Concurrent Programming

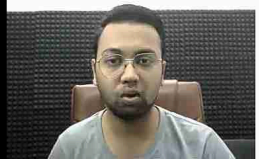
09 May 2021 09:54

Processes can Execute Parallelly

Processes are independent of each other

Anyone can start first.

S1: a = b + c;
S2: d = e + f;
S3: g = a / d;
S4: f = g x I;



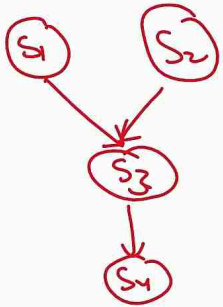
Shapes | Ink to Shape | Ink to Text | Math

Processes are independent of each other ✓

Anyone can start first.

Precedence graph

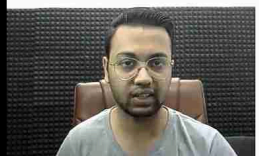
S1: a = b + c; 1
S2: d = e + f; 1
S3: g = a / d; 2
S4: f = g x l; 2

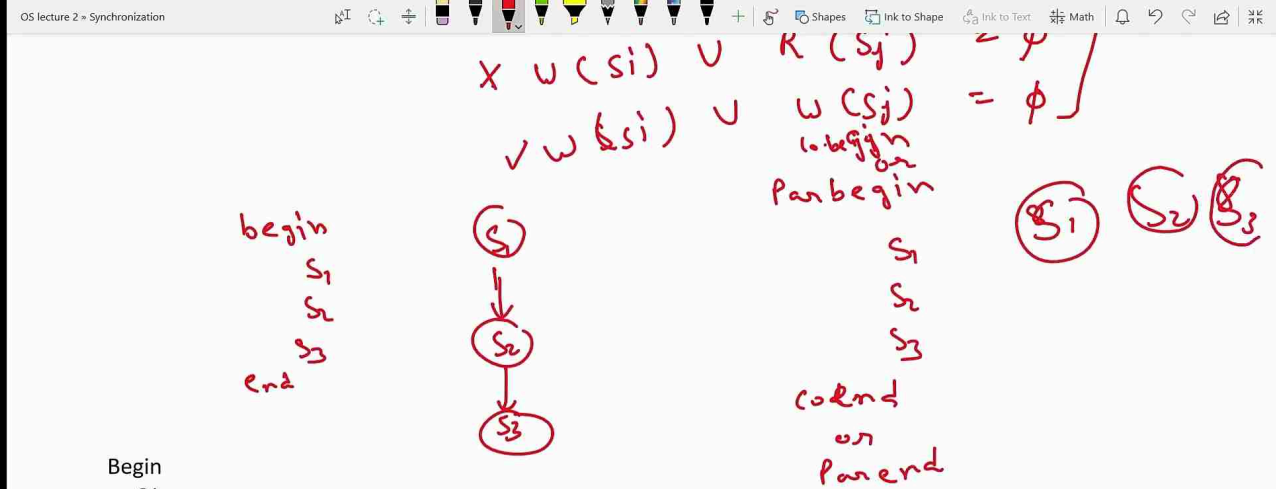


✓ R = [b, c]
W = [a]
R = [e, f]
W = [d]

Si Sj

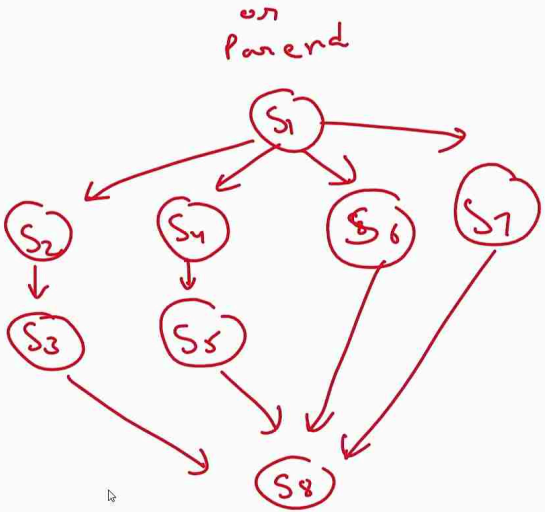
R(Si) ∪ W(Sj) = ϕ
W(Si) ∪ R(Sj) = ϕ
W(Si) ∪ W(Sj) = ϕ

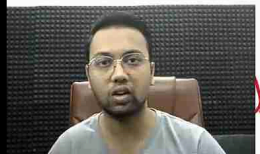


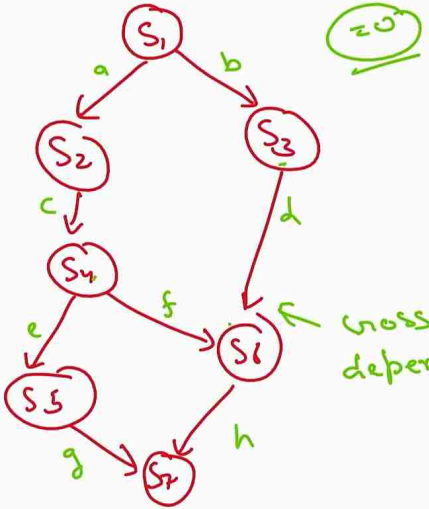


Begin
S1;
Parbegin

```
Begin
  S1;
    Parbegin
      Begin
        S2;
        S3;
      End
      Begin
        S4;
        S5;
      End
      S6;
      S7;
    Parend
    S8;
```







cross dependency

```
Parbegin
begin S1 Parbegin V(a), V(b) Parend, end
begin p(a) , S2 , V(c) end
begin p(b) , S3, V(d) end
begin p(c) S4 Parbegin V(e) V(f)
 Parend end
begin p(e) S5 V(g) end
begin parbegin p(f) p(d) parend
  S6 V(h) end

begin parbegin p(g)
```

Process Synchronization

A cooperating process is one that can affect or be affected by other processes executing in the system.

Cooperating processes can either

- directly share a logical address space (that is, both code and data)
- or be allowed to share data only through files or messages.

Concurrent access to shared data may result in data inconsistency!

In this chapter, we discuss various mechanisms to ensure –
The orderly execution of cooperating processes that share a logical address space,

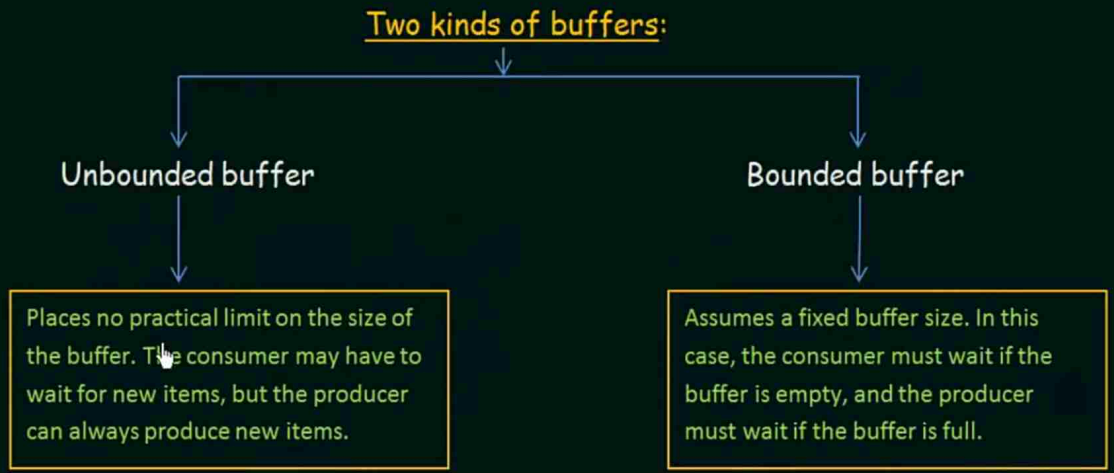
So that data consistency is maintained.

Producer Consumer Problem

A producer process produces information that is consumed by a consumer process.

For example, a compiler may produce assembly code, which is consumed by an assembler. The assembler, in turn, may produce object modules, which are consumed by the loader.

- One solution to the producer-consumer problem uses shared memory.
- To allow producer and consumer processes to run concurrently, we must have available a buffer of items that can be filled by the producer and emptied by the consumer.
- This buffer will reside in a region of memory that is shared by the producer and consumer processes.



"counter++" may be implemented in machine language (on a typical machine) as:

$register_1$ = counter
$register_1$ = $register_1$ + 1
counter = $register_1$

"counter--" may be implemented in machine language (on a typical machine) as:

$register_2$ = counter
$register_2$ = $register_2$ - 1
counter = $register_2$

T_0:	producer	execute	$register_1$	=	counter	{ $register_1$ = 5 }
T_1:	producer	execute	$register_1$	=	$register_1$ + 1	{ $register_1$ = 6 }
T_2:	consumer	execute	$register_2$	=	counter	{ $register_2$ = 5 }
T_3:	consumer	execute	$register_2$	=	$register_2$ - 1	{ $register_2$ = 4 }
T_4:	producer	execute	counter	=	$register_1$	{ counter = 6 }
T_5:	consumer	execute	counter	=	$register_2$	{ counter = 4 }

We would arrive at this incorrect state because we allowed both processes to manipulate the variable counter concurrently.

T_0:	producer	execute	$register_1$	=	counter	{ $register_1$ = 5 }
T_1:	producer	execute	$register_1$	=	$register_1$ + 1	{ $register_1$ = 6 }
T_2:	consumer	execute	$register_2$	=	counter	{ $register_2$ = 5 }
T_3:	consumer	execute	$register_2$	=	$register_2$ - 1	{ $register_2$ = 4 }
T_4:	producer	execute	counter	=	$register_1$	{ counter = 6 }
T_5:	consumer	execute	counter	=	$register_2$	{ counter = 4 }

We would arrive at this incorrect state because we allowed both processes to manipulate the variable counter concurrently.

A situation like this, 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.

Clearly, we want the resulting changes not to interfere with one another. Hence we need process synchronization.

The Critical-Section Problem

Consider a system consisting of n processes $\{ P_0, P_1, \ldots, P_n \}$.

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.

When one process is executing in its critical section, no other process is to be allowed to execute in its critical section.

That is, no two processes are executing in their critical sections 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 its critical section.
- The section of code implementing this request is the entry section.
- The critical section may be followed by an exit section.
- The remaining code is the remainder section.

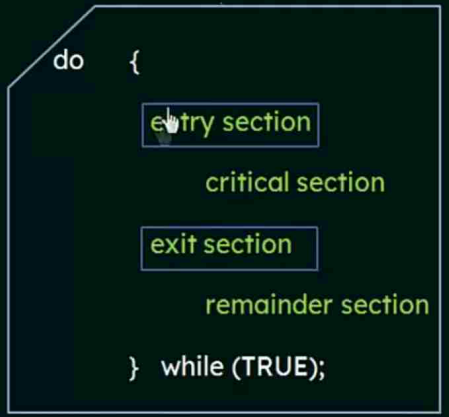


Figure: General structure of a typical process.

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

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 processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in the decision on which will enter its critical section next, and this selection cannot be postponed indefinitely.

3. Bounded waiting:

There exists a bound, or limit, on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

Critical Section Solution using "Lock"

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

* Execute in User Mode
* Multiprocess Solution
* No Mutual Exclusion Guarantee

Case 1

P_1 P_2 X

1234 | P_2 ✓

Lock != 0

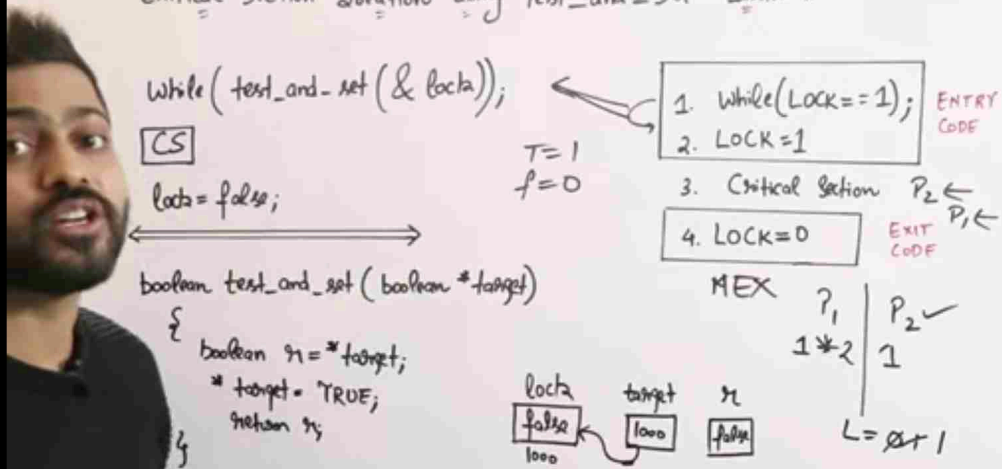
```
1. While(LOCK == 1);   ENTRY CODE
2. LOCK = 1
3. Critical Section
4. LOCK = 0            EXIT CODE
```

Lock = 0 → Vacant

1 → Full

Lock = $\emptyset_1$ * $\emptyset_1$

Critical Section Solution using "Test_and_Set" Instruction



Semaphores

- Semaphore proposed by Edsger Dijkstra, is a technique to manage concurrent processes by using a simple integer value, which is known as a semaphore.
- Semaphore is simply a variable which is non-negative and shared between threads. This variable is used to solve the critical section problem and to achieve process synchronization in the multiprocessing environment.
- A semaphore S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: wait ( ) and signal ( ).

wait ( ) → P [from the Dutch word proberen, which means "to test"]

signal ( ) → V [from the Dutch word verhogen, which means " to increment"]

wait ( ) → P [from the Dutch word proberen, which means "to test"]

signal ( ) → V [from the Dutch word verhogen, which means " to increment"]

Definition of wait ( ):

```
P (Semaphore S) {
        while (S <= 0)
          ; // no operation
        S-- ;
}
```

Definition of signal ( ):

```
V (Semaphore S) {
        S++ ;
}
```

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

Definition of wait ( ):

```
P (Semaphore S) {
    while (S <= 0)
      ; // no operation
    S-- ;
}
```

Definition of signal ( ):

```
V (Semaphore S) {
    S++ ;
}
```

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

Types of Semaphores:

1. Binary Semaphore:

The value of a binary semaphore can range only between 0 and 1. On some systems, binary semaphores are known as mutex locks, as they are locks that provide mutual exclusion.

2. Counting Semaphore:

Its value can range over an unrestricted domain. It is used to control access to a resource that has multiple instances.

Disadvantages of Semaphores

- The main disadvantage of the semaphore definition that was discussed is that it requires busy waiting.
- While a process is in its critical section, any other process that tries to enter its 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.

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.
- However, rather than engaging in busy waiting, the process can block itself.
- 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.

Deadlocks and Starvation

- The 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 by one of the waiting processes.
- The event in question is the execution of a signal() operation. When such a state is reached, these processes are said to be deadlocked.

P0
wait (S);
wait (Q);
.
.
.
signal (S);
signal (Q);

P1
wait (Q);
wait (S);
.
.
.
signal (Q);
signal (S);

Syntax of a Monitor

```
monitor monitor_name
{
// shared variable declarations
procedure P1 ( . . . ) {
. . .
}
procedure P2 ( . . . ) {
. . .
}
.
.
procedure Pn ( . . . ) {
. . .
}
initialization code ( . . . ) {
. . .
}
}
```

Monitors

- A high level abstraction that provides a convenient and effective mechanism for process synchronization.
- A monitor type presents a set of programmer-defined operations that provide mutual exclusion within the monitor.
- The monitor type also contains the declaration of variables whose values define the state of an instance of that type, along with the bodies of procedures or functions that operate on those variables.

Syntax of a Monitor

```
monitor monitor_name
{
// shared variable declarations
procedure P1 ( . . . ) {
. . .
}
procedure P2 ( . . . ) {
. . .
}
.
.
procedure Pn ( . . . ) {
. . .
}
initialization code ( . . . ) {
. . .
}
}
```


- A procedure defined within a monitor can access only those variables declared locally within the monitor and its formal parameters.
- Similarly, the local variables of a monitor can be accessed by only the local procedures.
- The monitor construct ensures that only one process at a time can be active within the monitor.

Condition Construct- **condition x, y ;**

The only operations that can be invoked on a condition variable are wait ( ) and signal ( ).

The operation x.wait( ) ; means that the process invoking this operation is suspended until another process invokes x.signal ( );

The x. signal ( ) operation resumes exactly one suspended process.

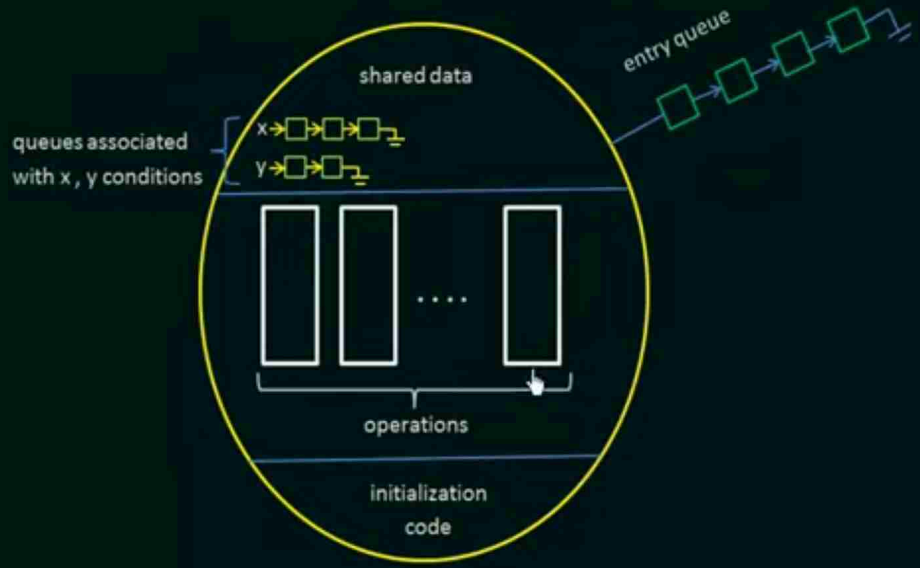
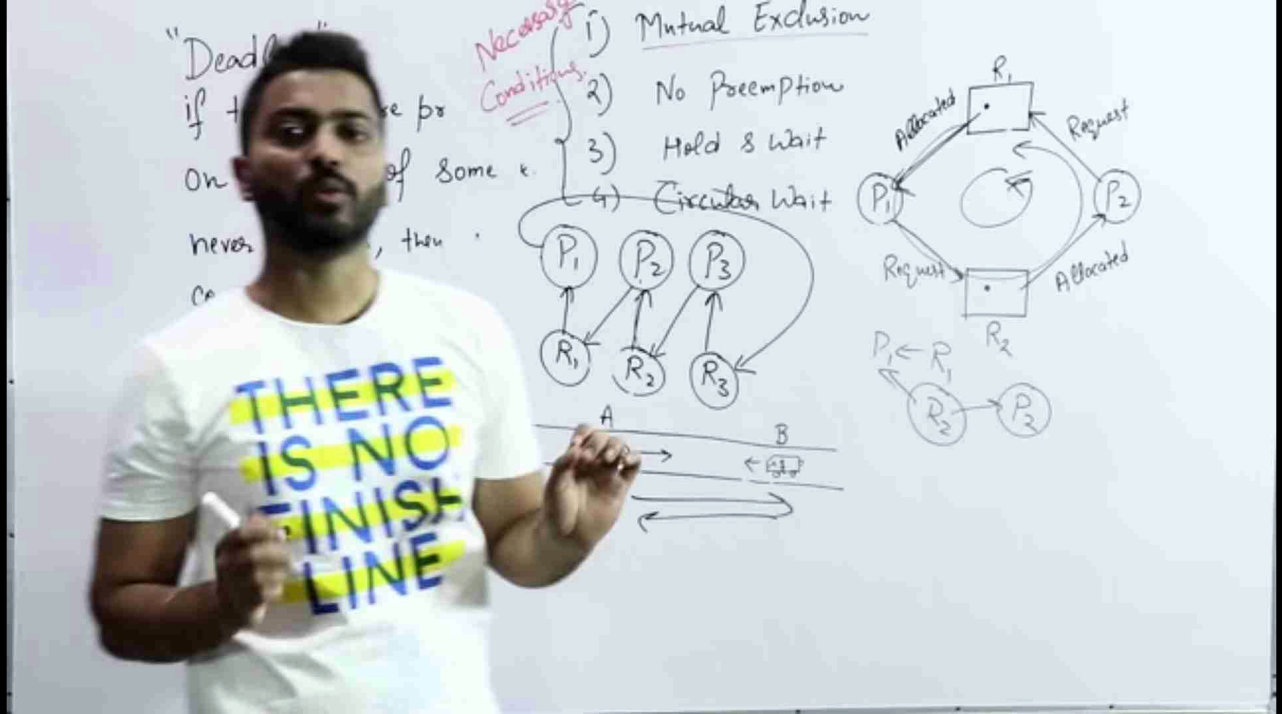
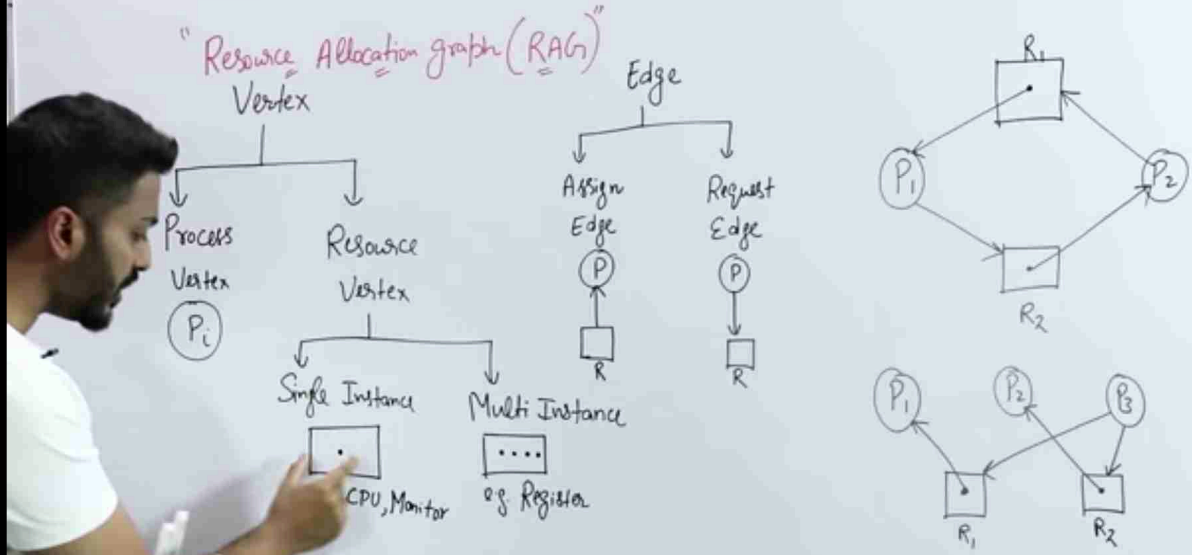


Fig: Schematic view of a monitor


Necessary Conditions
1) Mutual Exclusion
2) No Preemption
3) Hold & Wait
4) Circular Wait
P1
P2
P3
R1
R2
R3
A
B
Allocated
Request
Request
Allocated
P1 ← R1
R2
P2
THERE IS NO FINISH LINE


"Resource Allocation graph (RAG)"
Vertex
Process Vertex
P_i
Resource Vertex
Single Instance
CPU, Monitor
Multi Instance
e.g. Register
Edge
Assign Edge
P
R
Request Edge
P
R
R_1
P_1
P_2
R_2
P_1
P_2
P_3
R_1
R_2

urce Allocation graph (RAG)

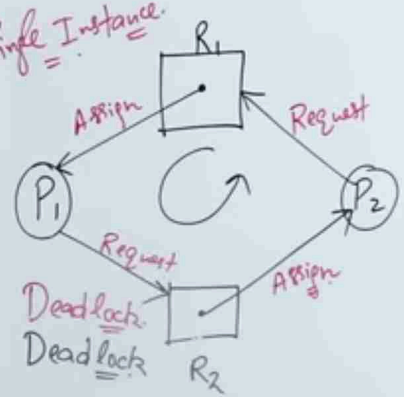
ertex

	Allocate		Request	
	R_1	R_2	R_1	R_2
P_1	1	0	0	1
P_2	0	1	1	0

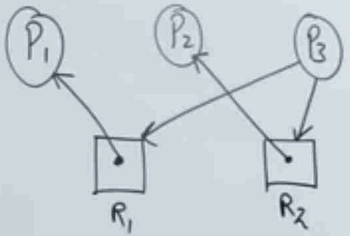
Availability $(R_1, R_2) = (0, 0)$

Register

Single Instance.



Deadlock



...urce Allocation graph (RAG)

...tex

Resource Vertex

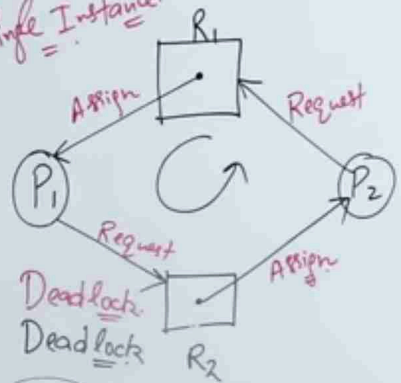
...le Inst...

	Allocate		Request	
	R_1	R_2	R_1	R_2
P_1	1	0	0	1
P_2	0	1	1	0

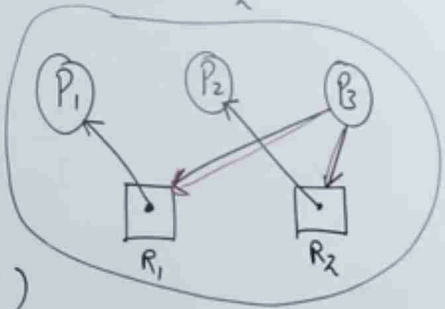
	Allocate		Request	
	R_1	R_2	R_1	R_2
P_1	1	0	0	0
P_2	0	1	0	0
P_3	0	0	1	1

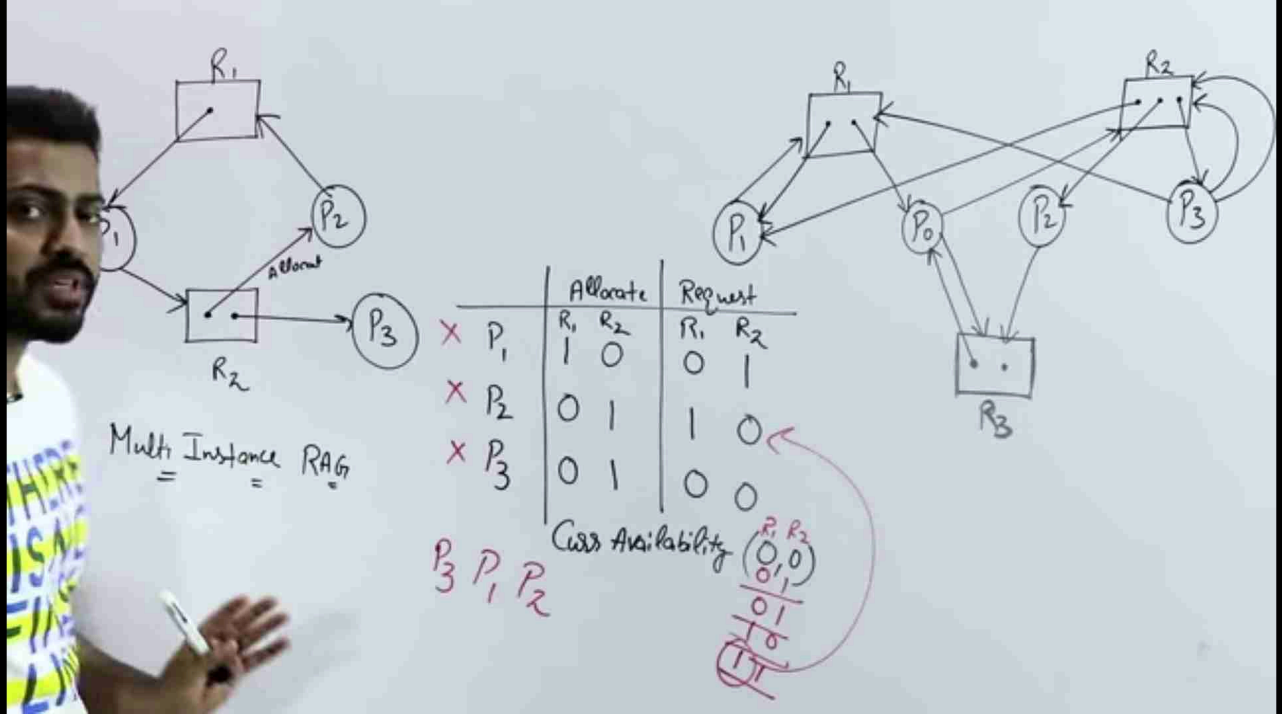
Availability (0,0)

Single Instance.



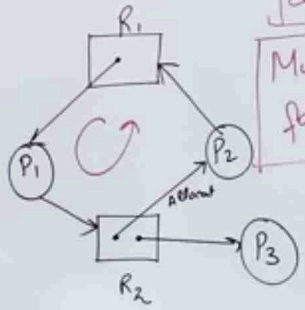
Deadlock



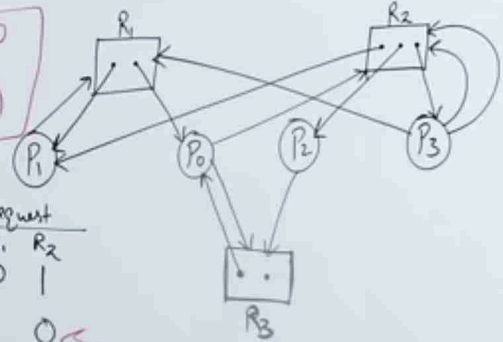

R1
P1
P2
Allocate
P3
R2
Multi Instance RAG
= =
Allocate
Request
R1 R2
R1 R2
P1 1 0 0 1
P2 0 1 1 0
P3 0 1 0 0
Curr Availability (0,0)
P3 P1 P2
R1
R2
P1
P0
P2
P3
R3

SI && CW ⇒ Deadlock (True)

Multi Instance && CW
⇓
false ≡ Deadlock (No)



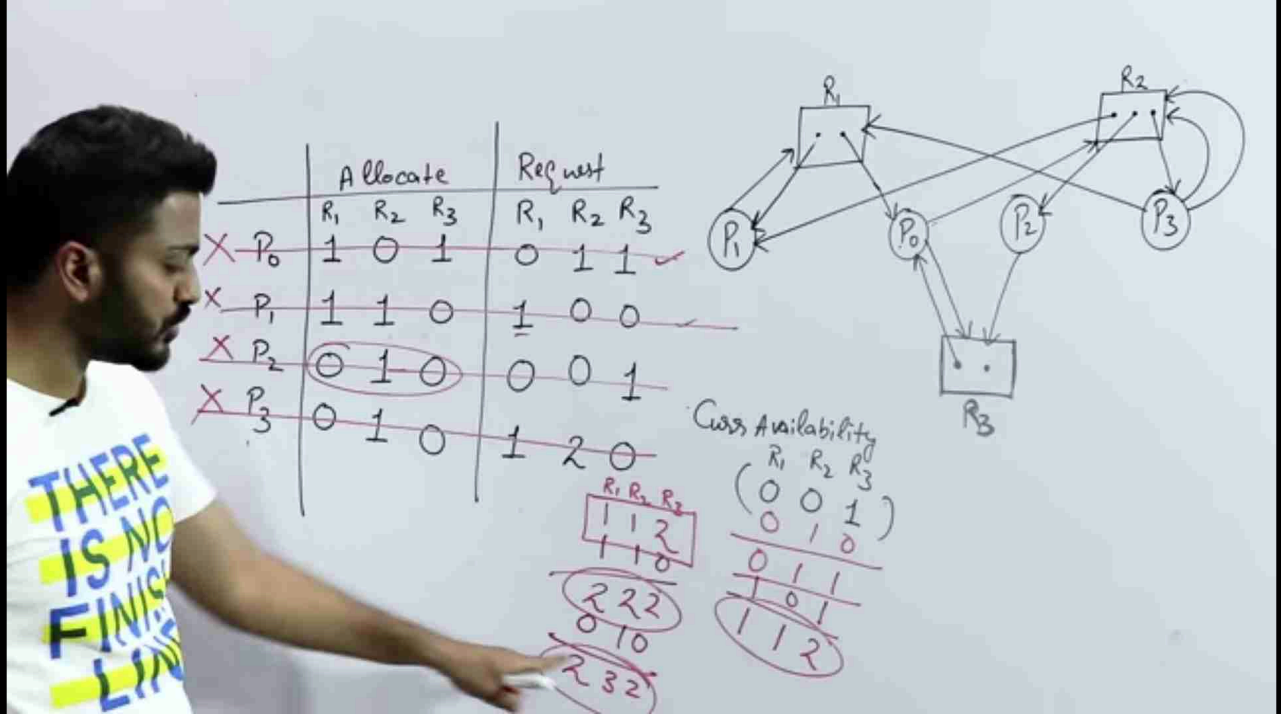
Multi Instance RAG



	Allocate		Request	
	R_1	R_2	R_1	R_2
× P_1	1	0	0	1
× P_2	0	1	1	0
× P_3	0	1	0	0

Avail Availability (R_1, R_2) (0,0) (0,1) (0,1) (1,1)

P_3 P_1 P_2

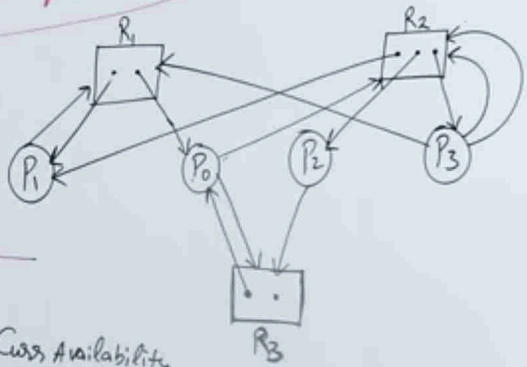

Allocate
Request
R1 R2 R3
P0 1 0 1 0 1 1
P1 1 1 0 1 0 0
P2 0 1 0 0 0 1
P3 0 1 0 1 2 0
R1 R2 R3
1 1 2
1 1 0
2 2 2
0 1 0
2 3 2
Curr Availability
R1 R2 R3
0 0 1
0 1 0
0 1 1
1 0 1
1 1 2
R1
R2
R3
P1
P0
P2
P3

false

MISS CW ⇒ Always Deadlock

	Allocate			Request		
	R_1	R_2	R_3	R_1	R_2	R_3
P_0	1	0	1	0	1	1
P_1	1	1	0	1	0	0
P_2	0	1	0	0	0	1
P_3	0	1	0	1	2	0

$P_2 \rightarrow P_0 \rightarrow P_1 \rightarrow P_3$



Cuss Availability

$R_1\ R_2\ R_3$

(0 0 1)

0 1 0

0 1 1

1 0 1

1 1 2

$R_1\ R_2\ R_3$

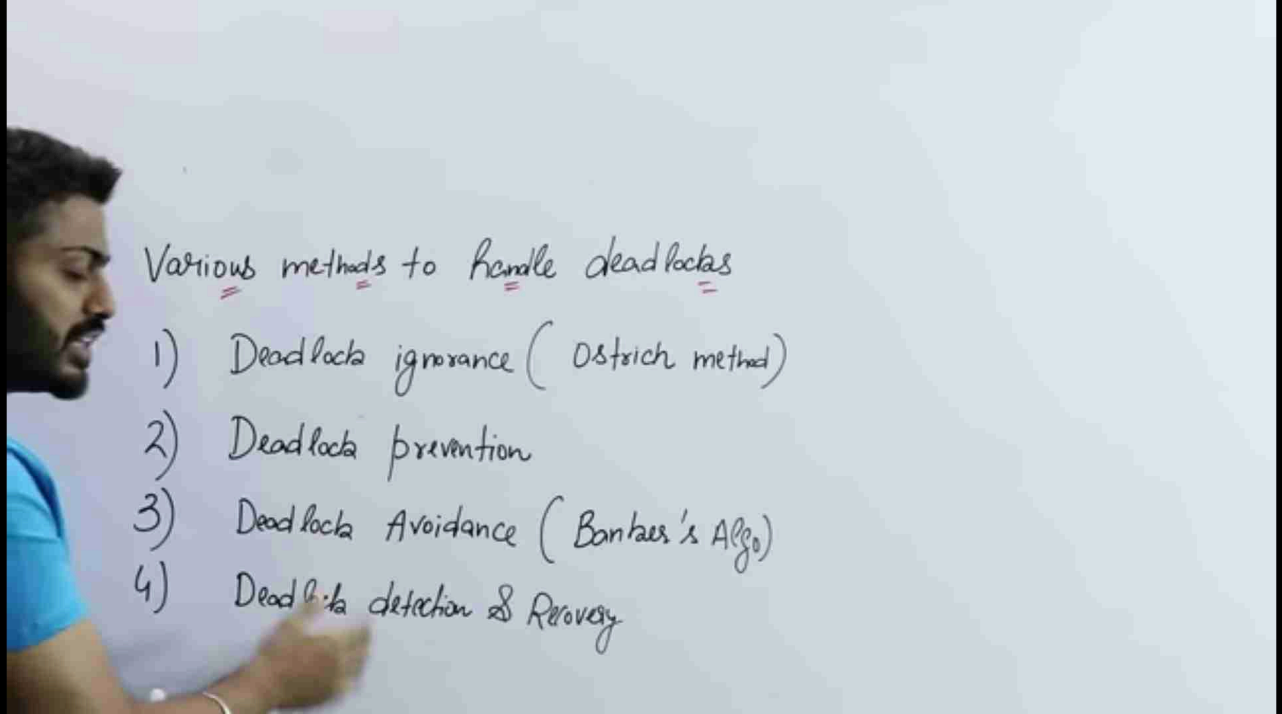
1 1 2

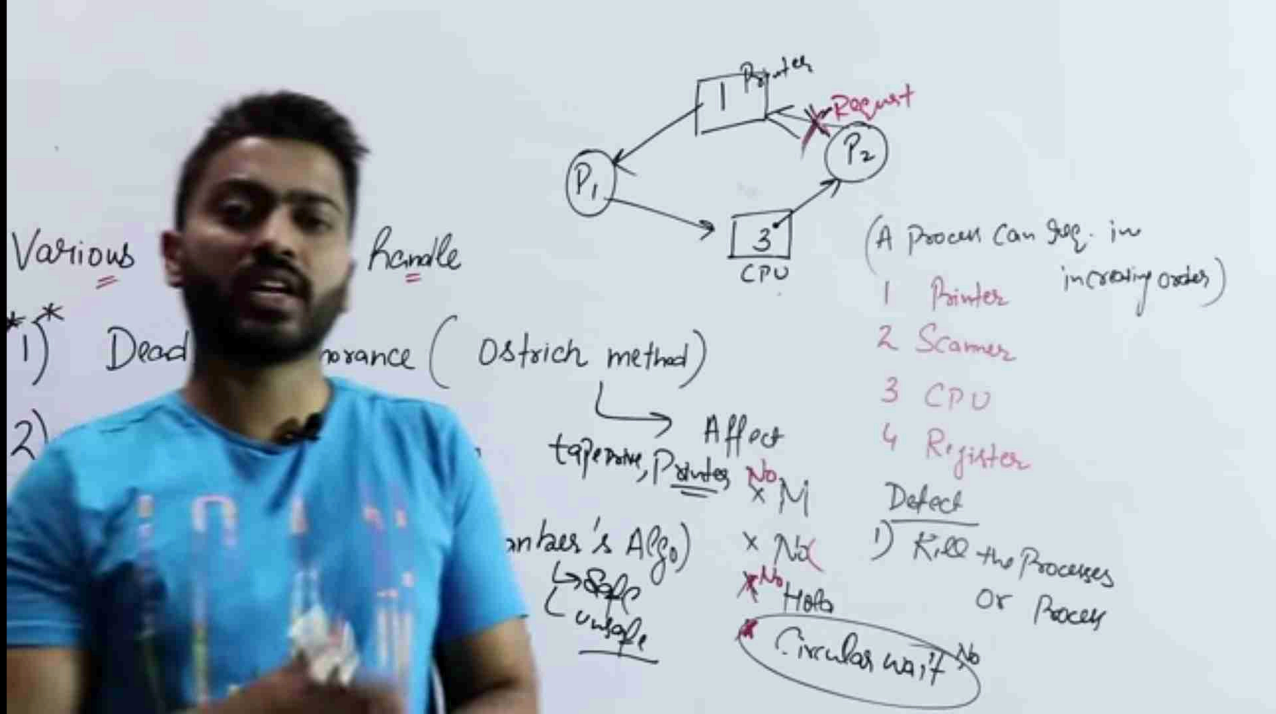
1 1 0

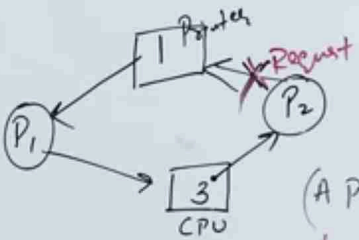
2 2 2

0 1 0

2 3 2


Various methods to handle deadlocks
1) Deadlock ignorance (Ostrich method)
2) Deadlock prevention
3) Deadlock Avoidance (Bankers's Algo)
4) Deadlock detection & Recovery


Various
Handle
1) Dead
ance (Ostrich method)
2)
Affect
tape, Printer
No
M
Banker's Algo)
Safe
Unsafe
No
Hold
Circular wait
No
Printer
1
Request
P1
P2
3
CPU
(A Process can req. in increasing order)
1 Printer
2 Scanner
3 CPU
4 Register
Defect
1) Kill the Processes
or Proces



(A Process can req. in increasing order)

1 Printer
2 Scanner
3 CPU
4 Register

...ous methods to Handle

Deadlock ignorance (Ostrich method) → Affect

Deadlock prevention → tape drive, Printer — No ✗ Mutual Exclusion

Deadlock Avoidance (Banker's Algo) → Safe / Unsafe — ✗ No Hold & Wait

Deadlock detection & Recovery — ✗ No Circular wait

Defect
1) Kill the Processes
2) Resource Preemption or Process

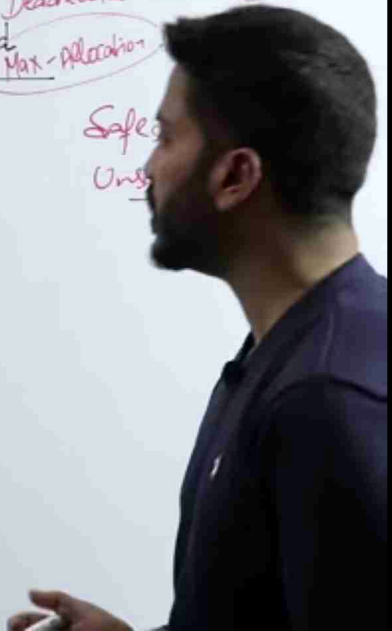
"BANKER's Algo"

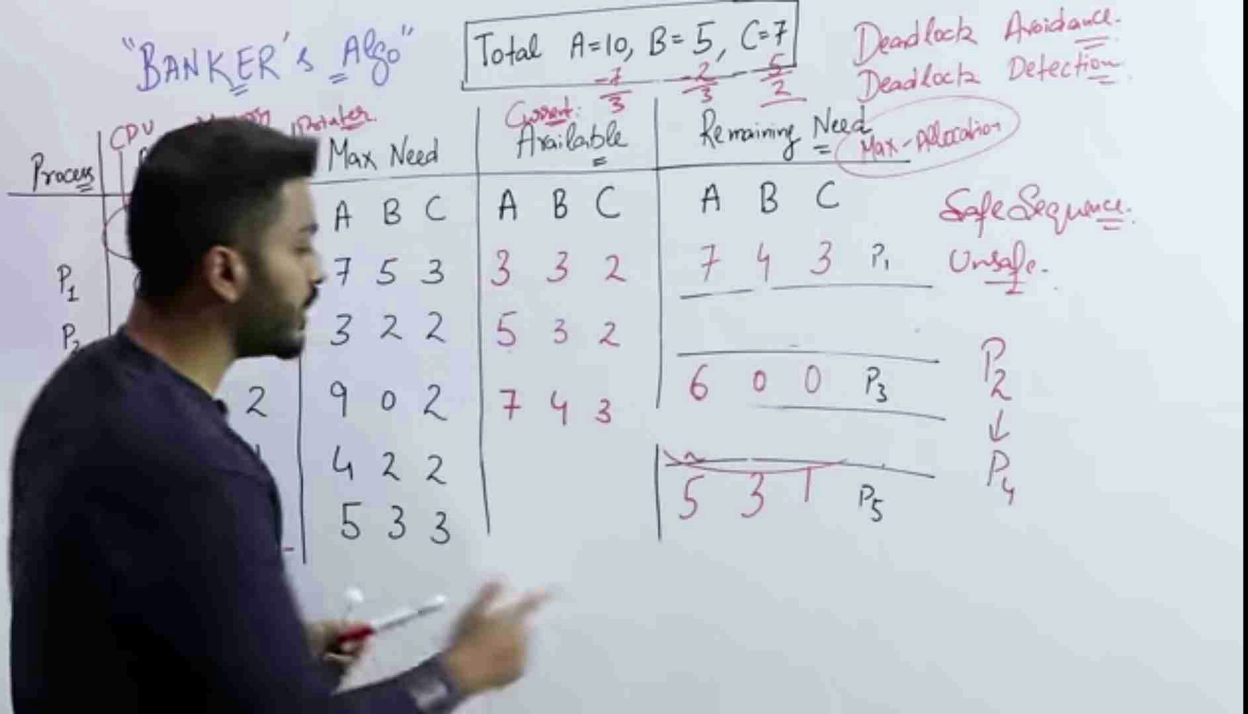
Total A=10, B=5, C=7

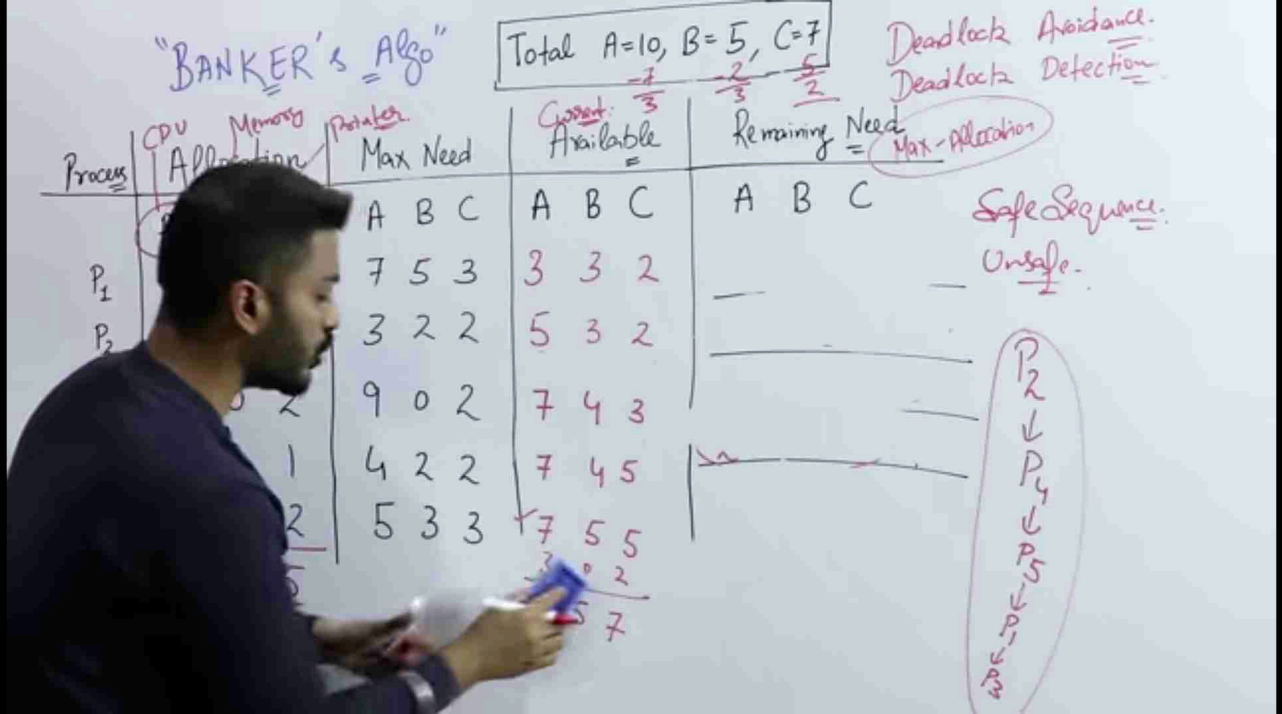
Deadlock Avoidance.
Deadlock Detection

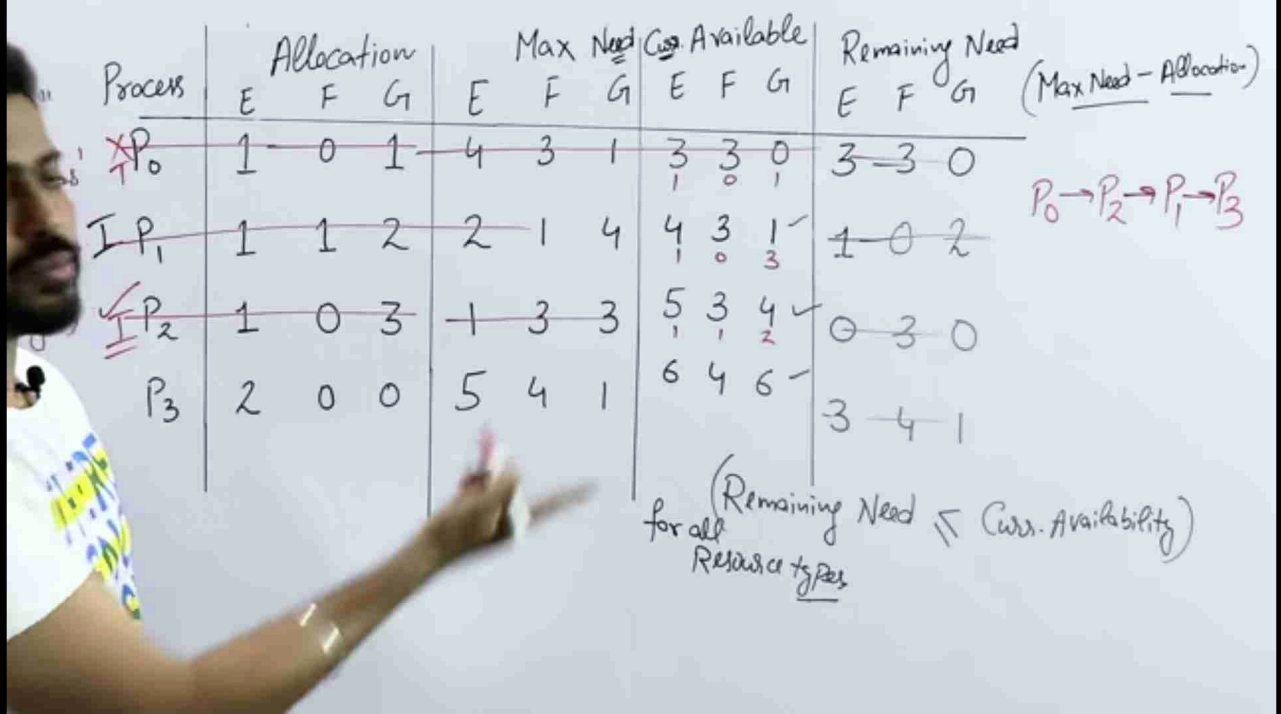
Process	Allocation (CPU / Memory / Printer) A	B	C	Max Need A	B	C	Current Available A	B	C	Remaining Need = Max - Allocation A	B	C
							3	3	2			
P_1	0	1	0	7	5	3				7	4	3
P_2	2	0	0	3	2	2				1	2	2
P_3	3	0	2	9	0	2				6	0	0
P_4	2	1	1	4	2	2				2	1	1
P_5	0	0	2	5	3	3				5	3	1
	7	2	5									

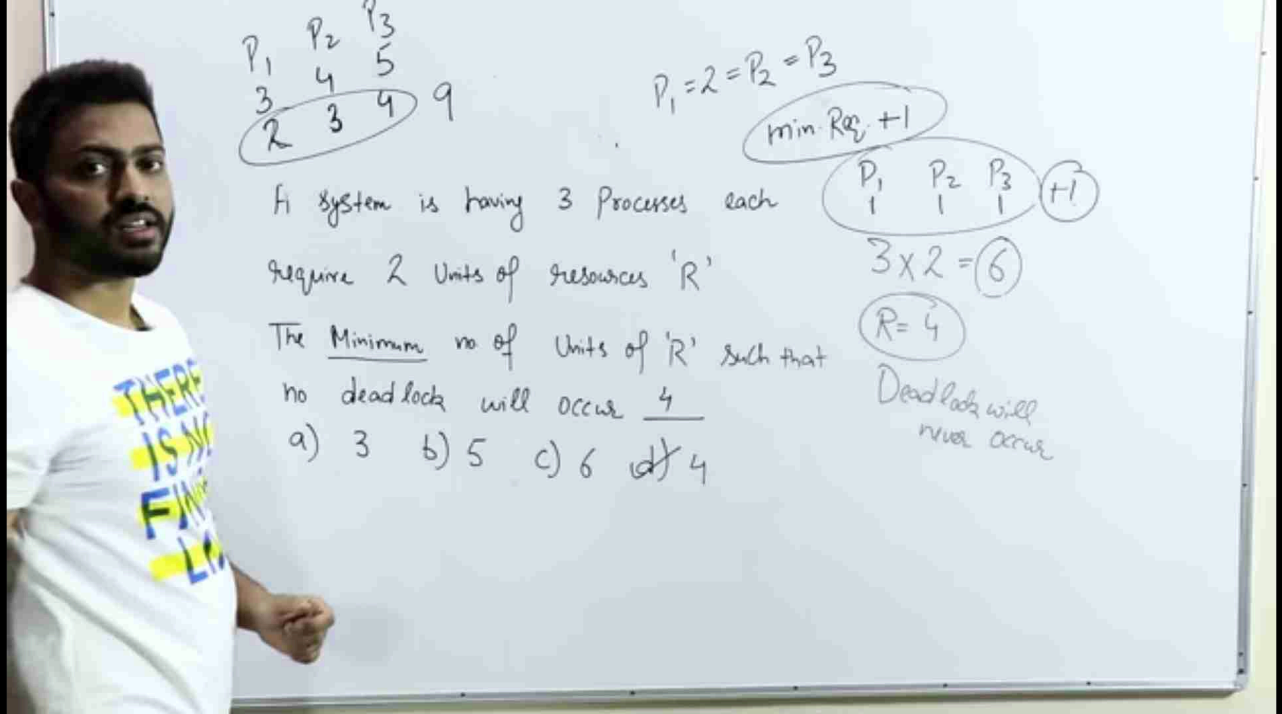
Safe
Unsafe




"BANKER's Algo"
Total A=10, B=5, C=7
Deadlock Avoidance.
Deadlock Detection.
Process
Max Need
Current Available
Remaining Need = Max - Allocation
A B C
7 5 3
3 2 2
9 0 2
4 2 2
5 3 3
3 3 2
5 3 2
7 4 3
7 4 3 P1
6 0 0 P3
5 3 1 P5
Safe Sequence.
Unsafe.
P2
P4


"BANKER's Algo"
Total A=10, B=5, C=7
Deadlock Avoidance.
Deadlock Detection.
Process
CPU
Memory
Peripheral
Allocation
Max Need
Current Available
Remaining Need = Max - Allocation
A B C
7 5 3
3 2 2
9 0 2
4 2 2
5 3 3
3 3 2
5 3 2
7 4 3
7 4 5
7 5 5
Safe Sequence.
Unsafe.
P2 ↓ P4 ↓ P5 ↓ P1 ↓ P3


Process
Allocation
E F G
Max Need
E F G
Curr. Available
E F G
Remaining Need
E F G
(Max Need − Allocation)
P0 1 0 1 4 3 1 3 3 0 3 3 0
P1 1 1 2 2 1 4 4 3 1 1 0 2
P2 1 0 3 1 3 3 5 3 4 0 3 0
P3 2 0 0 5 4 1 6 4 6 3 4 1
P0 → P2 → P1 → P3
for all Resource types
(Remaining Need ≤ Curr. Availability)


P1 P2 P3
3 4 5
2 3 4
9
P1 = 2 = P2 = P3
min Req +1
P1 P2 P3
1 1 1
+1
3 x 2 = 6
R = 4
Deadlock will never occur
A system is having 3 Processes each require 2 Units of resources 'R'
The Minimum no of Units of 'R' such that no deadlock will occur 4
a) 3 b) 5 c) 6 d) 4

Handling Deadlocks

Deadlock is a situation where a process or a set of processes is blocked, waiting for some other resource that is held by some other waiting process. It is an undesirable state of the system. The following are the four conditions that must hold simultaneously for a deadlock to occur.

1. **Mutual Exclusion** – A resource can be used by only one process at a time. If another process requests for that resource then the requesting process must be delayed until the resource has been released.
2. **Hold and wait** – Some processes must be holding some resources in the non-shareable mode and at the same time must be waiting to acquire some more resources, which are currently held by other processes in the non-shareable mode.
3. **No pre-emption** – Resources granted to a process can be released back to the system only as a result of voluntary action of that process after the process has completed its task

3. **No pre-emption** – Resources granted to a process can be released back to the system only as a result of voluntary action of that process after the process has completed its task.
4. **Circular wait** – Deadlocked processes are involved in a circular chain such that each process holds one or more resources being requested by the next process in the chain.

Methods of handling deadlocks: There are four approaches to dealing with deadlocks.

```
1. Deadlock Prevention
2. Deadlock avoidance (Banker's Algorithm)
3. Deadlock detection & recovery
4. Deadlock Ignorance (Ostrich Method)
```

These are explained below.

1. Deadlock Prevention: The strategy of deadlock prevention is to design the system in such a way that the possibility of deadlock is excluded. The indirect methods prevent the occurrence of one of three necessary conditions of deadlock i.e., mutual exclusion, no pre-emption, and hold

These are explained below.

1. Deadlock Prevention: The strategy of deadlock prevention is to design the system in such a way that the possibility of deadlock is excluded. The indirect methods prevent the occurrence of one of three necessary conditions of deadlock i.e., mutual exclusion, no pre-emption, and hold and wait. The direct method prevents the occurrence of circular wait.
Prevention techniques – Mutual exclusion – are supported by the OS.
Hold and Wait – the condition can be prevented by requiring that a process requests all its required resources at one time and blocking the process until all of its requests can be granted at the same time simultaneously. But this prevention does not yield good results because:

- long waiting time required
- inefficient use of allocated resource
- A process may not know all the required resources in advance

No pre-emption – techniques for 'no pre-emption are'

- If a process that is holding some resource, requests another resource that can not be immediately allocated to it, all resources currently being held are released and if necessary, request again together with the additional resource.
- If a process requests a resource that is currently held by another process, the OS may pre-empt the second process and require it to release its resources. This works only if both processes do not have the same priority.

Circular wait One way to ensure that this condition never holds is to impose a total ordering of all resource types and to require that each process requests resources in increasing order of enumeration, i.e., if a process has

requests resources in increasing order of enumeration, i.e., if a process has been allocated resources of type R, then it may subsequently request only those resources of types following R in ordering.

2. Deadlock Avoidance: The deadlock avoidance Algorithm works by proactively looking for potential deadlock situations before they occur. It does this by tracking the resource usage of each process and identifying conflicts that could potentially lead to a deadlock. If a potential deadlock is identified, the algorithm will take steps to resolve the conflict, such as rolling back one of the processes or pre-emptively allocating resources to other processes. The Deadlock Avoidance Algorithm is designed to minimize the chances of a deadlock occurring, although it cannot guarantee that a deadlock will never occur. This approach allows the three necessary conditions of deadlock but

occur. This approach allows the three necessary conditions of deadlock but makes judicious choices to assure that the deadlock point is never reached. It allows more concurrency than avoidance detection A decision is made dynamically whether the current resource allocation request will, if granted, potentially lead to deadlock. It requires knowledge of future process requests. Two techniques to avoid deadlock :

1. Process initiation denial
2. Resource allocation denial

Advantages of deadlock avoidance techniques:

- Not necessary to pre-empt and rollback processes
- Less restrictive than deadlock prevention

Disadvantages :

- Future resource requirements must be known in advance
- Processes can be blocked for long periods

Disadvantages :

- Future resource requirements must be known in advance
- Processes can be blocked for long periods
- Exists a fixed number of resources for allocation

Banker's Algorithm:

The Banker's Algorithm is based on the concept of resource allocation graphs. A resource allocation graph is a directed graph where each node represents a process, and each edge represents a resource. The state of the system is represented by the current allocation of resources between processes. For example, if the system has three processes, each of which is using two resources, the resource allocation graph would look like this:

Processes A, B, and C would be the nodes, and the resources they are using would be the edges connecting them. The Banker's Algorithm works by analyzing the state of the system and determining if it is in a safe state

represents a process, and each edge represents a resource. The state of the system is represented by the current allocation of resources between processes. For example, if the system has three processes, each of which is using two resources, the resource allocation graph would look like this:

Processes A, B, and C would be the nodes, and the resources they are using would be the edges connecting them. The Banker's Algorithm works by analyzing the state of the system and determining if it is in a safe state or at risk of entering a deadlock.

To determine if a system is in a safe state, the Banker's Algorithm uses two matrices: the available matrix and the need matrix. The available matrix contains the amount of each resource currently available. The need matrix contains the amount of each resource required by each process.

The Banker's Algorithm then checks to see if a process can be completed without overloading the system. It does this by subtracting the amount of each resource used by the process from the available matrix and adding it to the need matrix. If the result is in a safe state, the process is allowed to proceed, otherwise, it is blocked until more resources become available.

The Banker's Algorithm is an effective way to prevent deadlocks in multiprogramming systems. It is used in many operating systems, including Windows and Linux. In addition, it is used in many other types of systems, such as manufacturing systems and banking systems.

The Banker's Algorithm is a powerful tool for resource allocation problems, but it is not foolproof. It can be fooled by processes that consume more resources than they need, or by processes that produce more resources than they need. Also, it can be fooled by processes that consume resources in an unpredictable manner. To

3. Deadlock Detection: Deadlock detection is used by employing an algorithm that tracks the circular waiting and kills one or more processes so that the deadlock is removed. The system state is examined periodically to determine if a set of processes is deadlocked. A deadlock is resolved by aborting and restarting a process, relinquishing all the resources that the process held.

- This technique does not limit resource access or restrict process action.
- Requested resources are granted to processes whenever possible.
- It never delays the process initiation and facilitates online handling.
- The disadvantage is the inherent pre-emption losses.

4. Deadlock Ignorance: In the Deadlock ignorance method the OS acts like the deadlock never occurs and completely ignores it even if the deadlock occurs. This method only applies if the deadlock occurs very

applies if the deadlock occurs very rarely. The algorithm is very simple. It says " if the deadlock occurs, simply reboot the system and act like the deadlock never occurred." That's why the algorithm is called the **Ostrich Algorithm**.

Advantages:

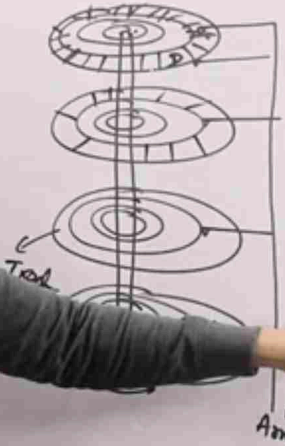
- Ostrich Algorithm is relatively easy to implement and is effective in most cases.
- It helps in avoiding the deadlock situation by ignoring the presence of deadlocks.

Disadvantages:

- Ostrich Algorithm does not provide any information about the deadlock situation.
- It can lead to reduced performance of the system as the system may be blocked for a long time.
- It can lead to a resource leak, as resources are not released when the system is blocked due to deadlock.

Disk Architecture

Platters → Surface → Track → Sectors → Data



$8 \times 2 \times 256 \times 512 \times 512\,KB$

$2^3 \times 2^1 \times 2^8 \times 2^9 \times 2^9 \times 2^{10} B$

Disk Size = $P \times S \times T \times S \times D$

$= 2^{40} B$

$= 1TB.$

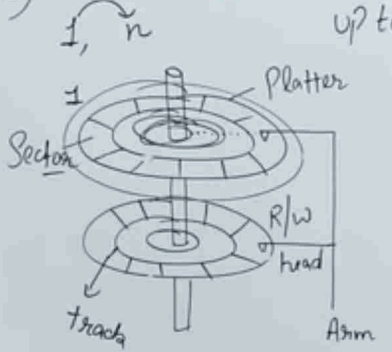
$1K = 2^{10}$
$1M = 2^{20}$
$1G = 2^{30}$
$1T = 2^{40}$

"Disk Scheduling Algorithms"

- FCFS (First Come first Serve)
- SSTF (Shortest Seek time first)
- SCAN
- LOOK
- CSCAN (Circular SCAN)
- CLOOK (Circular Look)

Goal: To minimize the Seek t

Seek time = time taken to reach up to desired track

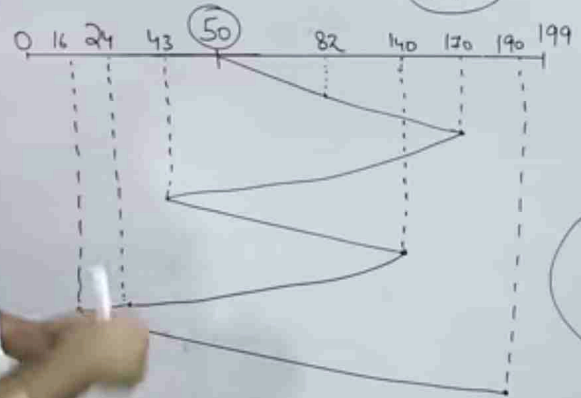


Platter
↓
Surface
↓
track
↓
Sector

"Disk Scheduling Algorithms"

→ FCFS (First Come first Serve)

= 642



Que. A disk contains 200 tracks (0-199)

Request queue contains track no.

82, 170, 43, 140, 24, 16, 190 respectively

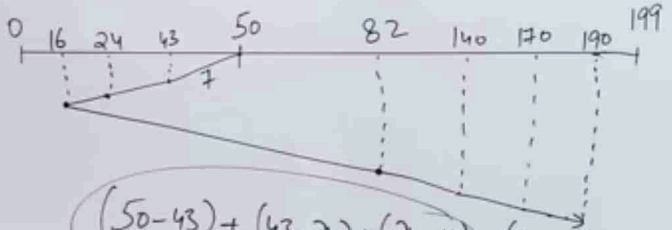
Current Position of R/W head = 50

Caclulate total no of tracks movement by R/W head.

(82-50) + (170-82) + (170-43) + (140-43)

+ (140-24) + (24-16) + (190-16)

(170-50) + (170-43) + (140-43) + (140-16) + (190-16)



$(50-43) + (43-24) + (24-16) + (82-16)$
$+ (140-82) + (170-140) +$
$(190-170)$

$(50-16) + (190-16)$

Qus. A disk contains 200 tracks

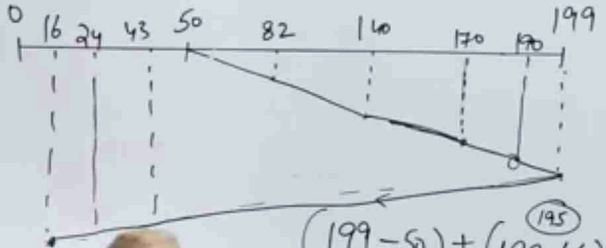
Request queue contains tracks

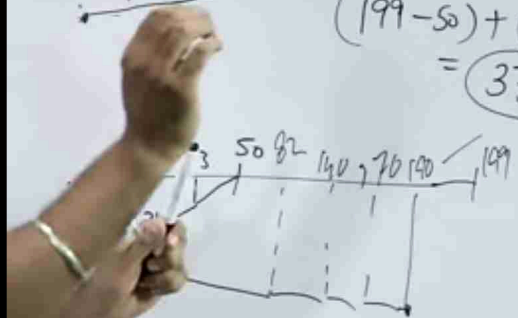
82, 170, 43, 140, 24, 16, 190

Current position of R/W head = 50

→ Calculate total no of tracks movement by head using Shortest Seek time first

→ if R/W head takes 1 ms to move from one track to another then total time taken ___ ?





Qus. A disk contains 200 tracks

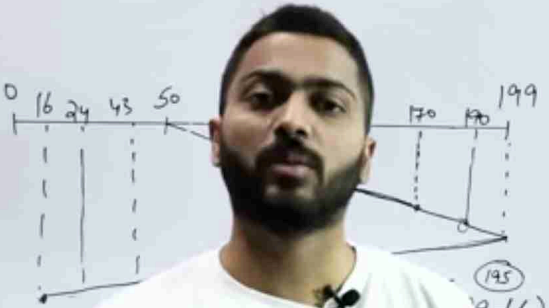
Request queue contains tracks

82, 170, 43, 140, 24, 16, 190

Current Position of R/W head = 50

→ Calculate total no of tracks movement by head using SCAN ?

→ if R/W head takes 1 ns to move from one track to another then total time taken ___ ?



Qus: A disk contains 200 tracks (0-199)

Request queue contains track no.

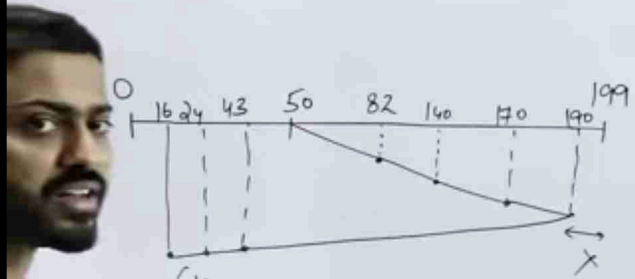
82, 170, 43, 140, 24, 16, 190 respectively

Current position of R/W head = 50

→ Calculate total no of tracks movement by R/W head using SCAN ?

→ if R/W head takes 1 ms to move from one track to another then total time taken 332 x 1 ms?

332 ms



$(190-50)+(190-16)$ * *

$= 314$

Direction is towards large value ?

Qus: A disk contains 200 tracks (0-19

Request queue contains tracks no.

82, 170, 43, 140, 24, 16, 190 respectiv

Current position of R/w head = 50

→ Calculate total no of tracks movement by R/w head using LOOK ?

→ if R/w head takes 1 ms to move from one track to another then total time taken ___ ?

Qus. A disk contains 200 tracks

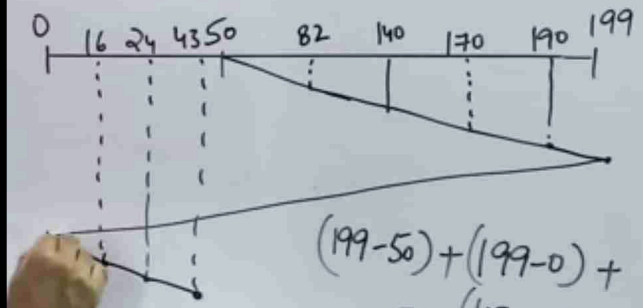
Request queue contains tracks

82, 170, 43, 140, 24, 16, 190

Current position of R/W head = 50

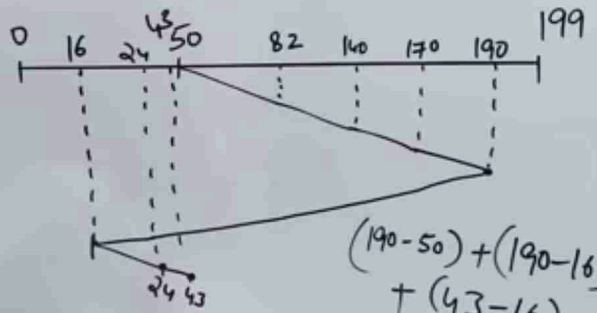
→ Calculate total no of tracks movement by head using C-SCAN?

– Direction is towards large value?



$(199-50)+(199-0)+(43-0)$

$=391$



$(190-50)+(190-16)$
$+(43-16)$
$= 341$

** Direction is towards large value

Qus. A disk contains 200 tracks

Request queue contains tracks

82, 170, 43, 140, 24, 16, 190

Current position of R/W head = 50

→ Calculate total no of tracks movement h
head using C-LOOK ?

→ if R/W head takes 1 ms to move from
one track to another then total time
taken ___ ?

File System in OS

↓

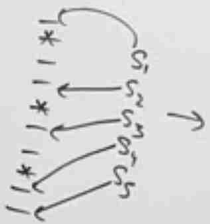
Software

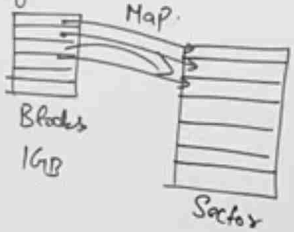
How stored

" fetched

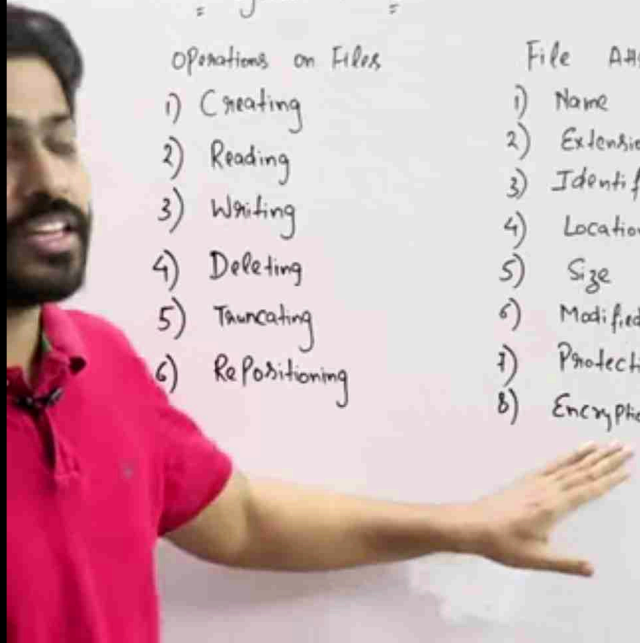
User → file → folder / → file system

Directory





File System in OS



Operations on Files

1) Creating
2) Reading
3) Writing
4) Deleting
5) Truncating
6) Repositioning

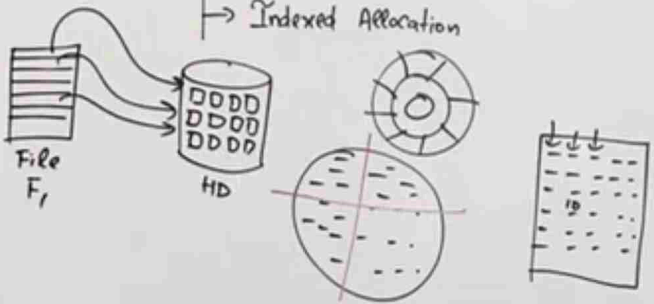
File Attributes

1) Name
2) Extension (Type)
3) Identifier /- - - ..
4) Location
5) Size
6) Modified date, Created date
7) Protection / Permission
8) Encryption, Compression

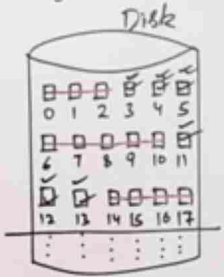
Allocation Methods.

1) Efficient Disk Utilization
2) Access faster

- Contiguous Allocation
- Non Contiguous Allocation
  - Linked List Allocation
  - Indexed Allocation



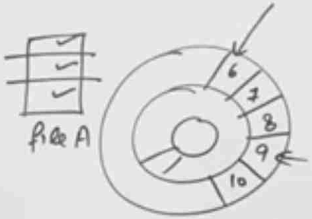
Contiguous Allocation



Directory

file	Start	length
A	0	3
B	6	5
C	14	4

D



Advantages

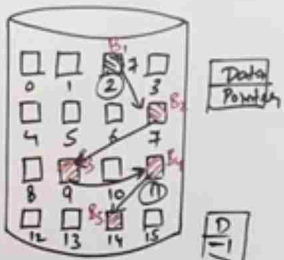
1) Easy to Implement
2) Excellent Read Performance

Sequential

Disadvantages

1) Disk will become fragmented
2) Difficult to grow file

Linked List Allocation



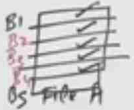
Data
Pointer

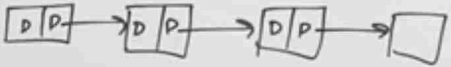
D
=1

Directory

file	start
A	2

Pointer

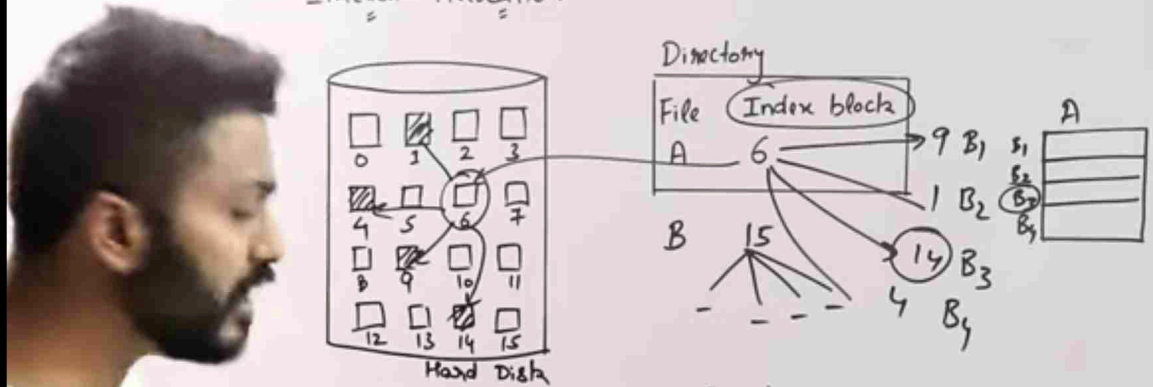




Advantages: 1) No External fragmentation
2) File size can increase

Disadvantages: 1) Large Seek time
2) Random Access / Direct Access Difficult
3) Overhead of Pointers

Indexed Allocation



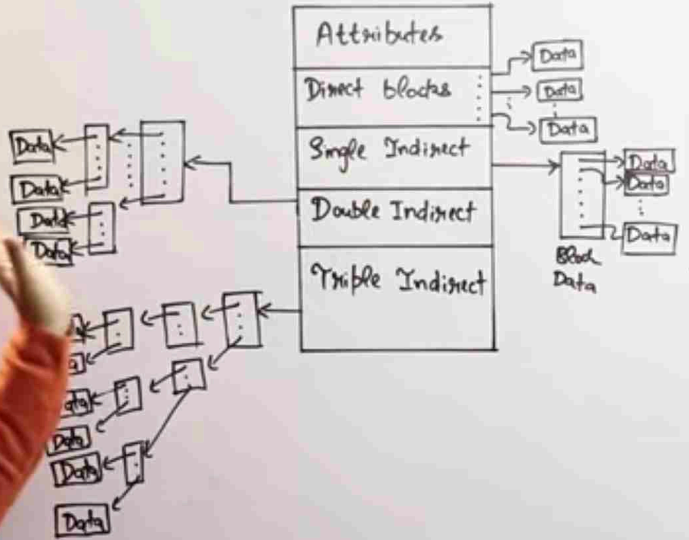
Advantages

1) Support direct access
2) No External fragmentation

Disadvantages

1) Pointer overhead
2) Multilevel Index

"Unix Inode Structure"



A file system uses Unix inode data structure which Contain 8 direct block addresses, One indirect block, One double and One triple indirect block. The size of each disk block is 128B and size of each block address is 8B. Find the max. possible file size?