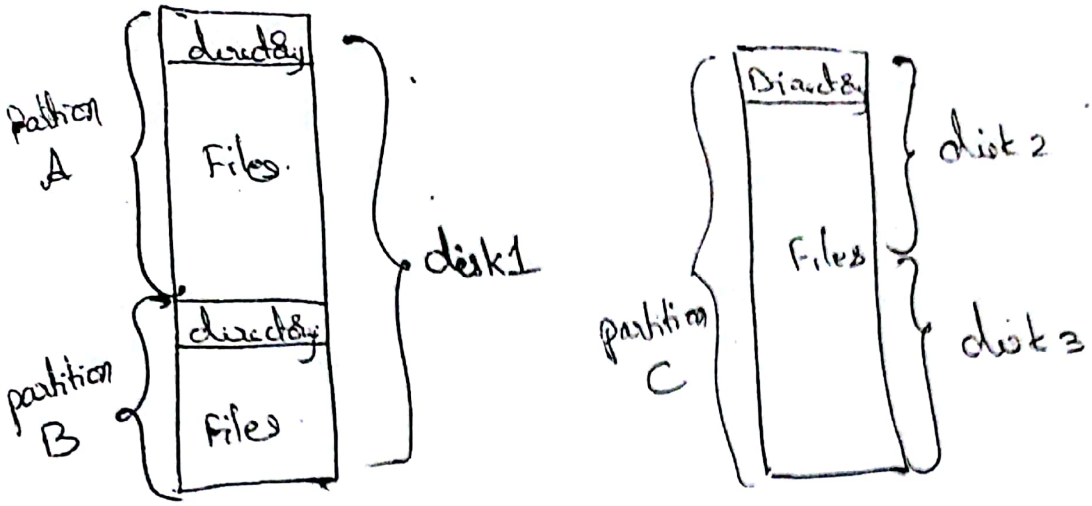


Directory Overview:-

To insert an entry, to delete an entry, to search for an named entry, to list all entries in the directory, a directory should have a logical structure.

The following are the operations that can be performed on a Directory

1. search for a file
2. Create a file
3. Delete a file
4. List a diretory
5. Rename a file
6. Traverse the FS.

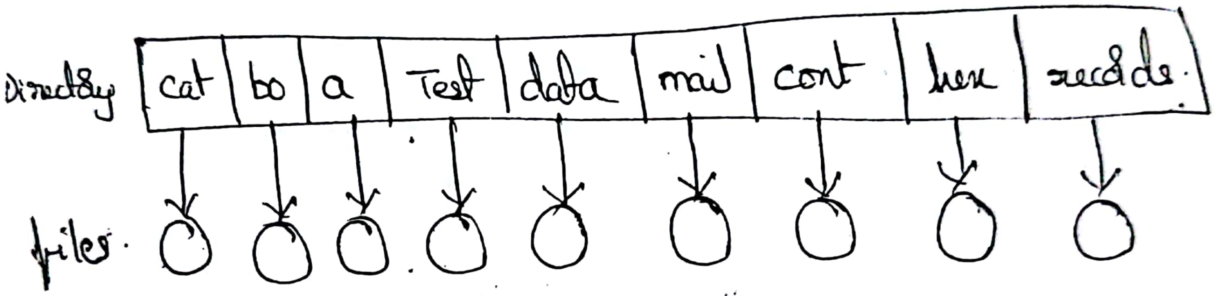
Schemas:

The following are the different schemas for defining a logical structure of a directory.

1) Single level Directory:

→ Simplest Directory structure is single level Directory.

→ All files are contained in the same directory



Limitations

1) Since all files of different users are in the same directory, files must have unique name

2) Limitation on the file name length.
Ex:- MS-DOC OS allows only 11 character filename
UNIX allows 255 characters.

3) It is difficult to remember the names of all the files as the no of files increases.

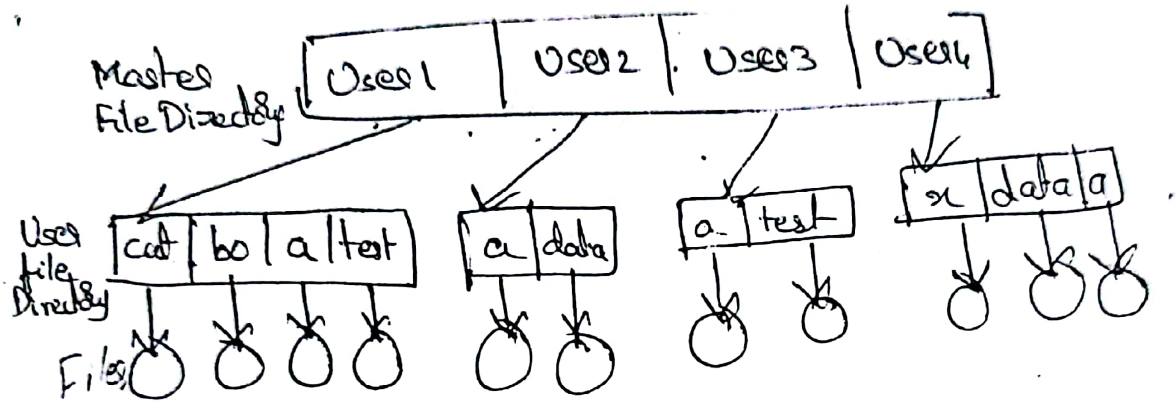
4) It leads to confusion of filenames among different users.

Scanned by CamScanner

(2) 2-level Directory :-

Since single level directory creates confusion among users with file names

The standard sol is to create a separate directory for each user



→ In this, each user has his own user file directory (UFD)

→ all UFD have similar structure, but each list only the files of a single user

→ when user logs in, the sys's MFD is searched.

→ MFD is indexed by user name (or) acc no & each entry points to UFD of that user

→ To create a file for a user, OS searches UFD to ascertain whether another file of that name exists.

Scanned by CamScanner

→ To delete a file, OS search local UFD (?)

thus, it can not accidentally delete another user's file that has the same name.

→ Although this schema resolves naming collision prob it still has disadv

① ⇒ Isolates one user from another.

② ⇒ If access is permitted, then the user must have the ability to name a file in another user's directory.

→ In this schema, to name a file, we must give → username
→ filename.

⇒ 2-level directory can be thought as a Tree.

Root — MFD

Its descendent — UFD's

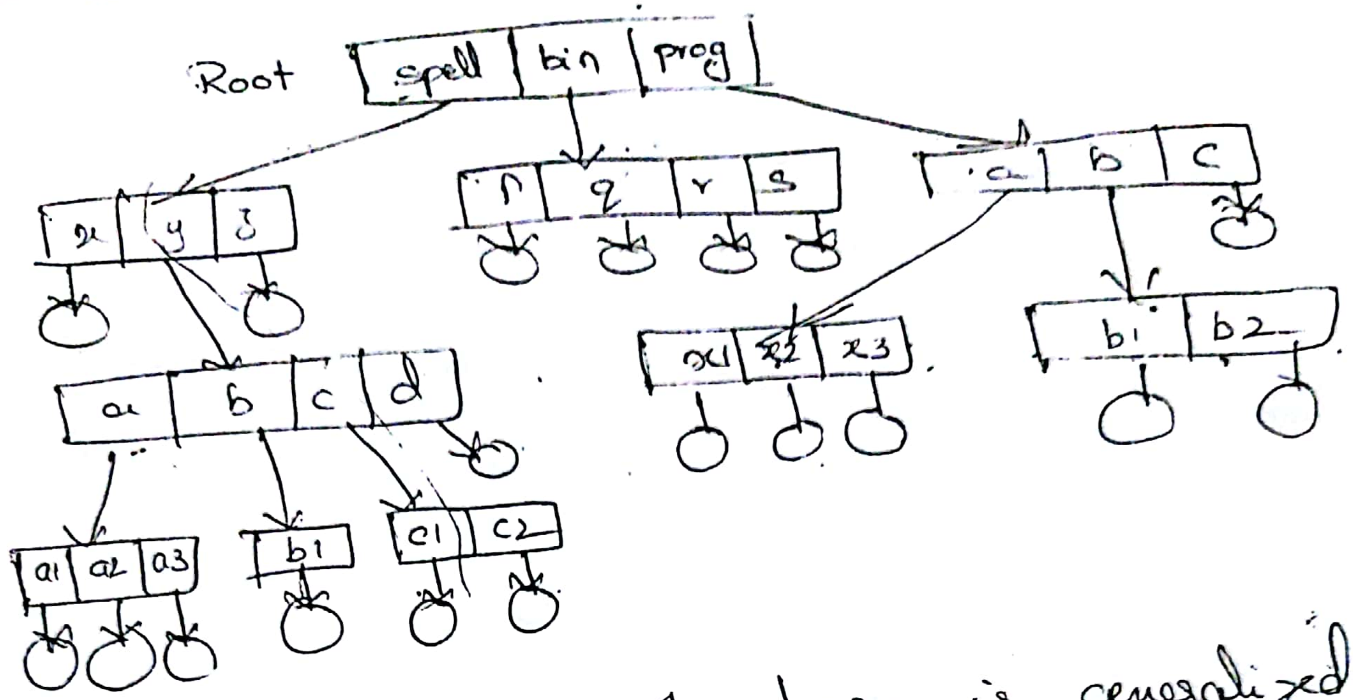
UFD's descendent — Files i.e leaves.

→ path name = username + filename

→ Every file has a pathname.

Ex:- /userB/test.

(3) Tree Structured Directory.



→ 2-level Directory structure is generalized to 2-level tree.

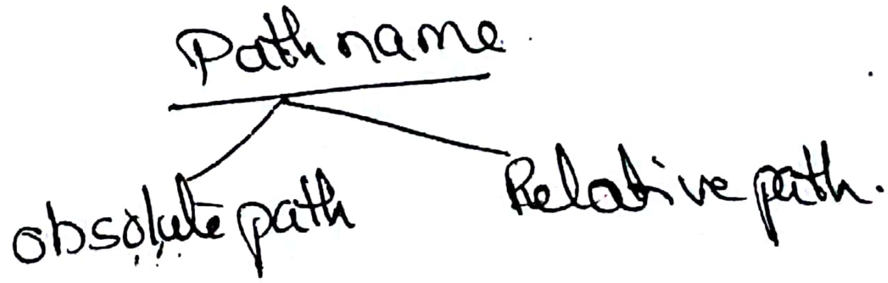
→ This generalization allows the user to create his own subdirectories.

→ A Tree is most common Directory structure.

→ A tree has a root directory

→ Every file in the system has unique pathname

→ Path names can be of 2 types.



→ An absolute path name begins with (root) & follow a path down to specified file

→ Relative path name defines path from the current directory

Ex: root/spell/mail/copy, is the absolute path.
Then C/C1 is the relative path.

→ Current directory should contain files that are of interest to the process.

→ when reference is made to a file, the current directory is searched.

→ How to handle deletion of a directory?

→ If a directory is empty, its entry in the directory that contains it can simply be deleted.

→ Suppose a directory is not empty, but contains files & sub directory, to delete it we use 2 approach.

1st approach: MS DOS will not delete the directory unless it is empty. User must 1st delete all files in that directory.

Disadv:-

Results in more amt of work.

2nd approach: UNIX rm command.

when a request is made to delete a directory all that directories files & subdirectories are also deleted.

Disadv:- More dangerous - bcoz entire directory structure is removed with one command.

→ with a tree structured directory sys users can be allowed to access, in addition to their files, the files of other users.

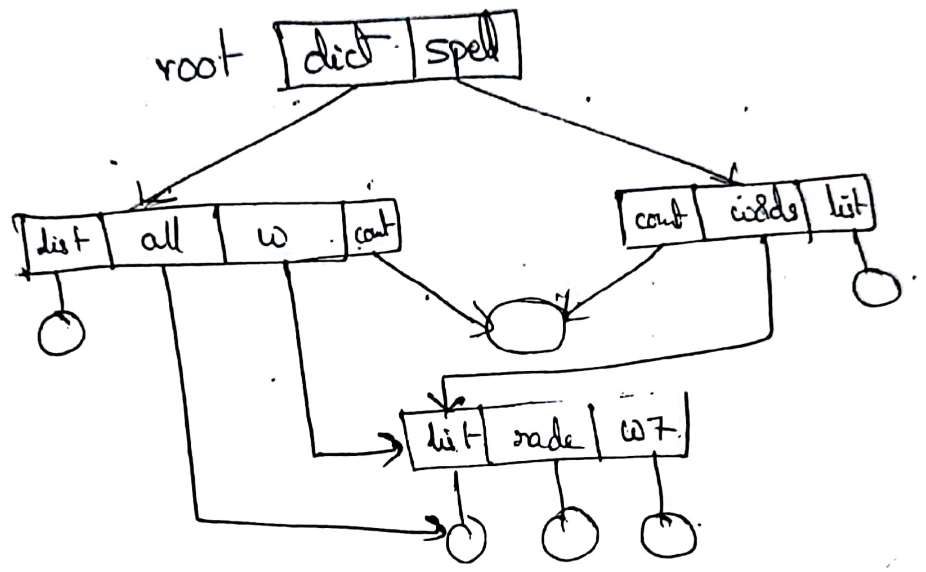
→ Path to a file in tree structured directory is longer than a path in a 2-level directory.

4) Acyclic Graph Directories:

→ Tree structure prohibits the sharing of files (or) Directories

→ Acyclic-graph is a graph with no cycle

→ this allows directories to share subdirectories & files



→ same file may be in 2 different directories

→ acyclic graph is a natural generalization of tree-structured directory scheme

→ It is imp to note that a shared file is not the same as 2 copies of the file.

→ with 2 copies

each programmer can view the copy rather than the original.

But if one programmer changes the file, the change will not appear in the other copy.

→ with shared file

Only one file exists, so any change made by one person are immediately visible to the other

⇒ shared files can be implemented in several ways.

I ① Create new entry called a link in directory

② Link is pointer to another file (or) subdirectory.

③ when a reference to a file is made, we search dir, if dir entry is marked as link, then the name of the real file is included in the link information.

④ we resolve the link by path name to locate the real file.

II
→ Simply duplicate all information about them in both sharing directories.

→ Thus both entries are identical & equal

prob — Maintain consistency when a file is modified

⇒ Prob of this scheme

→ shared structures are traversed more than once.

→ Another prob is deletion of file

If anyone deletes a file just remove it but this leads to a prob i.e pointer ~~with~~ will point to an non-existing file

If the deleted space is reused by other file then the pointer will point to the middle of the some other file

→ Deleting link:

→ In a sys where sharing is implemented by symbolic links, this situation is somewhat easier to handle.

→ Deletion of a link need not effect the original file, only the link is removed.

(5) General Graph Directory:

→ A serious prob with acyclic-graph structure is there are no cycles

→ Adding new files & subdirectories to an existing tree structured directory preserves tree structure nature

→ Adding links to an existing tree structured directory, destroyes tree structures resulting in a simple graph structure.

→ In acyclic shared files are traverse ...

→ If cycles are allowed to exist in the directory, we can avoid searching for components twice.

→ With acyclic graph directory structure value 0 in the reference count means no more reference to the file (or) directory & the file can be deleted.

→ When cycle exist reference count may not be 0 even when it is not refered.

→ This is due to the cycle in the directory structure.

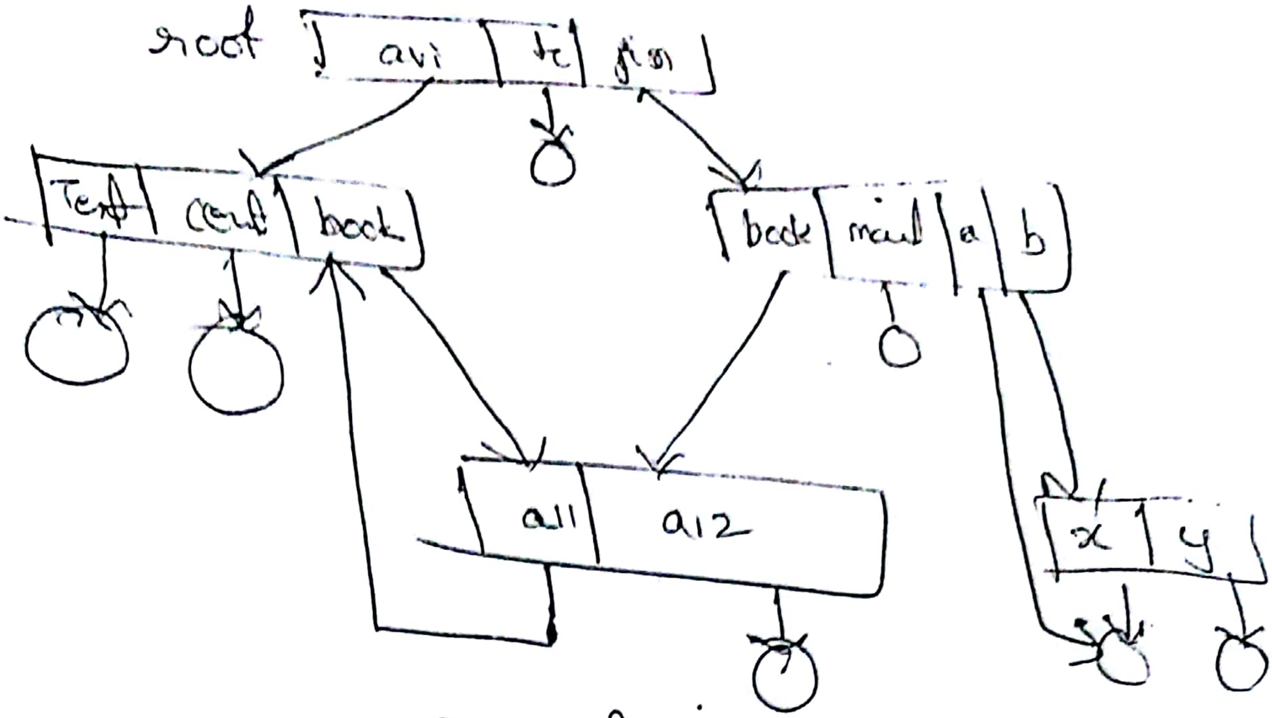
In this case we have to use garbage collection scheme.

→ If last reference has been deleted it is pushed to garbage collection so that the disk space can be reallocated.

→ This is time consuming & thus seldom attempt.

→ Thus a acyclic-graph structure is much easier to work with.

Fig



General graph directory.

File System Implementation:-

→ several ondisk & in memory structures are used to implement a file system.

→ (On disk) the FS contains

1) Boot control block (per volume)

contain info needed by the sys to boot an OS. In UFS, it is called the boot block

2) Volume control block (per volume)

contains volume details such as no of blocks, size of the block, free block count etc. In UFS, it is called super block.

3) Directory structure per FS is used to organize the fi

4) File control block per file contain many details about the file.

Scanned by CamScanner

In-memory information include

1. In-memory mount table — info about mounted volumes
2. In-memory Directory structure cache — dir info of recently accessed directories
3. System wide open file table — contains a copy of FCB of each open file as well as, other info
4. Per-process open file table — contains pointer to entry in the system-wide open file table as well as other info

→ To create new file, it (logical FS) allocates a new FCB.

A Typical FCB

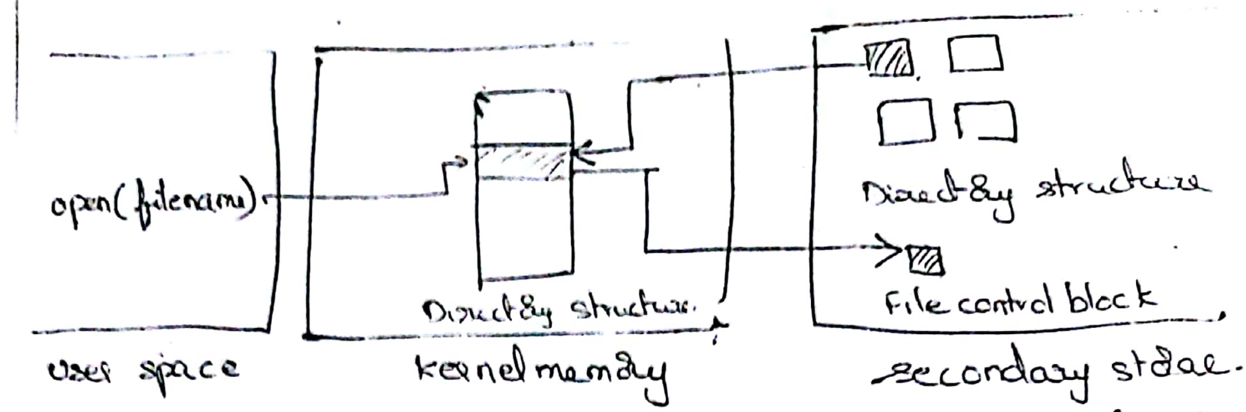
File permissions
file dates (create, access, write)
file owner, group, ACL
file size
file data blocks (or) pointers to file data blocks

→ The sys then reads the dir to the memory update it with new file & FCB & write it back to the disk

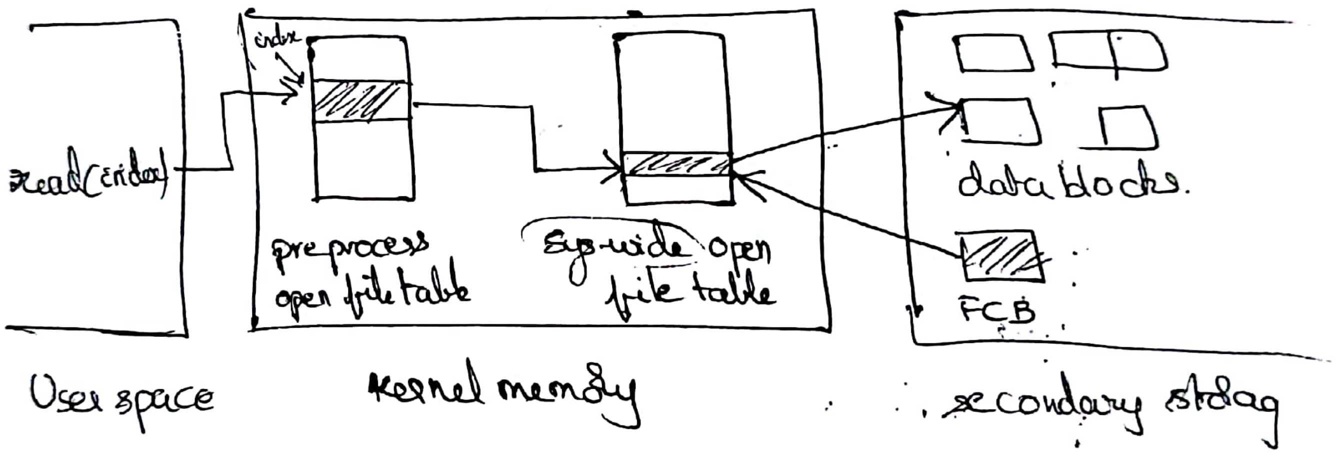
→ Now that a file has been created, it must be opened.

→ open() call passes a filename to FS.

In memory FS structures

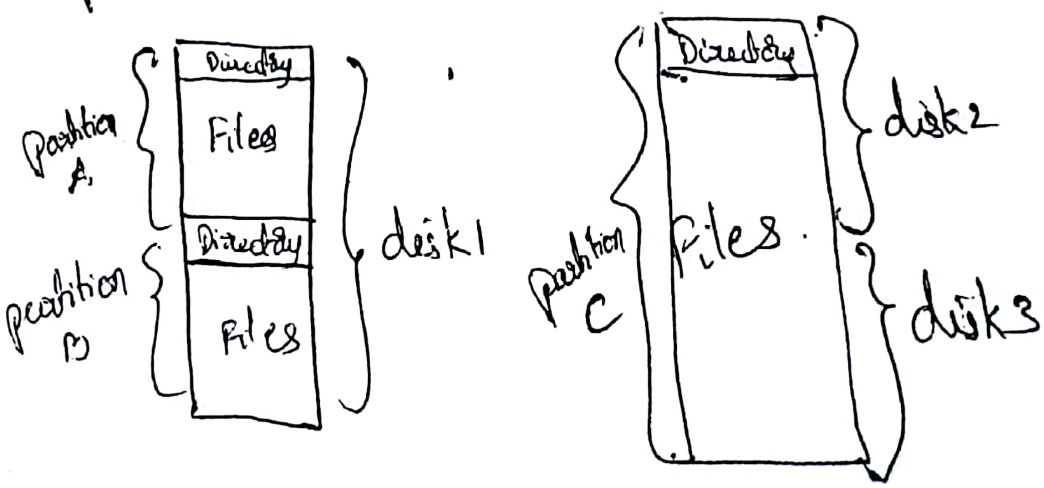


The open() searches sys-wide-open-file table to see if the file is already in use by another process
→ if it is YES pre-process open file table entry is created pointing to existing sys wide open file table



Partitions & Mounting :-

A disk can be sliced into multiple partitions(or)volume can span multiple partitions on multiple disks.



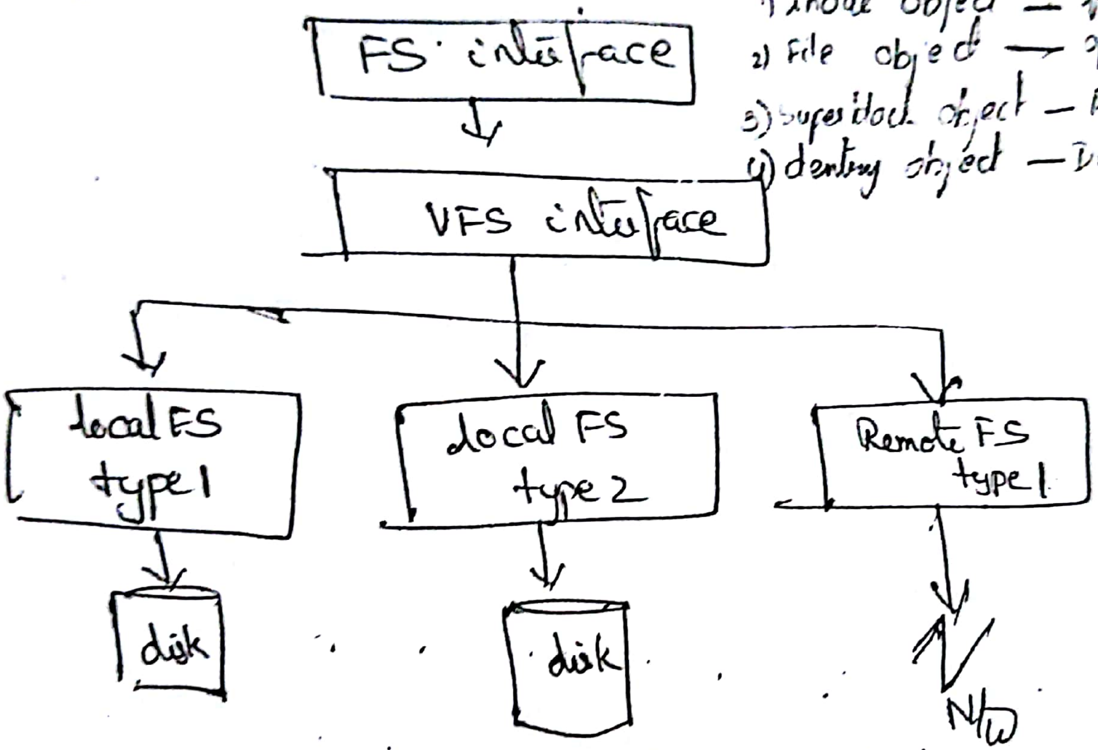
rc FS (or) cooked containing FS

→ Boot info can be stored in a separate partition

→ Root partition — contains OS kernel sometimes other system files — is mounted at boot time

→ On Unix, file sys can be mounted at any directory.

Virtual FS :-



unix VFS defines 4 object
1) inode object — file
2) File object — open file
3) superblock object — FS
4) dentry object — Directory entry

→ File sys implementation consist of 3 major layers

➡ 1st layer — FS interface based on open(), read(), write() & close() calls & file descriptors.

➡ 2nd layer — Virtual FS layer its functions are

a) It separates FS generic operations from their impl by defining VFS interface.

(b) VFS provides mechanism for uniquely represent a file throughout a N/w
VFS is based on vnode —
where Vnode is a file representation structure that contains a numerical value for N/w wide unique file.

→ thus VFS distinguishes local files from remote ones.

→ VFS distinguishes local files according to their types

- <u>3rd layer</u>: The layer implementing the FS type (or) remote FS protocol is the 3rd layer of the architecture.

<u>Directory Implementation</u>:

① <u>Linear list</u>:

→ Simplest method of implementing a directory is to use a linear list of file names with pointers to the data blocks.

→ this is simple to program but time consuming to execute.

→ <u>Creating file</u>:

1st search Directory - If there is no file with same name then add new entry at the end of the directory.

1 → Deleting file:

Search the Directory for name file

Then release the space allocate to it

→ Reuse Directory

→ Mark the entry as unused.

→ Attach it to a list of free directory entries

→ copy the last entry in directory into the freed location & to decrease the length of the directory.

→ Disadv.

→ Finding a file requires linear search.

→ Hence access to file is slow.

① → To over come - use cache.

→ cache stores most recently used directory info

→ It avoids reread the info from disk.

② → Using sorted list.

→ Finding a file requires binary search.

→ Hence decreases the avg search time

② Hash table

→ Another Data structure used for a file Directory is hash table.

→ for this - linear list is used to store directory entries

→ Along with it hash DS is used.

→ Hash table takes a value computed from file name & return a pointer to the file name in the linear list.

→ It decreases search time
→ Insertion & deletion are straight forward
→ provision is made for collisions – when 2 filename hash same location
→ chained-overflow hash table can be used. Each hash entry can be a linked list instead of an individual value

Disadv
- Hash table – fixed size
- Dependent on hash function

5) Unix File – Suhasini Material
- User's view of file system.
- Different types of files
  - Ordinary Files
    - Text files
    - Binary files
  - Directory files inode, ...
  - special files – I/O devices
  - FIFO files.
- Mounting / unmounting File systems.
- Important UNIX directories / Files.

a) User's view of FS :-
→ UNIX implements a hierarchical FS. FS starts with root directory at the top of the inverted tree.
⇒ Root directory contains a no. of directories which in turn contains a no. of files / subdirectories & so on.

② Swathi ①

UNIT - V

DEAD LOCK

Definition :— In a multiprogramming environment several processes may compete for a finite number of resources.

* A process requests resources, if the resources are not available at that time, the process enters a waiting state.
* Sometimes a waiting process is never again able to change state because the resources it has requested are held by other waiting processes. This situation is called as "Dead lock".

[OR]

Definition (2) :— A process request the resources.

* The resources are not available at that time, so the process enters into a waiting state.
* The requesting resources are held by another waiting process, both are waiting state this situation is called as "Dead lock".

P_1 and P_2 are two processes.

R_1 and R_2 are two [illegible]

P_1 Request the resource R_1, R_1 held by process P_2.

P_2 Request the resource R_2, R_2 held by process P_1. Then both are entered into the waiting state.

There is no work progress for process P_1 and P_2 it is also called as Dead lock.

System Model :—

A system consists of a finite no. of resources to be distributed, among a [number] of competing processes.

* Resources are partitioned into several types, each consisting of some no. of identical instances.
* Memory space, CPU cycles, files & I/O devices (printers, DVD drives) are examples of resource types.
* [illegible] then the resource type CPU has two instances

- If a process requests an instance of a resource type, the allocation of any instance of that type will satisfy the request.
- If it will not then the instances are not identical and the resource type classes have not been defined properly.
- For example, a system may have two printers. These two printers may be defined to be in the same resources class, no one cares which printer prints which output.
- A process must request a resource before using it and must release the resource after using it.
- A process may request as many resources as it requires to carry out its designated tasks.
- The no. of resources requested may not exceed the total no. of resources available in the system.
- A process cannot request 3 printers if the system has only 2.

Under the normal mode of operation a process may utilize a resource in only the following sequence.

1) Request :— If the request cannot be granted immediately (for example, if the resource is being used by another process), then the requesting process must wait until it can acquire the resource. (For example, the resource ... other process. It is done by using system calls such as allocate( ) for memory, open( ) for files, request( ) for devices.

2) Use :— The process can operate on the resource. (For example if the resource is a printer, the process can print on the printer.

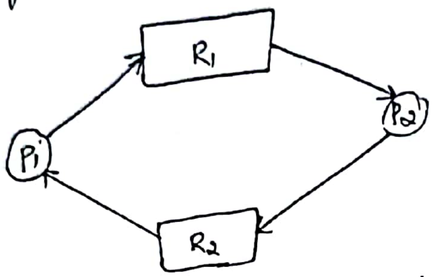
3) Release :— The process releases the resource.

The request and release of resources are system call. Examples are request and release device, open( ), close( ), file and allocate( ) free( ) memory system calls.

A set of processes is in a deadlock state when every process in the set is waiting for an event that can be caused by only another process in the set. The events with which we are mainly concerned here are resource acquisition and release. The resources may be either physical resources (for eg: printers, tape drivers, memory space ...) or logical resources (for eg: files, semaphores & monitors).

Resource Allocation Graph :--

* A resource allocation graph is a directed graph it is used to represent the deadlocks.
* The graph consisting of two types of nodes one for process it is represented by circles and second is for resources it is represented by squares.
* Graph consisting of two types of edges.. One is requesting edge and another one is assignment edge.



RESOURCE ALLOCATION GRAPH.

An edge from process to resource is said to be a requesting edge and an edge from a resource to a process is said to be an assignment edge.

DEADLOCK CHARECTERISATION :- In a deadlock, processes never finish executing and ... are tied up, preventing other jobs from starting.

Necessary Conditions :- Suppose the following ... hold regarding the way a process uses resources.

- Mutual Exclusion
- Hold & wait
- No preemption
- Circular wait.

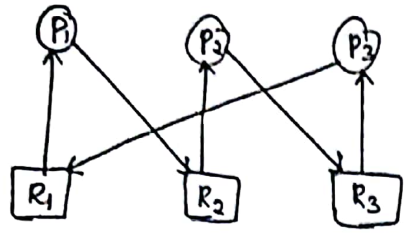
Mutual Exclusion :- Only one process may use a resource at a time. Once a process has been allocated a particular resource, it has exclusive use of the resource. No other process can use a resource while it is allocated to a process.

Hold & wait :- A process may hold a resource at the same time, it requests another one.

No preemption :- No resource can be forcibly removed from a process holding it. Resource can be released only by the explicit action of the process, rather than by the action of an external authority.

Scanned by CamScanner

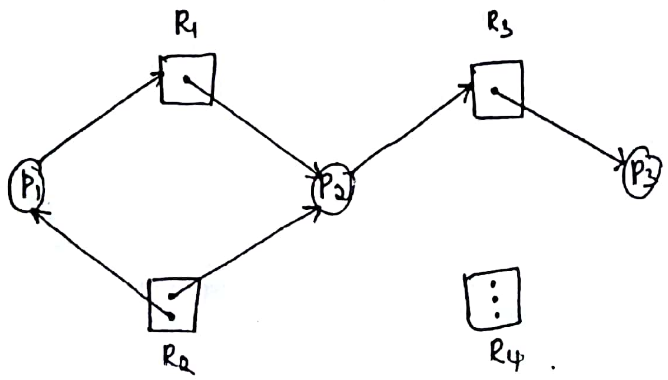
- **Circular Waiting** :— A situation can arise in which process P_1 holds resource R_1 while it requests resource R_2 and process P_2 holds R_2 while it requests resource R_1.
- Each process holds at least one resource needed by the next process in the chain.
- There may be more than two processes involved in a circular wait.



- A deadlock is possible only if all 4 of these conditions simultaneously hold in the community of processes. These conditions are necessary for a deadlock to exist.

Resource – Allocation Graph :— It is used to describe the (3) deadlock. It is also called system resource allocation graph.

- Graph consists of a set of vertices (V) & set of edges (E).
- All the active processes in the system denoted by $P = \{P_1, P_2, P_3, \ldots P_n\}$.
- Set of all resources type in the system is denoted by $R = \{R_1, R_2, R_3 \ldots R_n\}$.
- Request edge is an edge [illegible] to resource & denoted by $P_i \rightarrow R_j$.
- An assignment edge is an edge from resource to process & it is denoted by
- Holding of resource by process is denoted by assignment edge.
- Requesting of resource by process is denoted by request edge.
- For representing process and resource in the resource allocation graph is shown square & circle.
- Each process is represented by circle & resource by square.
- Dot within the square represents the number of instance.



RESOURCE ALLOCATION GRAPH.

Scanned by CamScanner

- System consists of 3 processes i.e., P_1, P_2 and P_3. and 4 resources i.e. R_1, R_2, R_3 and R_4.
- The set P, R and E consists

$$P = \{ P_1, P_2, P_3 \}$$
$$R = \{ R_1, R_2, R_3, R_4 \}$$
$$E = \{ P_1 \rightarrow R_1,\ P_2 \rightarrow R_3,\ R_1 \rightarrow P_2,\ R_2 \rightarrow P_2,\ R_2 \rightarrow P_1,\ R_3 \rightarrow P_3 \}$$

- **Resource instances :-**

Resource R_1 = one instance
" R_2 = two "
" R_3 = one "
" R_4 = three " .

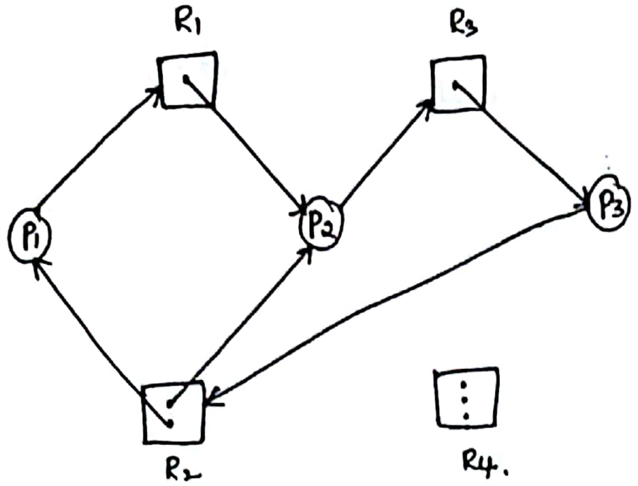
Process states :-

Process P_1 is holding instance of resource type R_2 and is waiting for an instance of resource type R_1.
Process P_2 is holding an instance of R_1 & R_2 & waiting for an instance of resource type R_3.
Process P_3 is holding an instance of R_3.
If the graph contains no cycles then no process in the system is deadlock. If the graph does not contain a cycle then a deadlock may exist.
Suppose that process P_3 requests an instance of resource type R_2. Since no resource instance is currently available a request edge $P_3 \rightarrow P_2$ is added to the graph. At this point two minimal cycles exist. Resource allocation graph with deadlock.
in the system.

$$P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$$

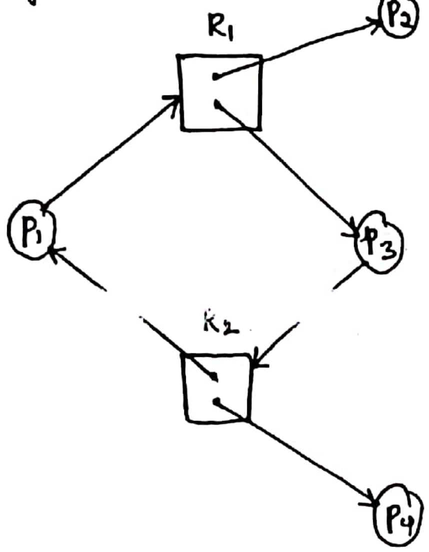
$$P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$$

Process P_1, P_2, P_3 are dead locked.



RESOURCE ALLOCATION GRAPH WITH DEAD LOCK.

* Process P_2 is waiting for the resource R_3, which is held by process P_3.
* Process P_3 is waiting for either process P_1 (or) process P_2 to release resource R
* Process P_1 is waiting for process P_2 to release resource R_1. Resource-allocation graph with a cycle but no dead lock is shown in fig.



RESOURCE ALLOCATION GRAPH WITH A CYCLE BUT NO DEAD LOCK.

* Let us consider ~~graphs~~

$$P_1 \longrightarrow R_1 \longrightarrow P_3 \longrightarrow R_2 \longrightarrow P_1 .$$

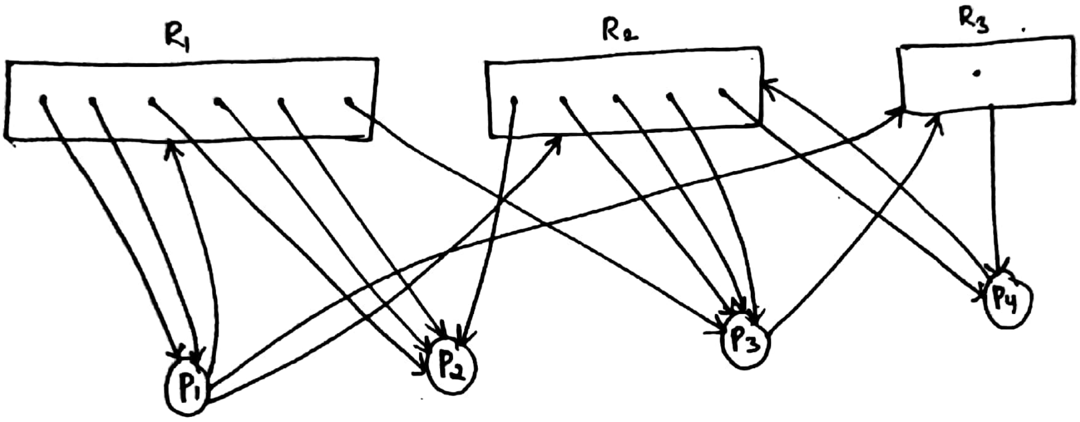
* There is a cycle but no dead lock. Because the process P_4 may release its instance of resource type R_2.

~~That resource is instance of resource type R2.~~

That resource can be allocated to P_3 breaking the cycle.

Eg:- Given the process resource usage & availability. Draw resource-allocation graph.

Process	Current Allocation			Outstanding requests			Available Resource		
	R_1	R_2	R_3	R_1	R_2	R_3	R_1	R_2	R_3
P_1	2	0	0	1	1	1	0	0	0
P_2	3	1	0	0	0	0			
P_3	1	3	0	0	0	1			
P_4	0	1	1	0	1	0			



R_1 = 6 instances

R_2 = 5 instances

R_3 = 1 instance.

DEADLOCK PREVENTION :-

Methods for preventing deadlock are two classes.

1. Indirect method.
2. Direct method.

- An indirect method is to prevent the occurrence of one of three necessary conditions i.e., mutual exclusion, hold & wait, and no preemption.
- Direct method is to ~~prevent~~ prevent the occurrence of a circular wait.
- The general meaning for prevention is take the medicine before the attack of disease.
- Deadlock prevention is same as take the preventive methods before attacking dead lock.

Mutual Exclusion :— It means only one process can use the resource at time. It means the resources are not shared by the no. of processes at a time. So the resources are non-shareable mode only.

- It must be supported by the Operating System.
- Resources such as files, may allow multiple accesses for reads but only exclusive access for writes.
- A printer is not shared by the no. of process at a time. So we can't convert a printer from non shareable to shareable mode.
- So we can't apply this prevention method if the system consisting of printers.

Hold & wait :— In this case & every processes in the deadlock state, must hold atleast one resource & waiting for atleast one resource.

- A process request the resources only when the process as none (protocol).
- This protocol is very expansive & time consuming.
- For example process wants to copy the data from a tape drive to a disk.
- Initially the process consisting of tape drive, disk file.
- Now the process to be request the printer.

Applying the above protocol the process should release the tape drive & disk file before the request of printer. So, it is time consuming.

- The second protocol is each process to request and be allocated all its resources before it being execution.
- It is very difficult to implement because more no. of resources are available before the execution.
- For example $P_1, P_2, P_3 \ldots P_{100}$ are 100 processes.
- Each requires a printer to complete their jobs. Applying this protocol the system must consisting of 100 printers. So, it is very difficult to implement.
- There are two main disadvantages to these protocols. First resource utilization is very poor.
- Second starvation is possible.

No preemption :— A process request some resources we first check whether are available. If they are available we allocate them.

- If they are not available we check whether they are allocated to some process that is waiting for additional resources.

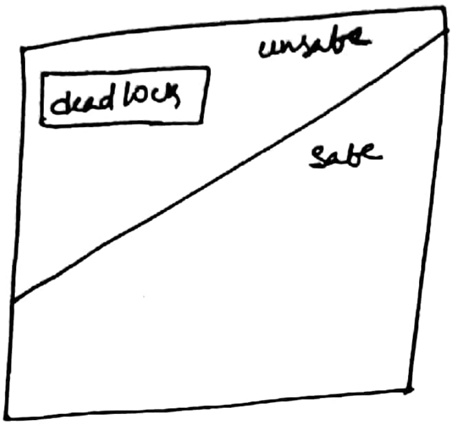
If so, we preempt the desired resource from the waiting process &
allocate them to the requesting process.
If resources are not either available (or) held by the waiting process, then
the requesting process must wait.
While it is waiting, some of its resource may be used by other process.

Circular Wait :— We can ensure that circular must not happened if we
apply a simple solution i.e., numbering all the resources type and each
process request resources in an increasing order of enumeration.
Whenever a process requests an instance of resource be R_J. It has
released any resources R_i such that $F(R_i) > F(R_J)$ and the second protocol is
"A process can initially request any no. of instances of a resource type say R_i
note that the process can request the instances of resource type R_J. (If and only
iff $F(R_J) > F(R_i)$.
If these two protocols are used then the circular wait condition cannot hold.

DEADLOCK AVOIDANCE :— Avoiding deadlock is to require additional
information about how resources are to be requested. lead to
To construct an algorithm that ensures that the system will never enter
state. such an algorithm defines the deadlock avoidance approach.
It is dynamically examines the resource-allocation state to ensure that a
circular wait condition can never exist.
The resource allocation state is defined by the no. of available & allocated
resources & the maximum demands of the processes.

Safe State :— A state is a safe if the system can allocate resources
to each process to avoid a deadlock.
If the system is in safe state only if there exists a safe sequence.
A sequence of processes $\langle P_1, P_2, P_3 \cdots P_n \rangle$ is a safe sequence.
The resources requests that P_i can still make can be satisfied by the current
available resources plus the resources held by all P_J with $J<i$.
In this situation if the resources that P_i needs are not immediately available
then P_i can wait until all P_J has finished.

- The state is not a deadlock state.
- Conversely a deadlock state is a unsafe state.
- Not all unsafe state are deadlock as unsafe state may be lead to a deadlock.
- As long as the state is safe the operating system can avoid unsafe state.
- In an unsafe state the O.S cannot prevent processes from requesting resources such as deadlock occurs.



Safe, Unsafe & deadlock state spaces.

Ex:- We consider a system with 12 magnetic tape drivers. and three process P0, P1, P2.
Process P0 requires 10 tape drivers, process P1 may [illegible] 4 tape drivers, P2 may need upto 9 tape drivers.

processes	Maximum Needs	Current Needs	Available
			3
P0	10	5	
P1	4	2	
P2	9	2	
		9	

- At time t0 process P0 is holding 5 tape drivers, process P1 is holding 2 tape drivers and process P2 is holding 2 tape drivers. It is a safe state
- The sequence <P1, P0, P2> satisfies the safety conditions.
- Process P1 can immediately be allocated all its tape drivers & then return them. Then process P0 can get all its tape drivers & return them.
- Finally P2 can get all its tape drivers & return them.

finally P_2 can get all its tape drivers & return them.

→ System can go from a safe state to an unsafe state.

Suppose at time t_1, process P_2 requests and 's' allocated as one more type drive. the system is no longer in safe state.

Process	Maximum Needs	Current Needs	Available
P_0	10	5	2
P_1	4	2	
P_2	9	3	

Process P_1 can be allocated all its tape drivers. The system will have only 4 available tape drivers.

Process P_0 is allocated 5 tape drivers but has a maximum 10, it may request 5 more tape drivers since they are unavailable Process P_0 must wait.

Similarly process P may request an additional 6 tape drivers & have to wait resulting in a dead lock.

Our mistake was in granting the request from process P_2 for one more tape drivers.

If we had made P_2 wait until either of the other processes had finished & released its resources then we could have avoid the deadlock.

Resource - Allocation graph Algorithm

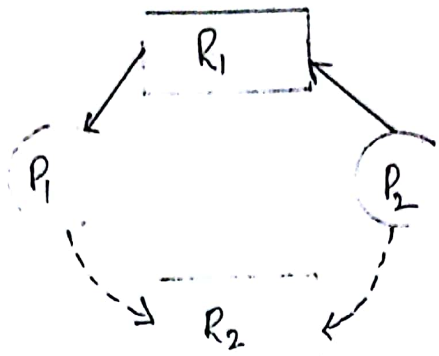
→ Resource allocation graph is used for deadlock avoidance

→ In resource allocation graph have the request & assignme edges

→ In this we introduce a new type of edge called a claim ed

→ A claim edge P_i - R_j indicates that process P_i may request resource R_j at some time in the future.

→ Ci & it is represented in the graph by a dashed line



Resource - allocation graph for dead lock avoidance

→ when process P_i request resource R_j the claim edge is conv converted to a request edge

→ when a resource R_j is released by P_i the assignment edge $R_j \dashrightarrow P_i$ is reconverted to a claim edge $P_i \rightarrow R_j$

→ cycle detection algorithm is used for detecting cycle in the graph

→ An algorithm for detecting a cycle in this graph requires an order of n^2 operations. where n is the number of processes in the system.

→ If no cycle exists, then the allocation of the resource will lea the system in a safe state.

→ If a cycle is found then the allocation puts the system is c unsafe state.

<u>Banker's Algorithm</u>:-

→ The Bankers algorithm is the best known of the avoidan strategies

→ The resource allocation graph algorithm is not applic to a resource allocation system with multiple instances each resource type

Scanned by CamScanner

→ The deadlock avoidance algorithm is applicable because it is less efficient than the resource allocation graph. This algorithm is commonly knowns as the banker's algorithm.

→ The name was choosen because the algorithm could be used in a banking system.

→ when a new process enters the system it must declare the maximum numbers of instances of each resource type that is may need.

→ when a user requests resources the system must determine whether the allocation of these resources will leave the system in a safe state

→ Otherwise the process must wait until ~~some~~ Some other process releases enough resources.

→ Several data structures must be implement the banker's algorithm.

→ Let n be the number of processes in the system & m be the numbers of resources type in the system.

→ Some following data structures.

a) Available:- A vector of length m indicates the numbes of available resources of each type.

If available [j] = k, there are k instances of resource type R_j available

b) Max:- An n×m matrix defines the maximum demand of each process. If max [i,j] = k then process P_i may request at most 'k' instances of resource type R_j.

) Allocation: An n×m matrix defines the numbes of resources each type currently allocated to each process. If allocation [i,j]=k, then process P_i is currently allocated k instances of resource type R_j

d) **Need**:- An n×m matrix indicates the remaining resources need of each process.

If need [i,j]=k, then process P_i may need 'k' more instances of resource type R_j to complete its task need [i,j] = Allocation [i,j]

→ we can treat each row in the matrices Allocation & need as vectors & refer to them as Allocation$_i$ & need$_i$

→ The vector allocation$_i$ specifies the resource currently allocated to process P_i

→ The vector need$_i$ specifies the additional the resources that process P_i may still request to complete its task.

Safety algorithm:-

The algorithm for finding out whether the system is in a safe state or not.

→ The algorithm can be described as follow.

1) Let work & finish be vector of length m & n respectively.
Initialise work = Available

finish [i] = false.

for i = 1, 2, 3 . . . . .

2) find an 'i' such that

a) finish [i] = false

b) Need [i] ≤ work.

if no such 'i' exist go to step 4.

3) work = work + Allocation;
finish [i] = true
goto step 2

4) If finish [i] = true for all i then the system is in a safe state.

Resource - request algorithm :-

→ Let $request_i$ be the request vector for process P

→ If $request_i [J] = K$, then process P_i wants 'K' instances of resource type R_J.

→ when a request for resources is made by process P_i the following actions are taken.

1) If $Request_i \leq Need_i$ then go to step 2. Otherwise raise an error condition, since the process has exceeded its maximum claim.

2) If $Request_i \leq Available$ then go to step 3 otherwise P_i must wait, since the resource are not available.

3) $Available := Available - Request_i;$

$Allocation_i := Allocation_i + Request_i;$

$Need_i := Need_i - Request_i;$

→ If the resulting resource - allocation state is a safe, the transaction is completed & process P_i is allocated its resource

→ If the new state is unsafe, then P_i must wait for the $request_i$ & the old resource - allocation state is restored

Examples on Bankers algorithm:

→ consider a system with five processes P_0 through P_4 & three resources type A, B & C.

→ Resource type A has 10 instances, resource B has 5 instances & resource C has 7 instances

→ Suppose that a time. to the following snap shot of the system has been taken.

Examples on Bankers algorithm:-

Process	Allocation			Available			Available		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	5	3	3	3	2
P_1	2	0	0	3	2	2			
P_2	3	0	2	9	0	2			
P_3	2	1	1	2	2	2			
P_4	0	0	2	4	3	3			

→ content of the matrix needed is defined as

Need = Max − Allocation

Process	Need A	Need B	Need C
P_0	7	4	3
P_1	1	2	2
P_2	6	0	0
P_3	0	1	1
P_4	4	3	1

Currently the system is an a safe state.

Safe Sequence:- Safe sequence is calculated as follows:-

1) Need of each process is compared with available
→ If $need_i \leq available_i$, then the resource are allocated to that process.

2) If need is greater than available, next process need is taken for comparison.

3) In the above example, need of process P_1 is (7,4,3) & availabl is (3,3,2)

need ≤ available (or) work

(7,4,2) ≰ (3,3,2) → false

So system will move for next process.

4) Need for process P_2 is (1,2,2) & available (3,3,2) so

need ≤ work (or) available

(1,2,2) ≤ (3,3,2) – True

then finish [i] : True

→ Request P_2 is granted & process P_2 is release the resource to the system.

work := work + Allocation

work := (3,3,2) + (2,0,0)

= (5,3,2)

This procedure is continued for all processes.

5) Next process P_3 need (6,0,0) is compared with new available (5,3,2)

need > Available (or) work = false

(6,0,0) > (5,3,2)

6) process P_4 need (0,1,1) is compared with available (5,3,2)

need < work.

(0,1,1) < (5,3,2) = True

work = work + Allocation

= (5 3 2) + (2 1 1)

= (7 4 3) (new available

7) Then process P_5 need (4 3 1) is compared with available. (7,4,3)

need < work

(4 3 1) < (7 4 3) = True

work = work + Allocation

= (7 4 3) + (0 0 2)

= (7,4,5) new available

8) The process P_1 need (7 4 3) is compared with available (7,4,5)

need ≤ work.

(7 4 3) ≤ (7 4 5) = work + Allocation

= (7 4 3) + (0 1 0)

(7 5 3) → new available

(i) Process P_3 need is (6,0,0) and available (7 5 3)

need ≤ work.

(600) ≤ (753)

work = work + Allocation

= (6 00) + (30 2)

= (902)

Safe sequence <P_2, P_4, P_5, P_1, P_3>

→ Suppose now that process P_1 requests one addition instance of resource type A & two instance of resource type C, so request = (1,0,2) · Request ≤ need.

(1,0,2) ≤ (7,4,3)

→ we first check the Request$_1$ ≤ Available i.e

(1,0,2) ≤ (3,3,2) → true.

Available = Available − Request$_1$

(3,3,2) − (1,0,2)

= (2,3,0)

Allocation = Allocation + Request

= (2,0,0) + (1,0,2)

= (3,0,2)

Need = (7,4,3) − (1,0,2)

= (6,4,1)

	Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	6	4	1	2	3	0
P_1	3	0	2	0	2	0			
P_2	3	0	2	6	0	0			
P_3	2	1	1	0	1	1			
P_4	0	0	2	4	3	1			

Deadlock Detection:

→ If the system is not using any deadlock avoidance & dead prevention then a deadlock situation may occur.

→ Deadlock detection approach do not limit resource access (or) restrict process actions.

→ With deadlock detection request resources are granted to processes whenever possible

→ OS performs an algorithm that allows it to detect the circular wait condition.

Single instance of each resource type:-

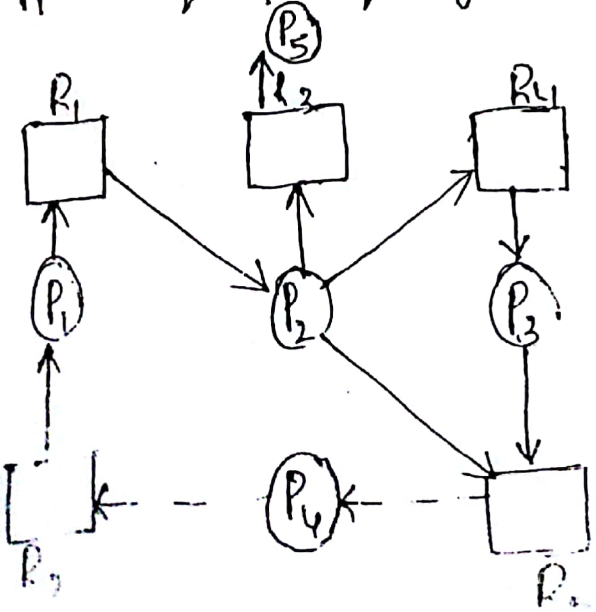
→ If all resources have only a single instance then deadlock detection algorithm uses the resource-allocation graph called a wait-for graph.

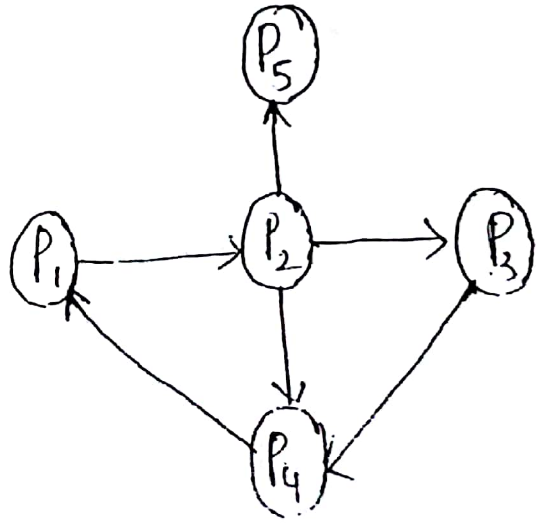
→ wait for graph can be drawn from the resource-allocation graph by removing the resource nodes & collapsing the appropriate edge.

→ An edge from P_i to P_j in a wait for graph implies that process P_i is waiting for the process P_j to release a resource that P_i needs.

→ An edge $P_i \to P_j$ exists in a wait-for graph & if the corresponding resource allocation graph contain two edges.

$P_i \to R_q$ & $R_q \to P_j$ for some resource R_q





→ A deadlock exists in the system only if the wait-for-graph contains a cycle

→ An algorithm to detect a cycle in a graph requires an order of n^2 operation.

where 'n' is the no. of vertices in the graph.

Several instances of a Resource type:-

→ The wait-for graph scheme is not applicable to a resource allocation system with multiple instances of each resource type

→ To overcome this we are using algorithm employs several time varying data structures that are similar to those used in the banker's algorithm

Eg:- consider a system with five processes P_0 through P_4 & three resource type A, B, C. Resource A has 7 instances Resource B has 2 instances, Resource C has six.

	Allocation			Request			Available		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	0	0	0	0	0	0
P_1	2	0	0	2	0	2			
P_2	3	0	3	0	0	0			
P_3	2	1	1	1	0	0			
P_4	0	0	2	0	0	2			

$< P_0, P_2, P_3, P_4, P_1 >$

→ Suppose now that process P_2 makes one additional request for an instance of type 'C'.

	Request		
P_0	0	0	0
P_1	2	0	2
	0	0	1
		0	0

→ resource is in dead locked state

Recovery from Dead lock:-

-> There are two methods for breaking a dead lock. one [illegible] simply to abort one by one processes to break the circular wait

-> The second option is, to preempt some resources from one (or) of the dead locked process.

Process termination:- It is one method to recover from deadlock, we use one of two methods for process termination. these are

i) **Abort all dead locked processes:-** It means release the all processes in the deadlocked state & start the allocation from the starting point. It is a great expensive..

(ii) **Abort one by one process until the deadlock cycle is eliminated**

-> In this method first abort the one of the processes in the deadlock state & allocated the resources to some other process in the dead lock state then check whether the dead lock breaked or not.

-> If not abort the another process from the deadlock state continue this process until we recover from dead lock.

This method also expensive but compare with first one it is better.

Resource Preemption:- There are 3 methods to eliminate dead locks using resource preemption.

1) **Selecting a victim:-** Select a victim resource from the dead lock state & preempt the one.

2) **Roll back:-** Roll back the processes & resources up to some [illegible] & restart from that state. This method requires keep more information about the state of processes.

[illegible] A resource [illegible]