores are example of the reusable resource.

→ Consumable resource is on that can be created & destroyed. There is no limit on the no. of Consumable resource of a particular type.

An interrupt, messages, signals and messages in I/O buffers are Example of consumable resources.

→ Process utilize the resources in the following sequence.

① Request for resource.

② use of resource.

③ Resource release.

Process uses system call for requesting the resource. After allocating resource to the process, it use or operate the resource.

→ The process, if ~~use or operate~~ Release the resource after use.

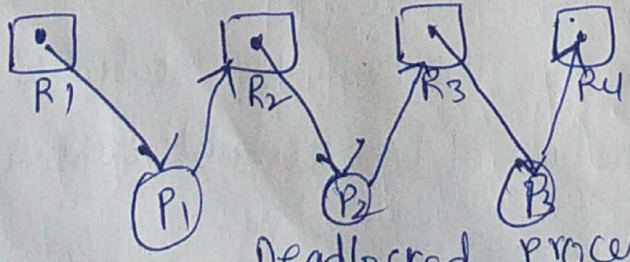
→ The resource is not available when it is requested, the requesting process is forced to wait.

Process 1	Process 2	Process 3	Process 4
Printf (" ")	int a;	Procedure ( )	Sort ( )
=	=	=	=
=	=	=	=
request (resource 1);	request (Resource 2);	request (resource 3);	request (resource 4)
/* Holding res 1 */	/* Holding res 2 */	/* Holding res 3 */	/* Holding res 4 */
=	=	=	=
Request (Resource 2)	request (Resource 3);	Request (Resource 4);	request (Resource 1)

Deadlock with 4 processes

→ Process 1 is holding resource 1 & requesting 2;
Process 2 is holding resource 2 & requesting resource 3.

→ Here deadlock occurs because none of the processes can proceed because all are waiting for a resource held by another blocked process



Deadlocked processes.

Resource Allocation Graphs :-

→ Resource allocation graph is introduced by Holt. It is a directed graph that depicts a state of the sim of resources & processes.

→ process & resource are represented by node in directed graph. Graph consist of a set of vertices (V) & set of edges (E)

fig : process.

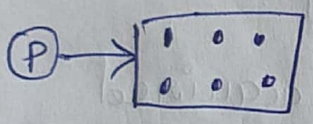
fig : Resource with 2 instances.

→ A process node is graphically represented by circle.

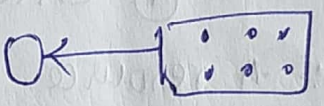
→ A resource node is graphically represented by rectangle

→ The number of bullet symbols in a resource node indicates how many units of that resource class exist in the system.

→ Claim edge $P \longrightarrow R$ indicated.



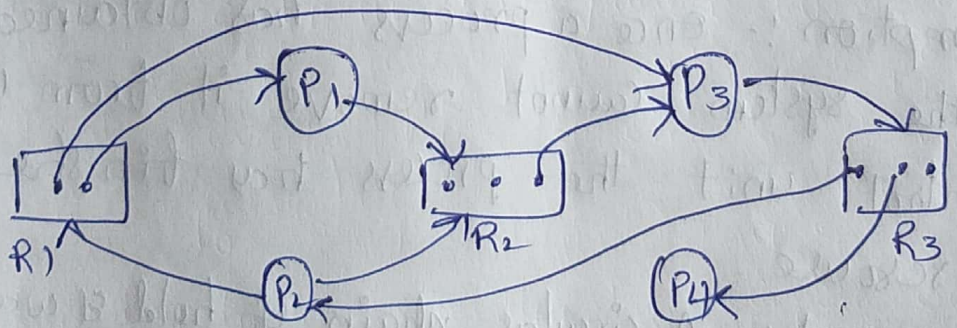
a) Request edge.



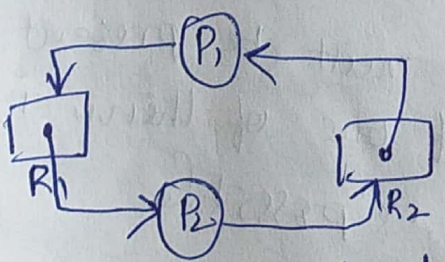
b) Claim edge.

Shows a resource allocation graph. System consist of process & resources.

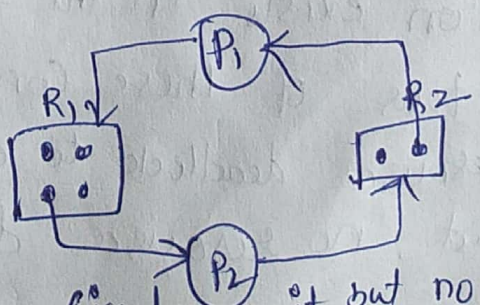
Process : P_1, P_2, P_3, P_4. Resource : R_1, R_2



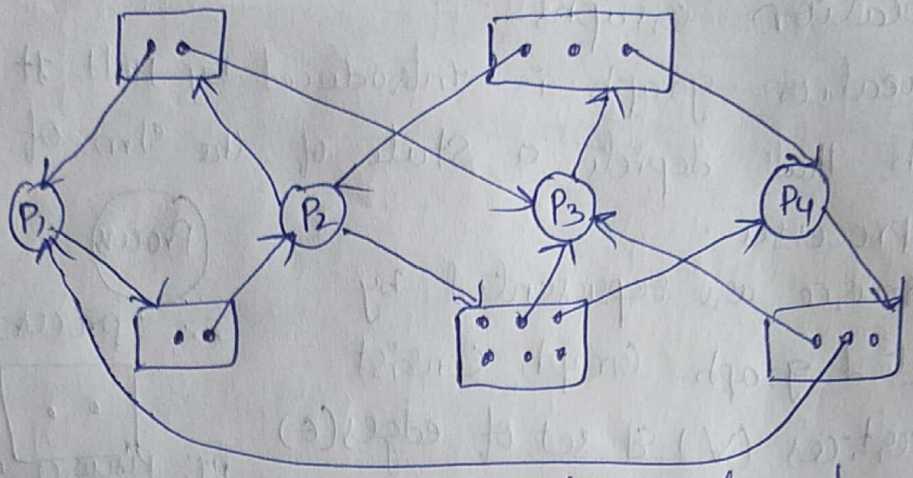
Resource allocation Graph.



Circular wait with deadlock.



Circular wait but no deadlock.



Resource allocation Graph.

Deadlock Characteristics :-

→ Necessary codition for Deadlock.

→ Following four conditions are necessary for deadlock to exist.

① Mutual exclusion ② Hold and wait

③ No preemption ④ Circular wait.

① Mutual Exclusion :- A resource may be acquired exclusively by only one process at a time.

② Hold and wait :- processes currently holding resources that were granted earlier can request new resources.

③ No preemption :- Once a process has obtained a resource, the system cannot remove it from the process control unit the process has finished using the resource.

④ Circular wait :- A circular chain of hold & wait condition exists in the S/m.

→ All four of these conditions must be present for a resource deadlock to occur. If one of them is absent, no resource deadlock is possible.

Deadlock Solution

There are four approaches for deadlock solution

① Deadlock prevention
② Deadlock avoidance
③ Deadlock detection
④ Deadlock recovery.

→ In deadlock prevention, aim is to condition a system to remove any possibility of deadlock occurring. poor resource utilization may be possible.

→ Avoidance: Deadlocks can be avoided by clearly identifying safe states and unsafe states.

→ Detection: Deadlock detection methods are used in systems in which deadlocks can occur.

→ Recovery: used to resolve the deadlock from a system.

→ deadlock prevention and detection algorithm is used for ignoring the deadlock.

① Deadlock prevention :-

→ To prevent a deadlock, the OS must eliminate one of the four necessary conditions.

① mutual exclusion ② Hold and wait.
③ No preemption ④ Circular wait.

① mutual exclusion: It is necessary in any computer system because some resources (memory, CPU) must be exclusively allocated to one uses at a time. No other process can use a resource while it is allocated to a process.

② Hold and wait :- If a process holding certain resources is denied a further request, it must release its original resources & if required request them again.

③ No preemption :- It could by bypassed by allowing the operating system to deallocate resources from process.

④ Circular wait : Circular wait can be bypassed if the operating system prevents the formation of a circle.

→ A deadlock is possible only if all four of these conditions simultaneously hold in the system.

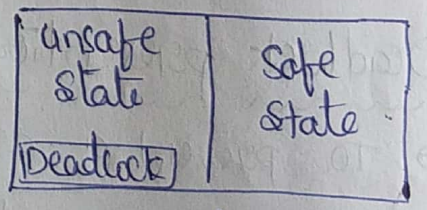
→ prevention strategies ensure that atleast on the conditions is always false.

② Deadlock Avoidance :-

→ Deadlock avoidance depends on additional information about the long term resource needs of each process.

→ The system must be able to decide whether granting a resource is safe or not & only make the allocation when it is safe.

→ when a process is created, it must declare its maximum claim, ie. the maximum no. of unit Resource.



Safe and unsafe state.

→ The resource manager can grant the request if the resources are available.

Bankers Alg is the deadlock avoidance algorithms.

→ Algorithm is checked to see if granting the request leads to an unsafe state. If it does, the request is denied.

Deadlock Avoidance

→ Simplest & most useful model requires that each process declare max no. of resource that it may need.

→ Deadlock avoidance alg dynamically examines the resource allocation can never be a circular condition.

→ If a system is in safe state there is no deadlock.

Avoidance: Ensure that a system will never enter a unsafe state.

Eg: Deadlock Avoidance by using Banker's Algorithm.

→ when a process gets all its resources then the job is over of that process it must be return in a finite amount of time.

Remaining

Process	Allocation A	B	C	maximum A	B	C	Available Current work	(Max - Allocation) Remaining need A	B	C
P_0	0	1	0	7	5	3	3 3 2 (10 → 3)	7	4	3
P_1	2	0	0	3	2	2	5 3 2	(7-0) 1	(5-1) 2	(3-0) 2
P_2	3	0	2	9	0	2	7 4 3	6	0	0
P_3	2	1	1	4	2	2	7 4 5	2	1	1
P_4	0	0	2	5	3	3	7 5 5	5	3	1

Add 7 2 5

Total Allocation A=10, B=5, C=7.

~~10 5 7 - current work available~~

< P_1, P_3, P_4, P_0, P_2 >

↳ Safe sequence.

→ In this way the process will exites.

Remaining need = maximum − Allocation

Current work = Total − Allocation) we will get current work

→ we have to calculate the sequence.
⇒ we need to find out the safe state process.
→ so, that safe state process is assigned first.
→ we need to calculate the next safe state process.
→ now here we are ~~to~~ using Bankers algorithm.

need i ≤ work (~~remai~~ i
(remaining need) (current work.

work = work + allocation (we need to update the work).

→ If the condition is satisfys we have to arrange in that sequence.
→ The resources should be allocated to remaining process.
→ For Example .P_0 - process uses 3 3 2 resources after completion of P_0 those resources are allocated to P_1 process. that P_1 process may use the that resources.

P_0 - need is 7 4 3. (i ≤ work)

= 743 ≤ 332

we need to check each individual value. not total value. ⇒ 743 ≤ 332 — x (unsatisfy) not in safe state

→ If ~~P0~~

means P_0 resource are not suffient to execute or complete the P_0 work.

P_1 — 122 ≤ 332 — ✓ (satisfy)

w ⇒ 332 + 200

= 532

now update with 332

now P_2 600 ≤ 532 × (unsatisfy).

P_3 → 211 ≤ 532 ✓ (satisfy

w = 532 + 211

⇒ 743

P_4 → 531 ≤ 743 ✓ (satisfy)

work = 743 + 002

= 745

now update the work with current work

→ Now again come to P_0, when the work is over of all the resources of that particular process are allocated to anothes processes.

→ The resources order is in increasing order, so there may be availability to execute previous processes.

→ now P_0 → 743 ≤ 745 ✓ (satisfy)

⇒ w = 745 + 010

⇒ 755

⇒ update the current work.

→ now P_2 → 600 ≤ 755 (satisfy)

w = 755 + 302

= 1057.

Dead Lock Detection :- Deadlock detection is the process of determining that a deadlock exists and identifying the processes & resources involved in the deadlock.

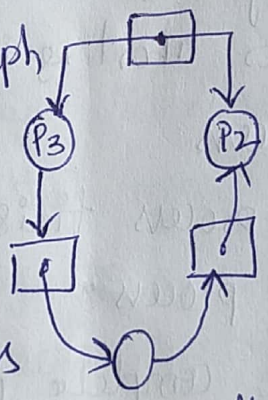
→ If processes are blocked on resources for an inordinately long time, the detection algorithm is executed to determine whether the current state is a deadlock.

wait for Graph :-

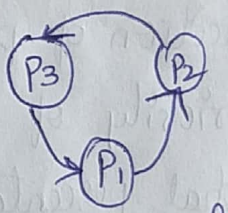
→ Any resources allocation graph with a single copy of resources can be transferred to a wait for Graph.

→ Fig shows resource allocation graph with corresponding wait for graph. The state of the system can be modeled by directed graph, called a wait for graph.

→ wait for Graph is a graph where each node represents a process. An edge $P_i \rightarrow P_j$ means that process P_i is blocked & waiting for process P_j to release a resource.



(a) Resource allocation Graph



(b) wait for Graph.

→ The wait for graph of a sim is always smaller than the resource allocation graph of that same system.

→ There is a deadlock in a system if and only if there is a loop in the wait for graph of that system.

→ Deadlock detection involves two issues:

① maintenance of the wait for graph

② searching of the wait for graph for the presence of cycles.

→ For multiple instances of a resource type use an alg similar to Banker's Alg

→ Deadlock detection requires overhead for run time cost of maintaining necessary information & executing the detection Alg.

Deadlock Recovery

→ Once deadlock has been detected in the sys, the deadlock must be broken by removing one or more of the four necessary conditions.

→ Here one or more processes will have to be preempted, thus releasing their resources so that the other deadlocked processes can become unblocked.

① process termination :-

→ Deadlock is removed by aborting a process. But Aborting process is not easy. All deadlocked processes are aborted.

→ Circular wait is ~~aborted~~ eliminated by aborting one by one process. There will be lot of overhead.

→ Deadlock detection alg must rerun after each process kill.

→ Selection of process for aborting is difficult. Parameters are -

① Priority of the process

② what percentage the process finished its execution?

③ Resource is used by process.

④ Need of resources to complete process remaining operation.

⑤ How many processes will need to be terminated.

⑥ process type : Batch or interactive

⑦ Resource preemption :-

→ Some times, resource temporarily take away from its current process & allocate it to another process.

→ for selecting victim, following factors are considered.

① priority of the process, Higher priority process are usally not selected

② cpu time is used by process. The process which is close to completion are usually not selected.

(c) The number of other process that would be affected if this process were selected as the victim.

(3) Recovery through rollback :-

(1) when a process in a sln terminates, the sln performs a rollback by undoing every operation related to the terminated process.

(2) checkpointing a process means that its state is written to a file so that it can be restarted later.

(3) risk in this method is that the original deadlock may recover by the non determinancy of concurrent processing may ensure that this does not happen.

(4) Starvation :

→ Starvation is one type situation in which a process waits for an event that might never occur in the sln.

→ select the victim only for finite number of time. use rollback method for selecting victim process.

comparision blw Detection, prevention & avoidance methods

Parameters	Avoidance	Detection	prevention.
(1) Resource Allocation policy	midway blw that of detection & prevention	very liberal	Conservative undes commit resource.
(2) Different Schemes	manipulate to find at least one safe path	invoke periodically to test for deadlock	preemption, resource ordering, requesting all resources at once.
(3) Advantages	No preemption necessary	Never delays process initiation.	No preemption necessary.
(4) Dis-advantages	Process can be blocked for long period	Inherent preemption losses	delays process initiation.

Process synchronization :-

⇒ logical control flows are concurrent if they overlap in time. this is known as concurrency.

→ Concurrency refers to any form of interaction among processes or threads.

Ex:- Hlw exception handlers, processes & unix signal handlers.

→ Concurrency is good for users because working on the same problem, simultaneous execution of programs.

→ Concurrency means that two or more calculations happen with in the same time & there is usually some sort of dependency blw them. parallelism means popular solution is interleaved processing.

→ But Concurrency describe a problem, while parallelism describe a solution.

Principle of concurrency :-

* concurrent access to share data may result in data inconsistency. maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.

Following are the example of concurrency in different types of os.

① Concurrency in multiprograming :- An interaction between multiple processes running on cpu.

② Concurrency in multithreading :- An interaction blw multiple threads running in one process.

③ Concurrency in multiprocessors: An interaction blw multiple cpu's running multiple processes or threads.

④ Concurrency in multi-computers: An interaction between multiple computers running distributed processes or threads.

→ Java is concurrent programing language.

Process synchronization means sharing sim resources by processes in a such a way that, concurrent access to shared data is handled there by minimizing the chance of in-consistent data.

→ It is required in uni-processor sim, multiprocessor sim, Network.

Race Condition:-

→ Race condition occurs when two or more operations occur in an undefined manner.

when two or more processes are reading or writing some shared data & the final result depends on who runs precisely. when, are called Race condition.

→ there is a "race condition" if the cpu outcome depends on the order of the execution. the out come depends on the cpu scheduling or "interleaving" of the threads.

Ex:- Bank Balane → Deposite

Bank Balame → Trancefer.

→ The concurrent execution of two processes is not guaranteed to be determinate, since different executions of the same program on the same data may not produce the same result.

Operating Sln concerns:-

Design & management issue for concurrency are as follows

① Track of various processes is kept by OS.

② OS allocates & deallocate slw & hlw resources to active process

③ Process execution speed do not depends upon other process execution speed.

④ OS protect user data & physical resources from un-authorized process.

Critical section problem:-

→ A critical section is a block of code that only one process at a time can execute so, when one process is in its critical section, no other process may be in its critical section.

→ The critical section problem is to ensure that only one process at a time is allowed to be operating in its critical section.

* Critical section means process may change some common variable, writing files, updating memory location, updating a process table etc.

→ When process is accessing shared modifiable data, it is said to be in a C.S.

→ Each process take permission from OS to enter into the critical section. structure as follows.

① Entry section

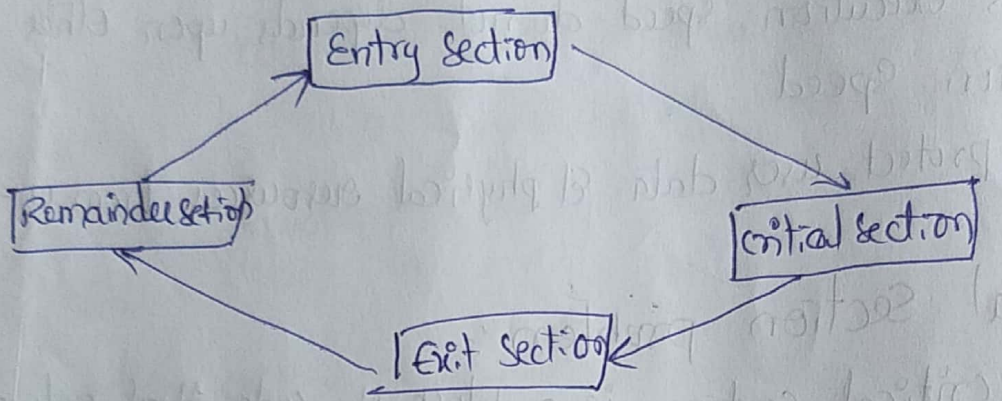
② Remainder section

③ Exit section.

① Entry section: It is a block of code executed in preparation for entering critical section

② Exit section: The code executed upon leaving the CS

③ Remainder section: Rest of the code is remainder section.



critical section.

* Each process cycles through remainder, entry, critical, exit sections. In this order.

Ex:- Process 1 int count = 5 Process 2

Count = count + 1 Count = count - 1

O/P: Count = ? (6 or) 5.

General framework :- Every solution will have the following layout

```
Entry section
critical section
Exit Section
Remainder section
```

→ To prevent critical section problem, the system should ensure that only one process at a time can execute the instruction in its critical section for a particular resource.

(15) Solution of critical section:-

→ solution to critical section to problem must satisfy the mutual exclusion, progress, & bounded waiting parameters.

→ Hlw & Slw solution to critical section problem is available.

→ process using kernal is active in the Slm at any given time.

Race condition is with implementing a kernal code.

→ Kernal data structure maintains a list of all open files in the Slm.

→ OS handles critical section problem by using kernal.

Kernal classified into ① preemptive kernal.

② Non-preemptive kernal.

Requirements of mutual exclusion:-

① At any time, only one process is allowed to enter in its critical section.

② Solution is implemented purely in Slw on a Machine.

③ A process remains inside its CS for a bounded time only.

④ A process must not be indefinitely postponed from entering its critical section.

⑤ A process cannot prevent any other process for entering into critical section.

⑥ Critical Region

→ Critical Region is an area of a process which is sensitive to inter-process communications. To guarantee mutual exclusion only one process is allowed to enter its critical Region at a time.

Mutual Exclusion :-

→ The need for mutual exclusion comes with concurrency. There are several kinds of concurrent exclusion.

1. Interrupt handlers
2. Interleaved preemptively scheduled threads.
3. multiprocessor clusters.
4. Distributed Systems.

mutual exclusion methods are used in concurrent programming to avoid the simultaneous use of a common resource, such as a global variable, by pieces of computer code called critical sections.

Approaches to implementing mutual exclusion :-

1. Software Method
2. Hardware Method
3. Programing language method.

Requirements of mutual Exclusion :-

1. At any time, only one process is allowed to enter in its CS.
2. Sln is implemented purely in Slw on a machine
3. A process remains inside its CS for a bounded time only
4. A process cant prevent any other process for entering into critical section.

(16) **Lock :-**

→ Lock is shared variable. Software mutual exclusion required that a process read a variable to determine that no other process is executing a CS, then set a variable know as lockvar

→ It indicates that the process is executing its CS.

→ Initially lock variable is initialized with zero. The process set it to 1 & enters its critical section.

lock = 0 No process in the CS.

lock = 1 Process in the CS.

→ Using simple lock variable, process synchronization problem is not solved. To avoid this, spinlock is used.

→ A lock that uses busy waiting is called a spin lock.

Mutex :-

→ mutex is used to ensure only one thread at a time can access the resource protected by the mutex.

→ The process that locks the mutex must be the one that unlocks it.

→ Mutex is good only for managing mutual exclusion to some shared resource.

→ Mutex is easy & efficient for implement.

→ Mutex can be in one of two states ① locked ② unlocked.

Peterson's solution :- Peterson's solution is sw based solution for critical section problem.

→ This algorithm will not work correctly on the modern computer architecture

→ It is simple algorithm for two process mutual exclusion with busy waiting.

→ This algorithm less complicated than dekker's Alg.

→ peterson's algorithm solves critical section problem of two processes only. It does not require any special hardware.

→ Two processes alternates execution between their critical section & remainder sections.

→ deadlock & indefinite postponement are impossible in peterson's solution as long as no process or thread terminates unexpectedly.

Synchronization Hardware :-

Test & Set Operations.

→ Test & set is a single individual machine instruction. It is simply known as TS. It is introduced by IBM.

→ In a one machine cycle, it tests to see if the key is available & , if it is , sets it to unavailable. TS are a hardware solution for critical section problem.

→ TS instruction eliminates the possibility of preemption occuring during the interval. The instruction : TestAndset(p,q).

→ The instruction reads the value of q , which may be either true or false. Then the value is copied into 'p' & the instruction sets the value of 'q' to true.

(13) * Process P_i would test the condition code using TS instruction before entering a critical section. If no other process was in this CS, then process P_i would be allowed to proceed & condition code would be changed from zero to one.

→ when process P_i exist the CS, the condition code is reset to 'zero' so another process can enter into CS.

→ If procss P_i finds a busy condition code then it is placed in a waiting loop. where it continues to test the condition code & waits until it is free.

Advantages:- Simple to implement. It can be used to support multiple CS. It works well for a small no. of processes.

Drawbacks:- It suffers from starvation, there may be deadlock. There is possibility of busy waiting.

Semaphores :-

* Semaphore is described by dijkstra. Semaphore is a non-negative integer variable that is used as a flag.

→ Semaphore is an OS abstract data type. It takes only integer value. Its used to solve critical section problem.

→ Dijkistra introduced two operations (P & V) to operate on semaphore to solve process synchronization problem.

→ A process calls the 'P' operation when it wants to enter its CS & calls 'V' operation when it wants to exit its CS.

→ P operation is also called as 'wait' operation.

→ V " " " " "signal" "

→ A wait operation on a semaphore decease its value by one.

wait: while S<0
do loops;
S:=S-1

→ A signal operation increments its value.

signal: S:=S+1;

→ A proper semaphore implementation requires that P & V be invisible operations. A semaphore operation is atomic.

→ There is no guarantee that no two processes can execute wait & signal operations on the same semaphore at the same time.

Binary semephore:-

* Binary semaphore is also known as mutex locks. It deals with the critical section for multiple processes.

+ Binary semaphore value is only blw 0 & 1.

Counting semaphore:-

→ used with that resource which has a finite no. of instances.

→ It works over unrestricted domain.

→ nonbinary semaphore often referred to as either a counting semaphore or general semaphore.

→ Process waiting for semaphore is stored in the queue for binary & counting semaphore. Queue uses FIFO policy.

Busy waiting:-

* Busy waiting is a situation in which a process is blocked on a resource but does not yield the processor. A Busy wait keeps the CPU busy in executing a process even as the process does nothing.

→ Busy waiting is also called as spin waiting.

(18) Properties of semaphores:

→ Semaphores are machine independent
→ Semaphores are simple to implement.
→ Correctness is easy to determine.
→ Can have many different critical sections with different semaphores.
→ Semaphore acquire many resources simultaneously.

Drawbacks of semaphore:

→ They are essentially shared global variables.
→ Access to semaphores can come from anywhere in a program.
→ There is no control or guarantee of proper usage.
→ There is no ligistic connection b/w the semaphore & the data to which the semaphore controls access.
→ They serve two purposes, mutual exclusion & scheduling constraints.

Semaphore programming.

① semget() :- To create a semaphore or gain access to one that exists, the ~~segment~~ semget system call is used.
→ The segment system call takes three arguments.
① The first argument, key, is used by the system to generate a unique semaphore identifier.
② The second argument nsems, is the no. of semaphores in the set.
③ The third argument semflag, is used to specify access permission and/or special creation condition.
→ If the segment system call fail, it returns -1 & sets the value stored in errno.
→ when a semaphore is first created, the kernal assigns to it an associated

a) Semaphore control Block (SCB)
b) A unique ID
c) A value (binary or a count)
d) A task-waiting list.

→ A kernal can support many different types of semaphores, including binary, counting & mutex semaphore.

Semctl() function :-

→ The Semctl system call allows the user to perform a variety of generalized control operations on the SIm Semaphore structure, on the semaphores as a set, & on individual semaphores.

```
# include <sys/types.h>
# include <sys/ipc.h>
# include <sys/sem.h>
int SemCtl (int semid, int semnum, int cmd, union semnumarg)
```

→ The semctl system call takes four arguments.

① The first argument semid is a valid semaphore identifier that was returned by a previous semget SIm call.

② The second argument to semnum, is the no. of semaphores in the semaphore set

③ The third argument to semctl, cmd, is an integer command value. The cmd value directs semctl to take one of several control actions. Each action requires specific access permissions to the semaphore control structure.

④ The fourth argument to semctl arg, is a union of type semun. Given the action specified by the preceding cmd argument, the data in arg can be one of any of the following four values.

(19) a. An integer already was set in the val member of semun union that used with SETVAL to indicate a change.
b. A reference to a semid-ds structure where information is returned when IPC_STAT.
c. A reference to an array of type unsigned short integers. the array is used either to initialize the semaphore set specifying GETALL
d. seminfo structure when IPC_INFO is required
→ If semctl fails, it returns -1 & sets errno to indicate the specific error.

Semop() function :-

* Additional operations on individual semaphores are implemented by using the semop system call. Its syntax is
```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys.sem.h>
int semop (int semid, struct sembuf *sops, unsigned nsops);
```
* The semop system call takes 3 arguments.
① The first argument, semid is a valid semaphore identifier that was returned by a previous successful semget system call.
② The second argument, sops, is a reference to the base address of an array of semaphore operations that will be performed on the semaphore set associated with by semid value.
③ the third argument, nsops, is the no. of elements in the array of semaphore operations.

If semop fails, it returns a value of -1 & sets errno to indicate the specific semaphore operations.

→ If a semop call must block after completing some of its component operations, the kernal rewinds the operation to the beginning to ensure atomicity of the entire call.

→ when semop value is negative, the process specifying the operation is attempting to decrement the semaphore.

→ when semop value is positive, the process is adding to the ~~college~~ semaphore value.

→ Again, when semaphore value is to be modified, the accessing process must have alter permission for the semaphore set.

→ when it zero, the process is testing the semaphore to determine if it is at 0.

Sem-post():-

Syntax:- #include <semaphore.h>

int sem-post (sem-t *sem);

→ The sem-post() function unlocks the specified semaphore by performing a semaphore unlock operation on that semaphore. when this operation results in positive semaphore value, no threads were blocked waiting for the semaphore to be unlocked; the value is simply incremented.

→ When this operation results in a semaphore value of zero, one of the threads waiting for the semaphore is allowed to return successfully from its invocation of the sem-wait function.

Sem_init():

Syntax:- #include <semaphore.h>

int sem_init (sem_t *sem, int pshared, unsigned int value);

* The sem_init() function is used to initialize the semaphore's value. The pshared argument must be '0' for semaphores local to a process. The value argument specifies the initial value for the semaphore.

→ The pshared argument indicated whether this semaphore is to be shared blw the threads of a process. or blw process.

* If pshared has the value 0, then the semaphore is shard blw the threads of a process, & should be located at some address that is visible to all threads.

* If pshared is non zero then the semaphore is shared blw processes. & should be located in a region of shared memory.

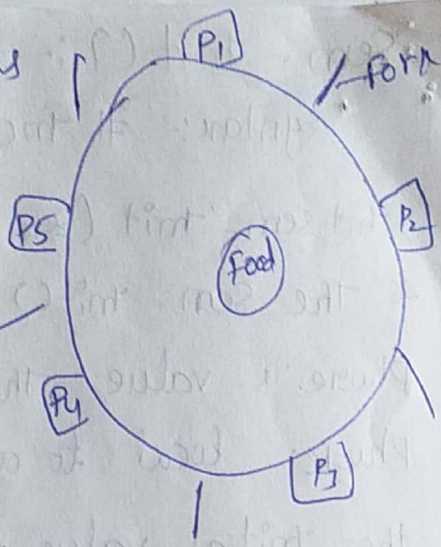
Classical problem of synchronization.

① Dining philosopher problem :-

* Dining philosphers problem is one of classical process synchronization problem. Here five philosphers are seated around a circular table. They spend their lives in thinking & eating. Five plates with five forks are kept on the circular table.

* while philosphers think, they ignore the eating & do not require fork. when a philospher decides to eat, then he/she must obtain a two fork, one from left side & another from right side fork.

* After consuming a food, the philosphers replace the fork & resumes thinking. A philosphers to the left or right of a dinning philosphers cannot eat while the dining philosphers is eating, since forks are a shared resource.



Code:

```
void dinning – philosphers ()
{
while (true)
{
Thinking ();
eating ();
}}
void eating ()
{
  take left fork ();
        Take right fork (i);
        eat food (delay);
        put right fork ();
        put left fork ();
        }
```

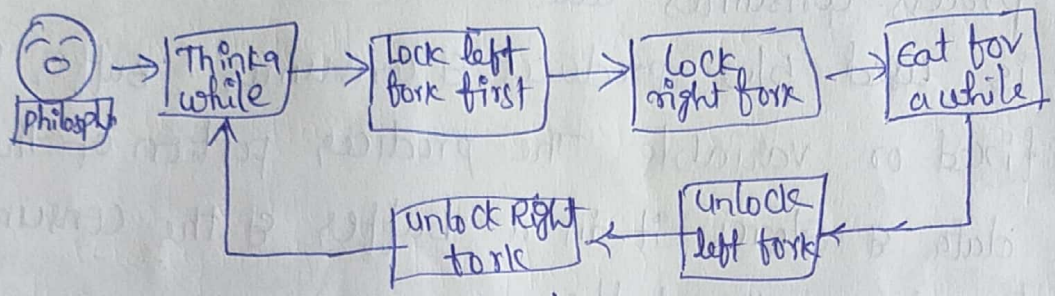
→ Here philosphers operates asynchronously & concurrently. It is possible for each philosphers to enter into eating mode.

Problem Analysis :-

Shared resource :- Here each fork is shared by two philosphers so we called it is a shared resource.

Race condition :- we do not want a philosphers to pick up a fork that has already been picked up by his neighbor.

21. Solution:- we consider each fork as a shared item & protected by a mutex lock. Before eating, each philospher first lock left fork & then right fork. then philospher have two locks so he can eat a food.



philospher states.

Reader-writter problem:-

→ when two types of processes need to access a shared resource such as a file or database. They called these Procedures reader & writers.

→ For example in railway reservation s/m, readers are those who want train information. because readers do not change the content of db. many readers may access the db at once. no need to enforce mutual exclusion.

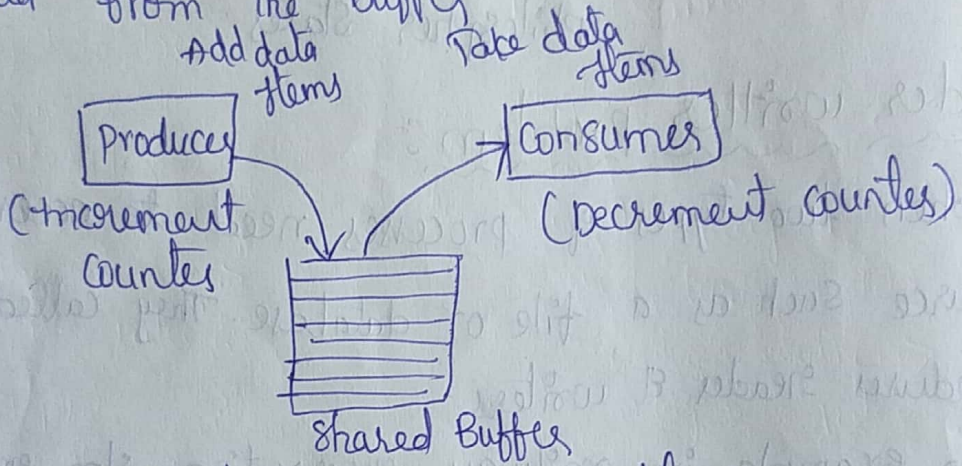
→ The writters are those who making reservations on a particular train. can modify the database, so it must have exclussive access. when writer is active, no other readers or writers may be active. It must enforce mutual exclusion.

→ If Reader having higher priority than the writes, then there will be starvation with writers. for writer having higher priority than reader then starvation with readers.

The producer consumer problem :-

→ producer-consumer problem is example classic problems of synchronization producer process produce data item that consumer process consumes later.

* Buffer is used b/w producer & consumer. Buffer size may be fixed or variable. The producer portion of the apln generates data & stores it in a buffer. & the consumer reads data from the buffer.



Producer-consumer problem.

→ producer consumer is also called bounded buffer problem

⇒ The producer consumer problem shows the need for synchronization in Slm where many processes share a resource.

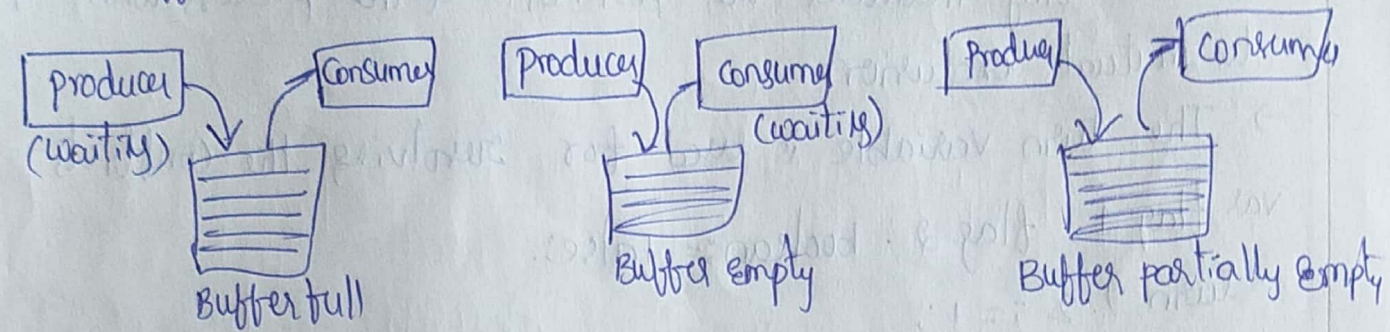
→ Problem arises when the buffer is full & producer wants to add a new data item. Solution to this is that producer goes to sleep until consumer removes data item from the buffer. Similar also to exist with consumer.

→ In order to synchronize these processes, both producer & consumer are blocked on some condition.

Example for producer consumer problem :-

printing word file.

22. Producer - consumes problem is solved by using semaphore, mutex and monitor. Mutual exclusion must be enforced on Buffer itself.



Code for Producer

```
Producer(void)
{
  int item;
  while (TRUE)
  {
  Produce_item(&item);
  Produce - if (counter==N)
  sleep();
  enter_item(item);
  counter = counter + 1;
    if (counter == 1)
    wakeup(consumer);
  }
```

Code for Consumer

```
Consumer(void)
{
  int item;
    while (TRUE)
    {
  if (count == 0)
  sleep()
remove_item(&item);
Counter = counter - 1;
if (count == N-1)
wakeup(producer);
Consume_item(item);
  }.
```

Dekkar's Solution :-

→ Dekkar's solution was the first correct solution to the critical section problem for two processes / Threads. It uses an array of Boolean values & and an integer variable.

→ Dekkars algorithm is purely software solution with no special purpose h/w instruction. It uses flag to indicate a process desire to enter its critical section.

* processes are loops indefinitely, repeating entering & reentering its critical section. Here flag & turn are two global variables. variable flag indicates the position of the process with respect to mutual exclusion.

→ The turn variable is used for resolving the conflicts.

```
var flag 1, flag 2 : boolean := false;
turn 1.2 := 1;
```

```
P1 : begin
---
flag 1 := True
turn := 2
while flag 2
and turn = 2 do;
<critical section>
flag 1 := false;
end P1.
```

```
P2 : begin
---
flag 2 := true;
turn := 1;
while flag 1
and turn = 1 do;
<critical section>
flag 2 := false;
end P2.
```

→ Dekkars algorithm is correct & its satisfies mutual exclusion. & it is free from deadlock & starvation.

→ The process P_1 indicates its desire to enter its critical section by setting its flag to true. The process then proceeds to the while test & determine whether P_2 also wants to enter its critical section.

Limitations

→ It does not provide strict alternation.

→ It will not work with many modern CPU's

→ It won't work on Symmetric multi-processors (SMP) CPU's.

Drawbacks of slw solutions

* There is possibility of busy waiting
* Programming part is complicated.
* Makes difficult assumptions about the memory s/m.

Monitors :-

* Monitors is an object that contains both data & procedures needed to perform allocation of a shared resource.

→ Monitor is implemented in programming languages like pascal, Java & C++.

→ Monitor is an abstract data type for which only one process may be executing a procedure at any given time.

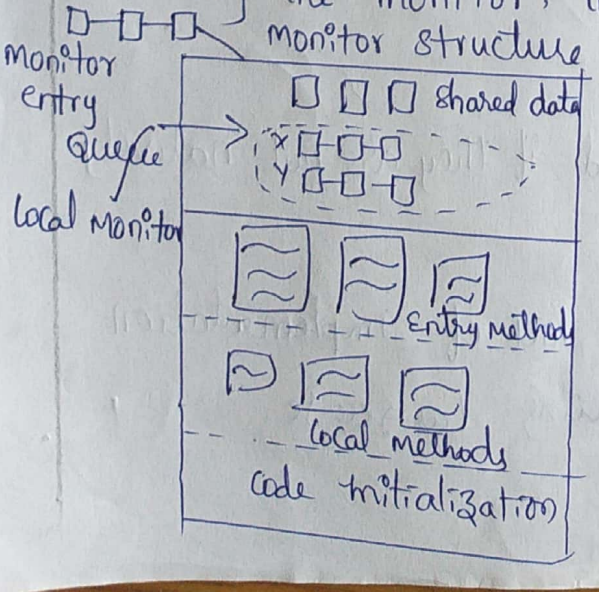
→ Monitor is a collection of procedure, variables and data structure.

→ Data inside the monitor may be either global to all routines within the monitor or local to a specific routine.

→ If the data in monitor represents some resource, then the monitor provides a mutual exclusion facility for accessing the resource.

→ when a process calls a monitor procedure, the first few instruction of the procedure will check to see if any other process is currently active within the monitor.

→ If process is active then calling process will be suspended until the other process has left the monitor. If no other process is using the monitor, the calling process may enter



→ Monitor supports synchronization by the use of condition variables that are contained within the monitor & accessible only within the monitor. Every conditional variables has associated queue.

cwait (condition variable)

csignal (condition variable.

cwait :- It suspended execution of the calling process on condition.

csignal: Resume execution of some process blocked after a cwait on the same condition.

The cwait must come before the c signal.

→ CPU is a resource that must be shared by all processes. The part of the kernal that apportions CPU time b/w processes is called the Scheduler.

→ A condition variables is like a semaphore, with two differences.

① A semaphore counts the no. of excess up operations, but a signal operation on a condition variable has no effect until some P is waiting

② A wait on a condition variable automatically does an up on the monitor mutex & blocks the caller.

Drawbacks of Monitors:-

* There is possibility of deadlocks in the case of nested monitor calls.
* monitor cannot easily be added if they are not natively supported by the language.
* monitor access concept if its lack of implementation most commonly used programming language.