UNIT-3
CHAPTER-8
MEMORY MANAGEMENT

TOPICS:

- **Swapping**
- **Contiguous Memory Allocation**
- **Paging**
- **Segmentation**
- **Segmentation with paging**

Memory Management

- ➢ Memory Management is the process of controlling and coordinating computer memory, assigning portions known as blocks to various running programs to optimize the overall performance of the system.

- ➢ It is the most important function of an operating system that manages primary memory. It helps processes to move back and forward between the main memory and execution disk. It helps OS to keep track of every memory location, irrespective of whether it is allocated to some process or it remains free.

Why Use Memory Management?

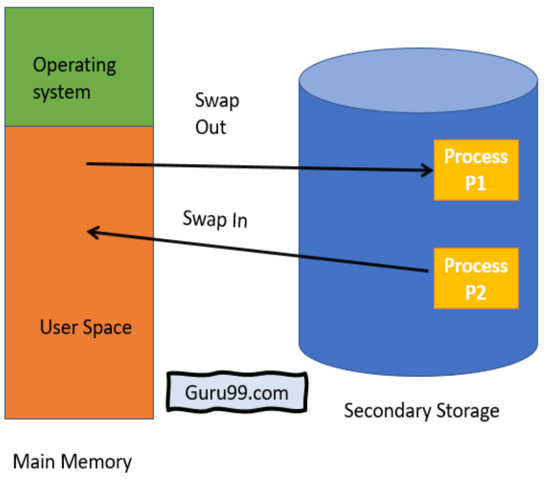
Here, are reasons for using memory management:

- ➢ It allows you to check how much memory needs to be allocated to processes that decide which processor should get memory at what time.
- ➢ Tracks whenever inventory gets freed or unallocated. According to it will update the status.
- ➢ It allocates the space to application routines.
- ➢ It also make sure that these applications do not interfere with each other.
- ➢ Helps protect different processes from each other
- ➢ It places the programs in memory so that memory is utilized to its full extent.

What is Swapping?

Swapping is a method in which the process should be swapped temporarily from the main memory to the backing store.
It will be later brought back into the memory for continue execution.

Backing store is a hard disk or some other secondary storage device that should be big enough inorder to accommodate copies of all memory images for all users.
It is also capable of offering direct access to these memory images.



Benefits of Swapping

Here, are major benefits/pros of swapping:

- ✓ It offers a higher degree of multiprogramming.
- ✓ Allows dynamic relocation. For example, if address binding at execution time is being used, then processes can be swap in different locations. Else in case of compile and load time bindings, processes should be moved to the same location.
- ✓ It helps to get better utilization of memory.
- ✓ Minimum wastage of CPU time on completion so it can easily be applied to a priority-based scheduling method to improve its performance.

Contiguous Memory Allocation

- In contiguous memory allocation, each process is contained in a single contiguous block of memory.
- Memory is divided into several fixed-size partitions. Each partition contains exactly one process.
- When a partition is free, a process is selected from the input queue and loaded into it.
- The free blocks of memory are known as holes.
- The set of holes is searched to determine which hole is best to allocate.

Memory Protection

Memory protection is a phenomenon by which we control memory access rights on a computer.

The main aim of it is to prevent a process from accessing memory that has not been allocated to it.

Hence prevents a bug within a process from affecting other processes, or the operating system itself, and instead results in a segmentation fault or storage violation exception being sent to the disturbing process, generally killing of process.

Memory Allocation in OS

Memory allocation is a process by which computer programs are assigned memory or space. It is of three types :

First Fit Allocation

The first hole that is big enough is allocated to the program.

Best Fit Allocation

The smallest hole that is big enough is allocated to the program.

Worst Fit Allocation

The largest hole that is big enough is allocated to the program.

Segmentation

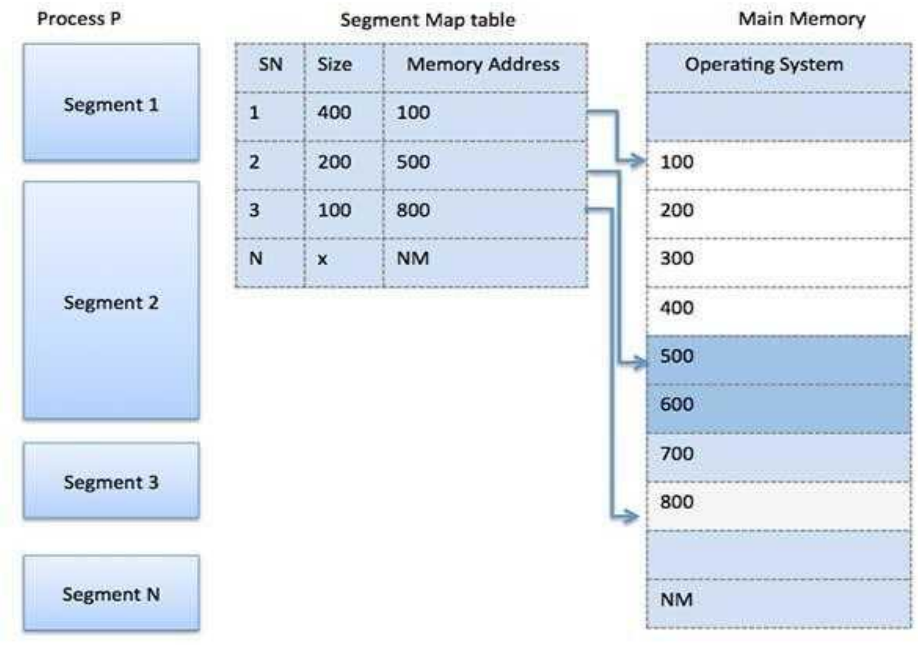
- Segmentation is a memory management technique in which each job is divided into several segments of different sizes, one for each module that contains pieces that perform related functions.
- Each segment is actually a different logical address space of the program.

- When a process is to be executed, its corresponding segmentation are loaded into non-contiguous memory though every segment is loaded into a contiguous block of available memory.

- Segmentation memory management works very similar to paging but here segments are of variable-length where as in paging pages are of fixed size.

- A program segment contains the program’s main function, utility functions, data structures, and so on.
- The operating system maintains a segment map table for every process and a list of free memory blocks along with segment numbers, their size and corresponding memory locations in main memory.
- For each segment, the table stores the starting address of the segment and the length of the segment.

A reference to a memory location includes a value that identifies a segment and an offset.



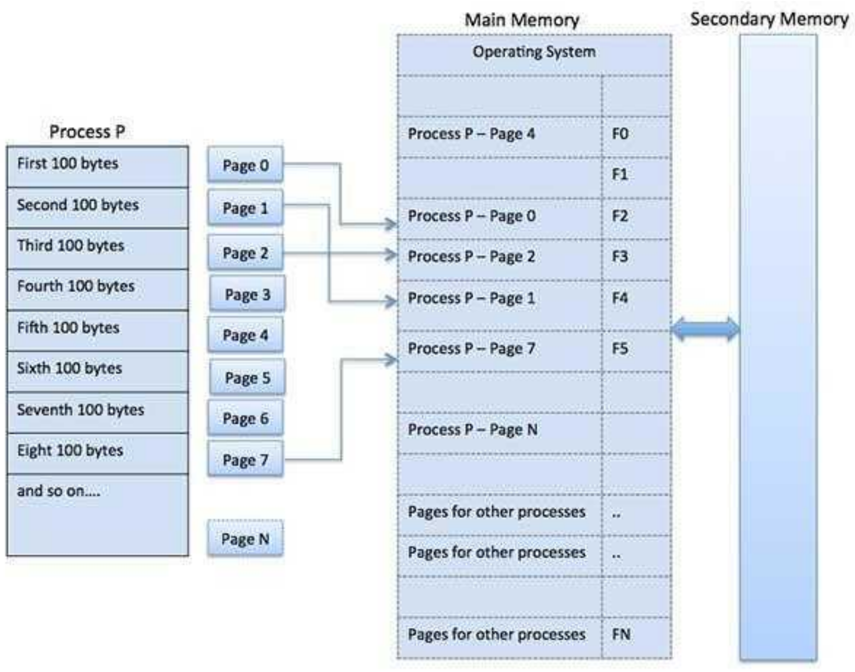
SN	Size	Memory Address
1	400	100
2	200	500
3	100	800
N	x	NM

Paging

- A computer can address more memory than the amount physically installed on the system.
- This extra memory is actually called virtual memory and it is a section of a hard that's set up to emulate the computer's RAM.
- Paging technique plays an important role in implementing virtual memory.

- Paging is a memory management technique in which process address space is broken into blocks of the same size called pages (size is power of 2, between 512 bytes and 8192 bytes).
- The size of the process is measured in the number of pages.

- Similarly, main memory is divided into small fixed-sized blocks of (physical) memory called frames and the size of a frame is kept the same as that of a page to have optimum utilization of the main memory and to avoid external fragmentation.



Address Translation

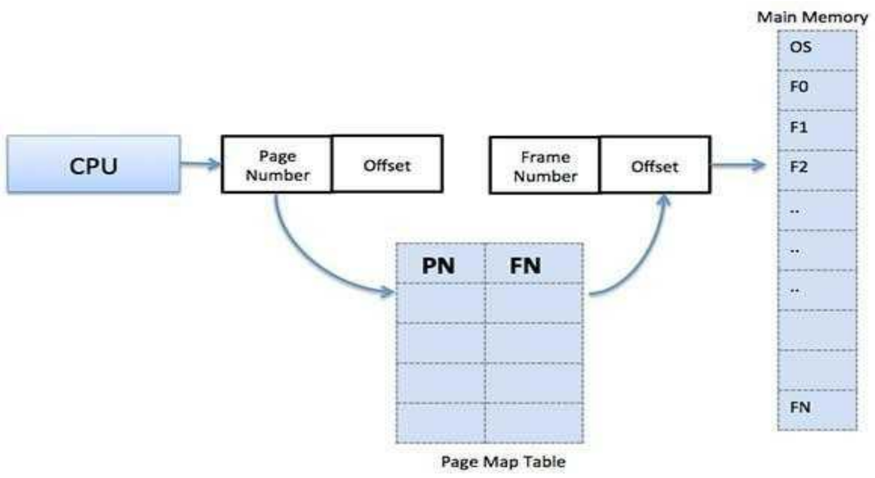
Page address is called logical address and represented by page number and the offset.

Logical Address = Page number + page offset

Frame address is called physical address and represented by a frame number and the offset.

Physical Address = Frame number + page offset

A data structure called page map table is used to keep track of the relation between a page of a process to a frame in physical memory.



- When the system allocates a frame to any page, it translates this logical address into a physical address and create entry into the page table to be used throughout execution of the program.

- When a process is to be executed, its corresponding pages are loaded into any available memory frames.
- Suppose you have a program of 8Kb but your memory can accommodate only 5Kb at a given point in time, then the paging concept will come into picture.
- When a computer runs out of RAM, the operating system (OS) will move idle or unwanted pages of memory to secondary memory to free up RAM for other processes and brings them back when needed by the program.

- This process continues during the whole execution of the program where the OS keeps removing idle pages from the main memory and write them onto the secondary memory and bring them back when required by the program.

Advantages and Disadvantages of Paging
Here is a list of advantages and disadvantages of paging –

- Paging reduces external fragmentation, but still suffer from internal fragmentation.
- Paging is simple to implement and assumed as an efficient memory management technique.
- Due to equal size of the pages and frames, swapping becomes very easy.
- Page table requires extra memory space, so may not be good for a system having small RAM.

Segmented Paging:___

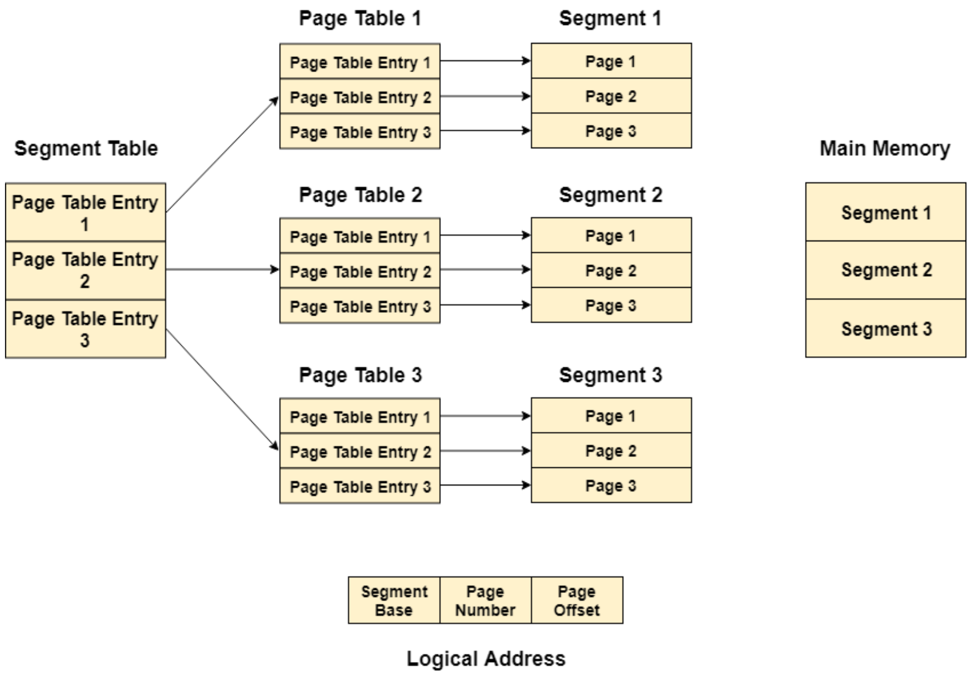
- Pure segmentation is not very popular and not being used in many of the operating systems.
- However, Segmentation can be combined with Paging to get the best features out of both the techniques.

- In Segmented Paging, the main memory is divided into variable size segments which are further divided into fixed size pages.

- Pages are smaller than segments.
- Each Segment has a page table which means every program has multiple page tables.
- The logical address is represented as Segment Number (base address), Page number and page offset.

Segment Number → It points to the appropriate Segment Number.
Page Number → It Points to the exact page within the segment
Page Offset → Used as an offset within the page frame

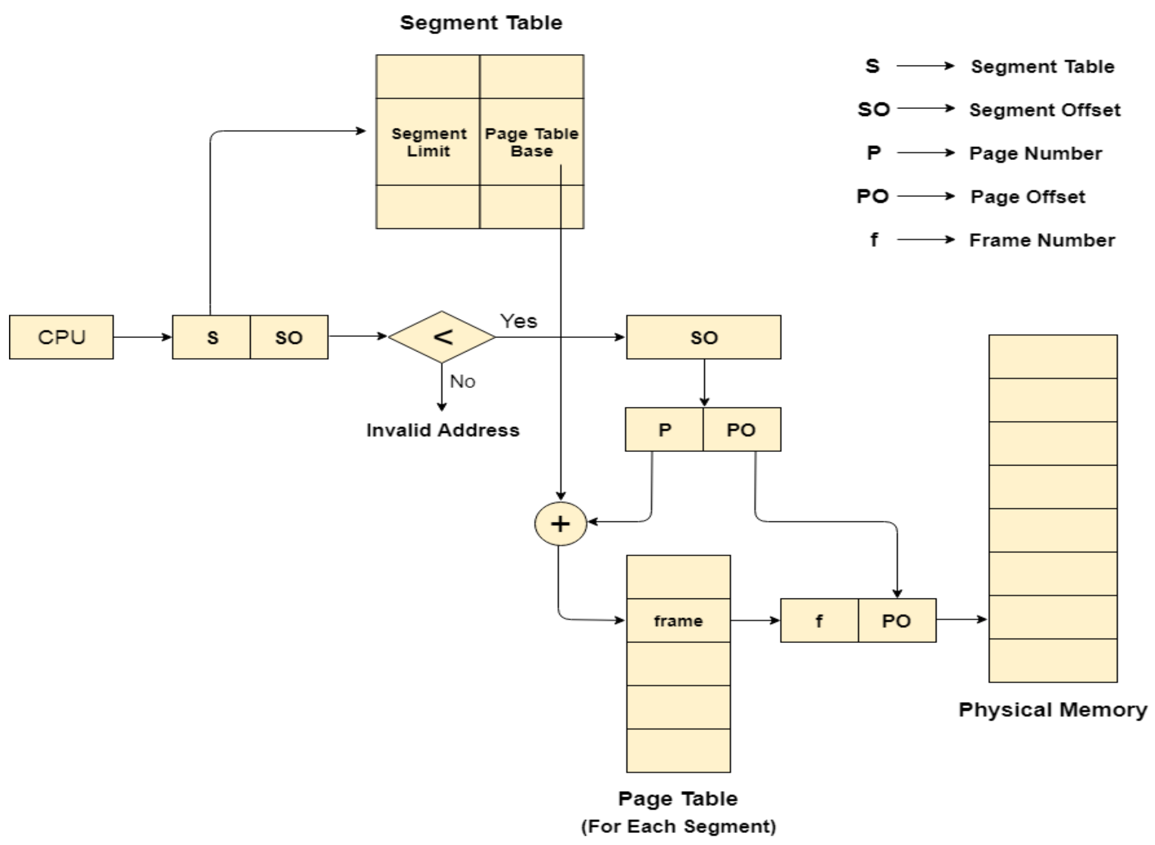
- Each Page table contains the various information about every page of the segment.
- The Segment Table contains the information about every segment.
- Each segment table entry points to a page table entry and every page table entry is mapped to one of the page within a segment.



Translation of logical address to physical address

- The CPU generates a logical address which is divided into two parts: Segment Number and Segment Offset.
- The Segment Offset must be less than the segment limit. Offset is further divided into Page number and Page Offset.
- To map the exact page number in the page table, the page number is added into the page table base.

- The actual frame number with the page offset is mapped to the main memory to get the desired word in the page of the certain segment of the process.



Advantages of Segmented Paging

- It reduces memory usage.
- Page table size is limited by the segment size.
- Segment table has only one entry corresponding to one actual segment.
- External Fragmentation is not there.
- It simplifies memory allocation.

Disadvantages of Segmented Paging

- Internal Fragmentation will be there.
- The complexity level will be much higher as compare to paging.
- Page Tables need to be contiguously stored in the memory.

Chapter -9
VIRTUAL MEMORY

TOPICS:

- **Demand Paging**
- **Process Creation**
- **Page Replacement**
- **Allocation of Frames**
- **Thrashing**

Virtual Memory:

- ➢ Virtual Memory is a storage mechanism which offers user an illusion of having a very big main memory.
- ➢ It is done by treating a part of secondary memory as the main memory. In Virtual memory, the user can store processes with a bigger size than the available main memory.

- ➢ Therefore, instead of loading one long process in the main memory, the OS loads the various parts of more than one process in the main memory. Virtual memory is mostly implemented with demand paging and demand segmentation.

Background

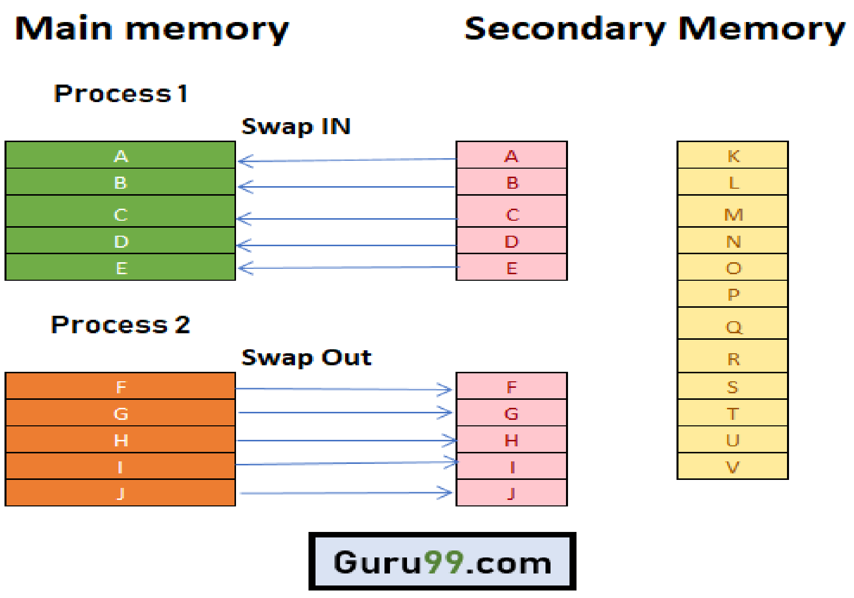
• Preceding sections talked about how to avoid memory fragmentation by breaking process memory requirements down into smaller bites (pages), and storing the pages non-contiguously in memory. However the entire process still had to be stored in memory somewhere.

• In practice, most real processes do not need all their pages, or at least not all at once, for several reasons:

1. Error handling code is not needed unless that specific error occurs, some of which are quite rare.
2. Arrays are often over-sized for worst-case scenarios, and only a small fraction of the arrays are actually used in practice.
3. Certain features of certain programs are rarely used, such as the routine to balance the federal budget. ☺

- The ability to load only the portions of processes that were actually needed (and only when they were needed) has several benefits:
  - ➢ Programs could be written for a much larger address space (virtual memory space) than physically exists on the computer.
  - ➢ Because each process is only using a fraction of their total address space, there is more memory left for other programs, improving CPU utilization and system throughput.
  - ➢ Less I/O is needed for swapping processes in and out of RAM, speeding things up.

What is Demand Paging?



- ➢ A demand paging mechanism is very much similar to a paging system with swapping where processes stored in the secondary memory and pages are loaded only on demand, not in advance.

- ➢ So, when a context switch occurs, the OS never copy any of the old program’s pages from the disk or any of the new program’s pages into the main memory.

Instead, it will start executing the new program after loading the first page and fetches the program’s pages, which are referenced.

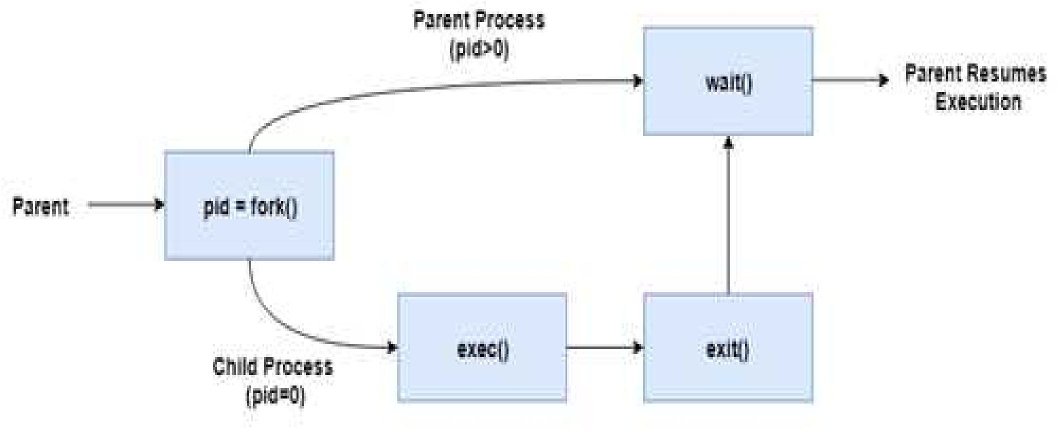
- During the program execution, if the program references a page that may not be available in the main memory because it was swapped, then the processor considers it as an invalid memory reference.
- That's because the page fault and transfers send control back from the program to the OS, which demands to store page back into the memory.

Process Creation

- A process may be created in the system for different operations. Some of the events that lead to process creation are as follows –
- User request for process creation
- System Initialization
- Batch job initialization
- Execution of a process creation system call by a running process

- A process may be created by another process using **fork().** The creating process is called the parent process and the created process is the child process.
- A child process can have only one parent but a parent process may have many children.
- Both the parent and child processes have the same memory image, open files and environment strings. However, they have distinct address spaces.

A diagram that demonstrates process creation using fork() is as follows –

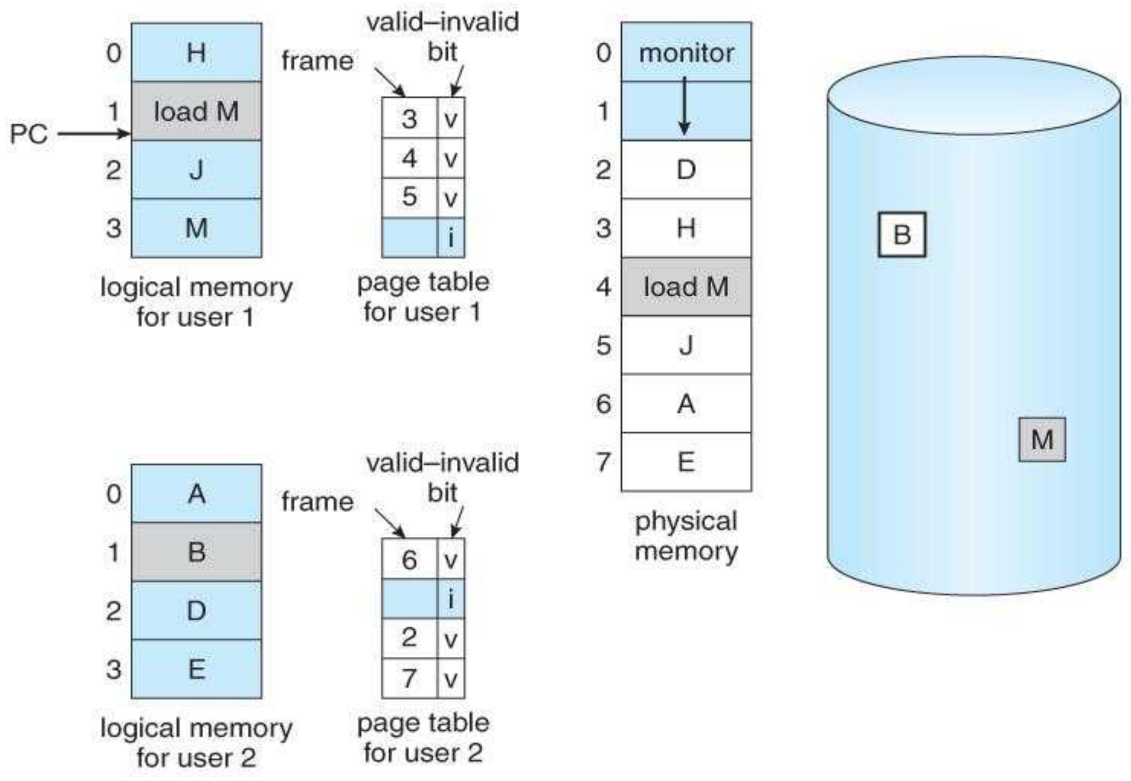


Process Creation Using fork()

Page Replacement

- In order to make the most use of virtual memory, we load several processes into memory at the same time. Since we only load the pages that are actually needed by each process at any given time, there is room to load many more processes than if we had to load in the entire process.
- However memory is also needed for other purposes (such as I/O buffering), and what happens if some process suddenly decides it needs more pages and there aren't any free frames available? There are several possible solutions to consider:

1. Adjust the memory used by I/O buffering, etc., to free up some frames for user processes. The decision of how to allocate memory for I/O versus user processes is a complex one, yielding different policies on different systems. (Some allocate a fixed amount for I/O, and others let the I/O system contend for memory along with everything else.)
2. Put the process requesting more pages into a wait queue until some free frames become available.
3. Swap some process out of memory completely, freeing up its page frames.
4. Find some page in memory that isn't being used right now, and swap that page only out to disk, freeing up a frame that can be allocated to the process requesting it. This is known as page replacement, and is the most common solution. There are many different algorithms for page replacement, which is the subject of the remainder of this section.



Basic Page Replacement

• The previously discussed page-fault processing assumed that there would be free frames available on the free-frame list. Now the page-fault handling must be modified to free up a frame if necessary, as follows:

1. Find the location of the desired page on the disk, either in swap space or in the file system.

2. Find a free frame:

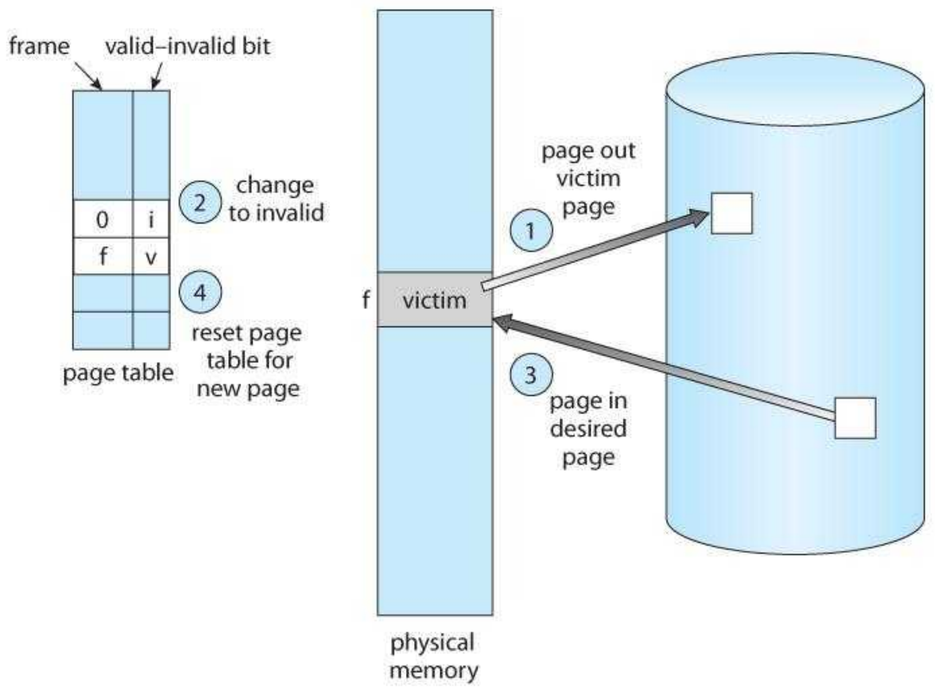
a. If there is a free frame, use it.

b. If there is no free frame, use a page-replacement algorithm to select an existing frame to be replaced, known as the victim frame.

c. Write the victim frame to disk. Change all related page tables to indicate that this page is no longer in memory.

3. Read in the desired page and store it in the frame. Adjust all related page and frame tables to indicate the change.

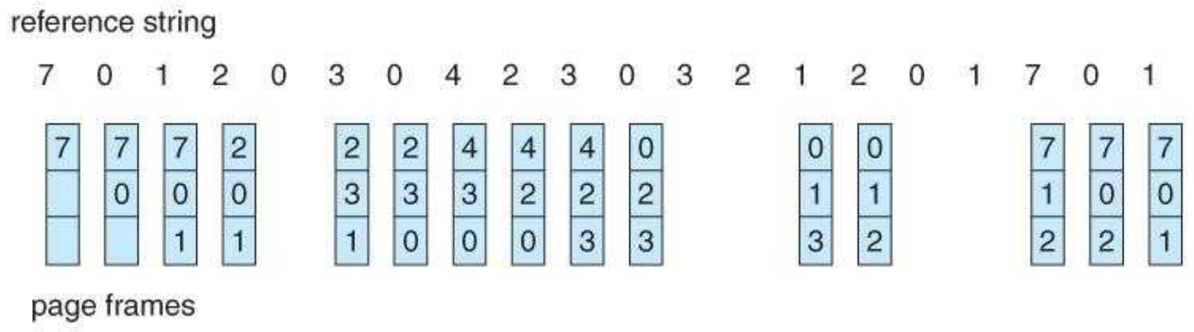
4. Restart the process that was waiting for this page.



- There are two major requirements to implement a successful demand paging system. We must develop a frame-allocation algorithm and a page-replacement algorithm.
- The former centers around how many frames are allocated to each process (and to other needs), and the latter deals with how to select a page for replacement when there are no free frames available.

FIFO Page Replacement

- A simple and obvious page replacement strategy is FIFO, i.e. first-in-first-out.
- As new pages are brought in, they are added to the tail of a queue, and the page at the head of the queue is the next victim.
- In the following example, 20 page requests result in 15 page faults:

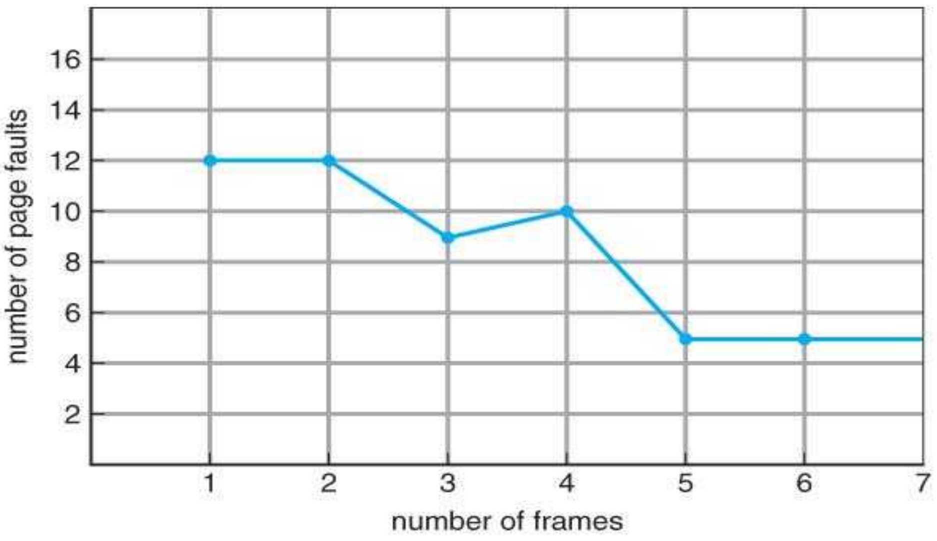


FIFO page-replacement algorithm.

- Although FIFO is simple and easy, it is not always optimal, or even efficient.

- An interesting effect that can occur with FIFO is Belady's anomaly, in which increasing the number of frames available can actually increase the number of page faults that occur! Consider, for example, the following chart based on the page sequence (1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5) and a varying number of available frames.

Obviously the maximum number of faults is 12 (every request generates a fault), and the minimum number is 5 (each page loaded only once), but in between there are some interesting results:



Features:

Whenever a new page loaded, the page recently comes in the memory is removed.

So, it is easy to decide which page requires to be removed as its identification number is always at the FIFO stack.

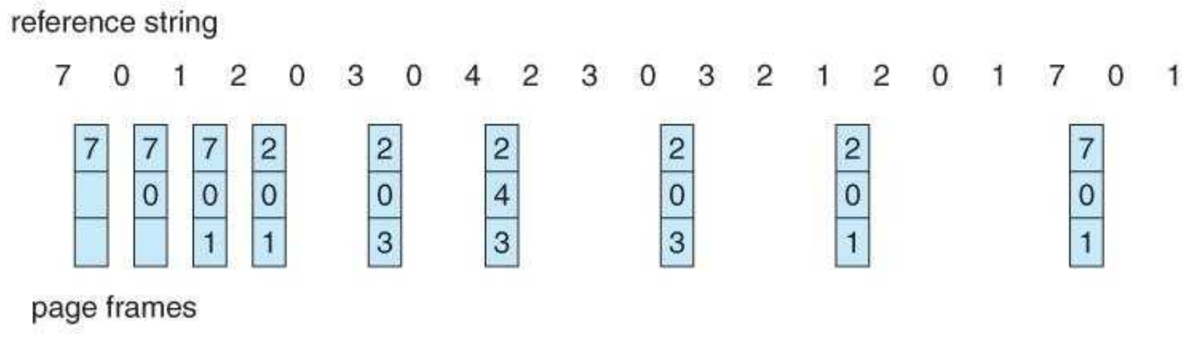
The oldest page in the main memory is one that should be selected for replacement first.

Optimal Page Replacement

- ✓ The discovery of Belady's anomaly lead to the search for an optimal page-replacement algorithm, which is simply that which yields the lowest of all possible page-faults, and which does not suffer from Belady's anomaly.
- ✓ Such an algorithm does exist, and is called OPT or MIN. This algorithm is simply "Replace the page that will not be used for the longest time in the future."
- ✓ For example, Figure shows that by applying OPT to the same reference string used for the FIFO example, the minimum number of possible page faults is 9.

Since 6 of the page-faults are unavoidable (the first reference to each new page), FIFO can be shown to require 3 times as many (extra) page faults as the optimal algorithm. (Note: The book claims that only the first three page faults are required by all algorithms, indicating that FIFO is only twice as bad as OPT.)

✓ Unfortunately OPT cannot be implemented in practice, because it requires foretelling the future, but it makes a nice benchmark for the comparison and evaluation of real proposed new algorithms.



Optimal page-replacement algorithm

Features:

- ➢ Optimal algorithm results in the fewest number of page faults. This algorithm is difficult to implement.
- ➢ An optimal page-replacement algorithm method has the lowest page-fault rate of all algorithms.
- ➢ This algorithm exists and which should be called MIN or OPT.

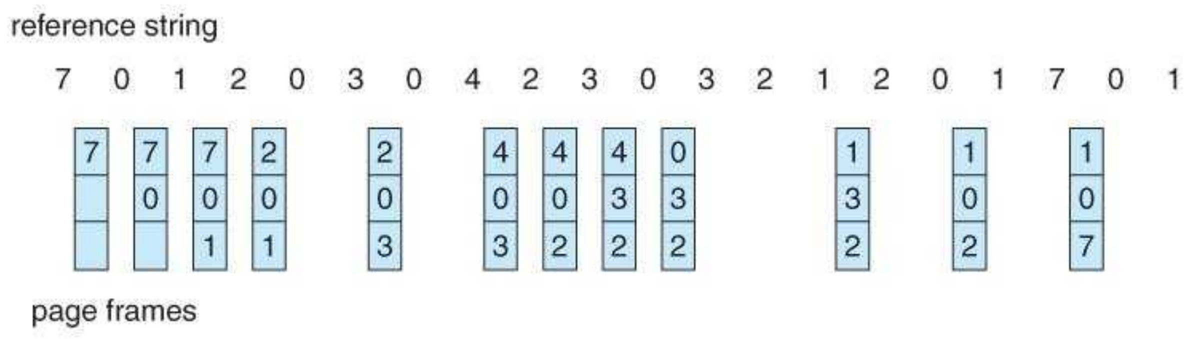
Replace the page which unlike to use for a longer period of time. It only uses the time when a page needs to be used.

LRU Page Replacement

• The prediction behind LRU, the Least Recently Used, algorithm is that the page that has not been used in the longest time is the one that will not be used again in the near future. (Note the distinction between FIFO and LRU: The former looks at the oldest load time, and the latter looks at the oldest use time.)

• Some view LRU as analogous to OPT, except looking backwards in time instead of forwards. (OPT has the interesting property that for any reference string S and its reverse R, OPT will generate the same number of page faults for S and for R. It turns out that LRU has this same property.)

• Figure 9.15 illustrates LRU for our sample string, yielding 12 page faults, (as compared to 15 for FIFO and 9 for OPT.)



LRU is considered a good replacement policy, and is often used. The problem is how exactly to implement it. There are two simple approaches commonly used:

1.Counters. Every memory access increments a counter, and the current value of this counter is stored in the page table entry for that page. Then finding the LRU page involves simple searching the table for the page with the smallest counter value. Note that overflowing of the counter must be considered.

2.**Stack.** Another approach is to use a stack, and whenever a page is accessed, pull that page from the middle of the stack and place it on the top. The LRU page will always be at the bottom of the stack. Because this requires removing objects from the middle of the stack, a doubly linked list is the recommended data structure.

Features:

- The LRU replacement method has the highest count.
- This counter is also called aging registers, which specify their age and how much their associated pages should also be referenced.
- The page which hasn't been used for the longest time in the main memory is the one that should be selected for replacement.
- It also keeps a list and replaces pages by looking back into time.

Allocation of frames in Operating System

An important aspect of operating systems, virtual memory is implemented using demand paging. Demand paging necessitates the development of a page-replacement algorithm and a **frame allocation algorithm**. Frame allocation algorithms are used if you have multiple processes; it helps decide how many frames to allocate to each process.
There are various constraints to the strategies for the allocation of frames:

- You cannot allocate more than the total number of available frames.
- At least a minimum number of frames should be allocated to each process. This constraint is supported by two reasons. The first reason is,

as less number of frames are allocated, there is an increase in the page fault ratio, decreasing the performance of the execution of the process. Secondly, there should be enough frames to hold all the different pages that any single instruction can reference.

Frame allocation algorithms –

The two algorithms commonly used to allocate frames to a process are:

Equal allocation: In a system with x frames and y processes, each process gets equal number of frames, i.e. x/y. For instance, if the system has 48 frames and 9 processes, each process will get 5 frames. The three frames which are not allocated to any process can be used as a free-frame buffer pool.

Disadvantage: In systems with processes of varying sizes, it does not make much sense to give each process equal frames. Allocation of a large number of frames to a small process will eventually lead to the wastage of a large number of allocated unused frames.

Proportional allocation: Frames are allocated to each process according to the process size.
For a process p_i of size s_i, the number of allocated frames is $\mathbf{a_i = (s_i/S)*m}$, where S is the sum of the sizes of all the processes and m is the number of frames in the system. For instance, in a system with 62 frames, if there is a process of 10KB and another process of 127KB, then the first process will be allocated (10/137)*62 = 4 frames and the other process will get (127/137)*62 = 57 frames.

Advantage: All the processes share the available frames according to their needs, rather than equally.

Thrashing

If a process cannot maintain its minimum required number of frames, then it must be swapped out, freeing up frames for other processes.

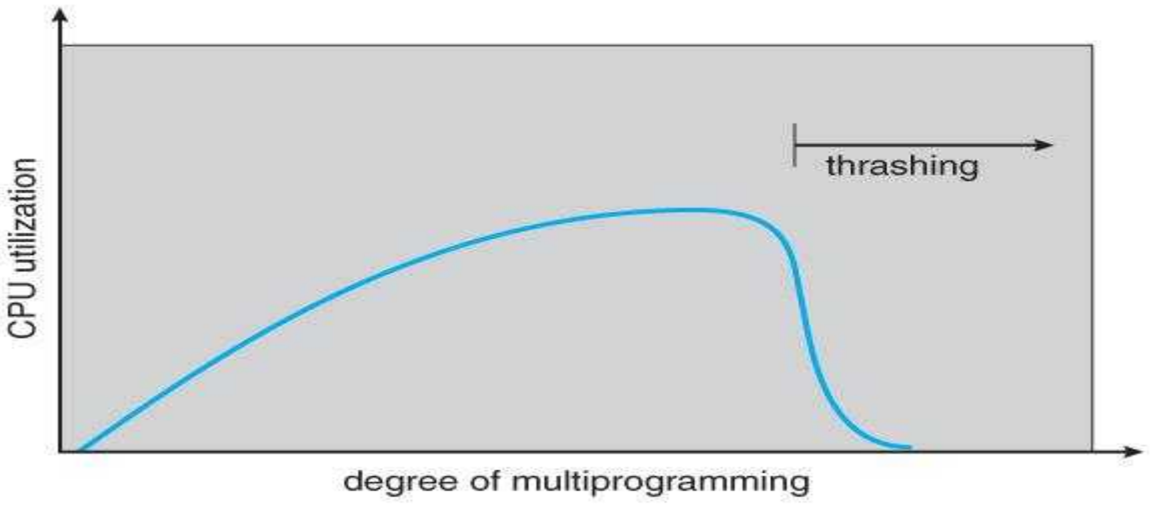
This is an intermediate level of CPU scheduling.

But what about a process that can keep its minimum, but cannot keep all of the frames that it is currently using on a regular basis? In this case it is forced to page out pages that it will need again in the very near future, leading to large numbers of page faults.

A process that is spending more time paging than executing is said to be thrashing.

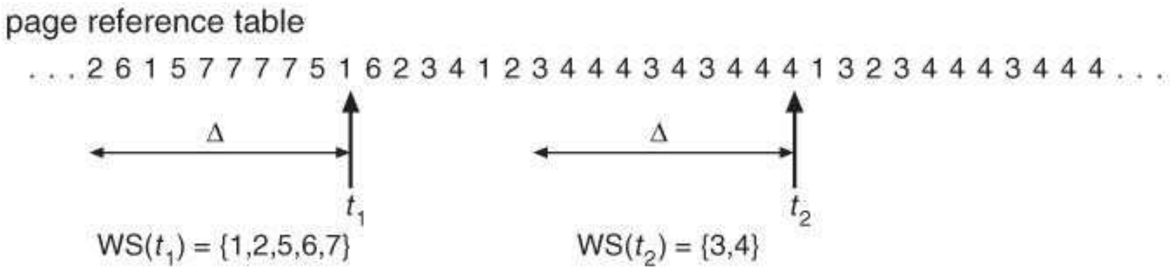
Cause of Thrashing

- Early process scheduling schemes would control the level of multiprogramming allowed based on CPU utilization, adding in more processes when CPU utilization was low.

- The problem is that when memory filled up and processes started spending lots of time waiting for their pages to page in, then CPU utilization would lower, causing the schedule to add in even more processes and exacerbating the problem! Eventually the system would essentially grind to a halt.
- Local page replacement policies can prevent one thrashing process from taking pages away from other processes, but it still tends to clog up the I/O queue, thereby slowing down any other process that needs to do even a little bit of paging (or any other I/O for that matter.)
- 

Working-Set Model

- The ***working set model*** is based on the concept of locality, and defines a ***working set window***, of length ***delta.*** Whatever pages are included in the most recent delta page references are said to be in the processes working set window, and comprise its current working set.



Chapter -1O
FILE SYSTEM INTERFACE

TOPICS:

- **File Concepts**
- **Access Methods**
- **Directory Structure**
- **File System Mounting**
- **File Sharing**

File Concept

Computers can store information on various storage media, such as magnetic disks, magnetic tapes, and optical disks. So that the computer system will be convenient to use, the operating system provides a uniform logical view of information storage. The operating system abstracts from the physical properties of its ...

File Attributes

Name – only information kept in human-readable form

● **Identifie**r – unique tag (number) identifies file within file system

● **Type** – needed for systems that support different types

● **Location** – pointer to file location on device

● **Size** – current file size

● **Protection** – controls who can do reading, writing, executing

● **Time, date, and user identification** – data for protection, security,
and usage monitoring

● Information about files are kept in the directory structure, which is
maintained on the disk..

File Operations

File is an abstract data type

● Create

● Write – at write pointer location

● Read – at read pointer location

● Reposition within file - seek

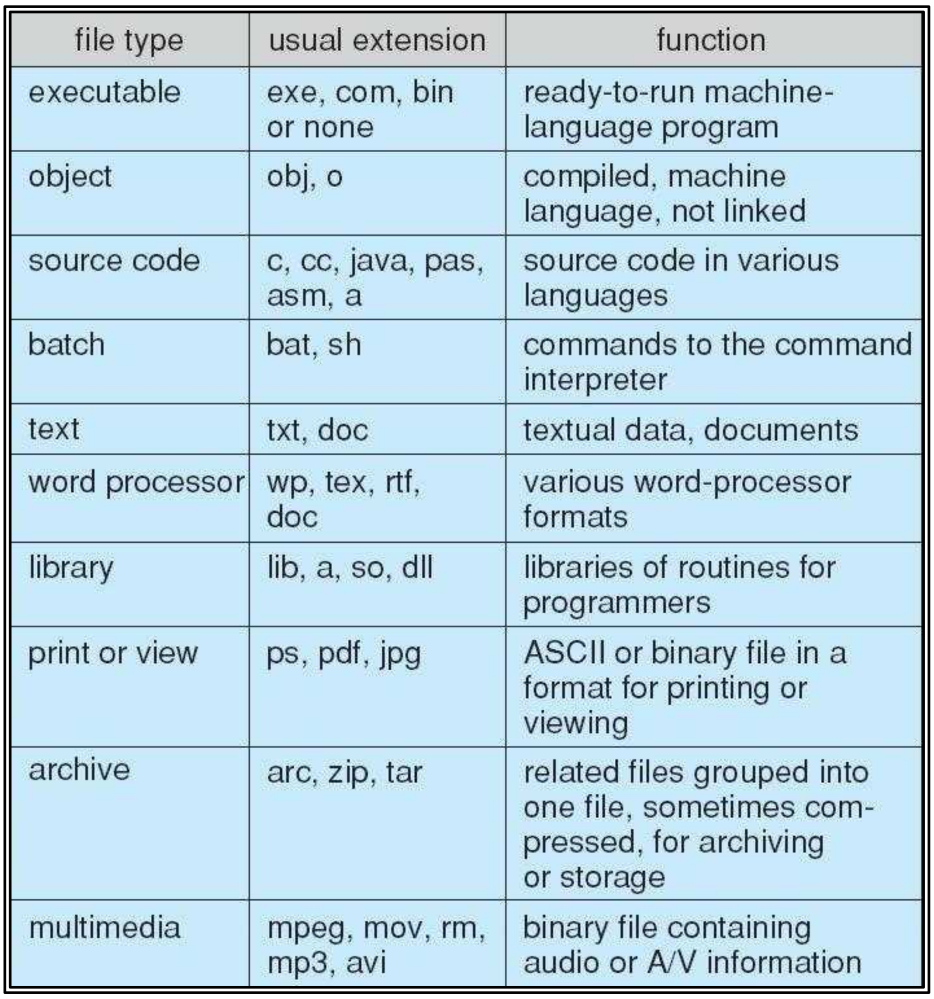
● Delete

● Truncate

● Open(Fi) – search the directory structure on disk for entry Fiand move the content of entry to memory

● Close (Fi) – move the content of entry Fiin memory to directory structure on disk…

File Types – Name, Extension

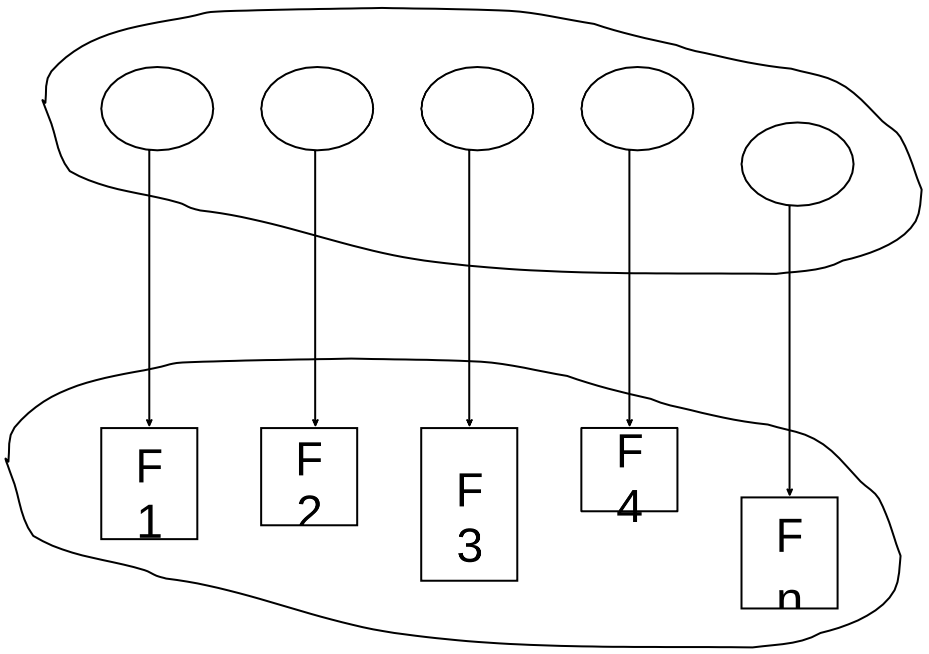


file type	usual extension	function
executable	exe, com, bin or none	ready-to-run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, pas, asm, a	source code in various languages
batch	bat, sh	commands to the command interpreter
text	txt, doc	textual data, documents
word processor	wp, tex, rtf, doc	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	ps, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	arc, zip, tar	related files grouped into one file, sometimes com-pressed, for archiving or storage
multimedia	mpeg, mov, rm, mp3, avi	binary file containing audio or A/V information

Directory Structure

● A collection of nodes containing information about all files

Directory



Files

Operations Performed on Directory

- Search for afile
- Create afile
- Delete afile
- List adirectory
- Rename afile

Directory Organization

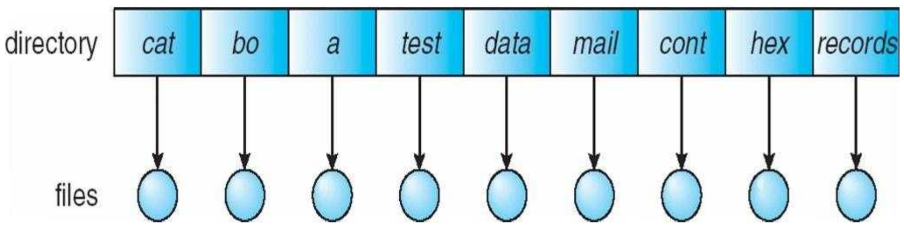
The directory is organized logically to obtain

- Efficiency – locating a file quickly
- Naming – convenient to users
- Two users can have same name for different files
- The same file can have several different names
- Grouping – logical grouping of files by properties, (e.g., all Java programs, all games, …)

Single-Level Directory

A single directory for allusers

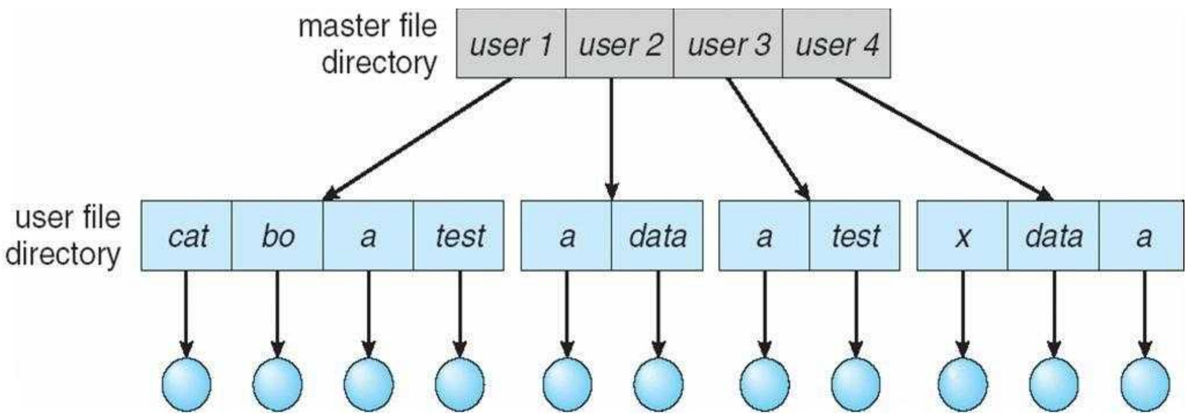
- Namingproblem



- Groupingproblem

Two-Level Directory

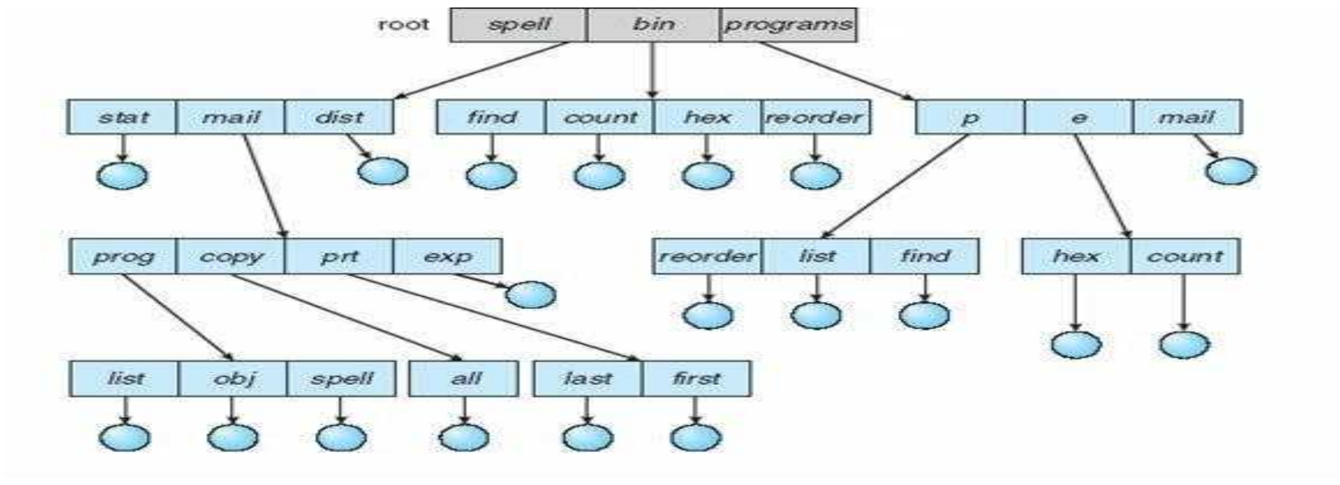
Separate directory for each user



- Pathname
- Can have the same file name for differentuser
- Efficientsearching
- No groupingcapability

Tree-Structured Directories

- Efficientsearching



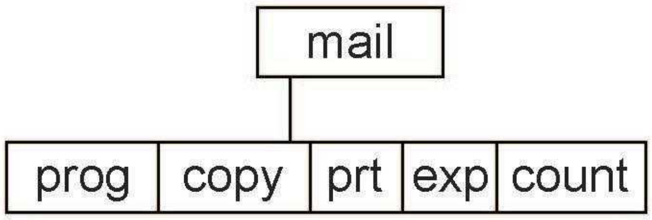
- GroupingCapability
- Current directory (workingdirectory)
- cd/spell/mail/prog
    - **typelist**
    - Absolute or relative path name
    - Creating a new file is done in current directory
    - Delete a file

      rm <file-name>

Creating a new subdirectory is done in current directory

mkdir <dir-name>

Example: if in current directory /mail mkdir count.

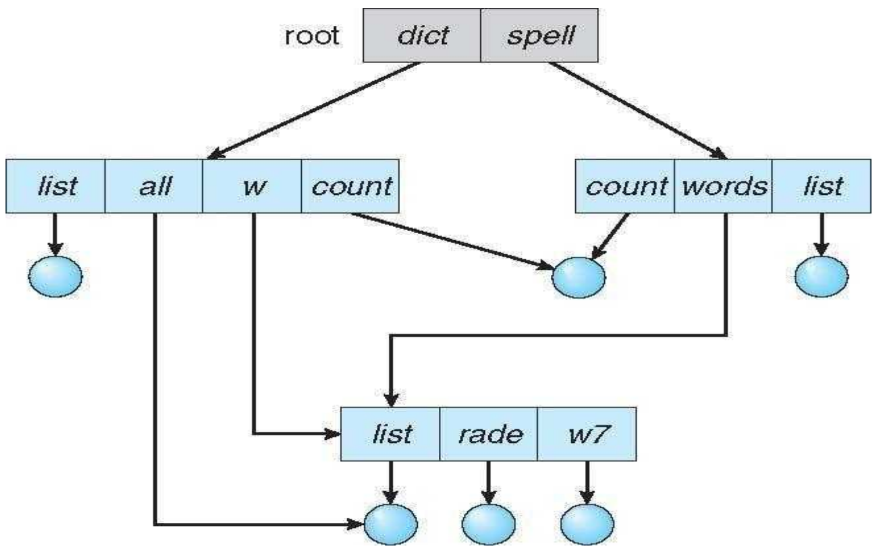


Deleting "mail" ⇒deleting the entire subtree rooted by "mail"

Acyclic-Graph Directories

- Have shared subdirectories andfiles

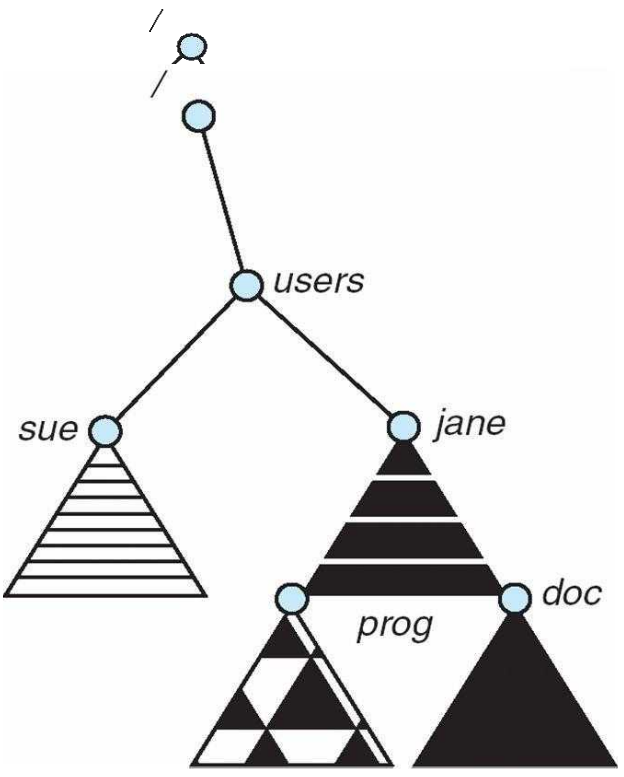
File System Mounting

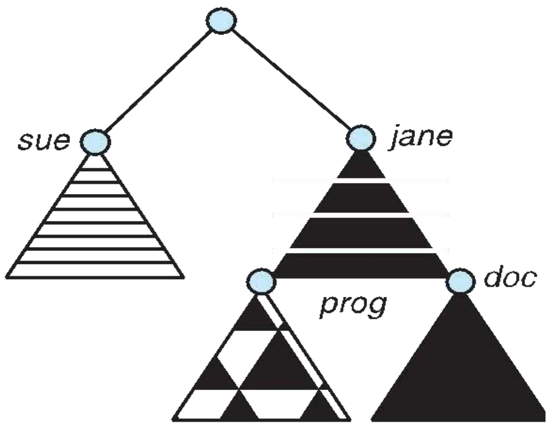


File System Mounting

- Afilesystemmustbe**mounted**beforeitcanbeaccessed
- Aunmountedfilesystem(i.e.,Fig.11-11(b))ismountedata**mount point**

Mount Point





(b)

File Sharing

- Sharing of files on multi-user systems is desirable
- Sharing may be done through a protection scheme
- On distributed systems, files may be shared across a network
- Network File System (NFS) is a common distributed file-sharing method
- If multi-user system
- **User IDs** identify users, allowing permissions and protections to be per-user

Group IDs allow users to be in groups, permitting group access rights

- Owner of a file / directory
- Group of a file / directory

File Sharing – Remote File Systems

Uses networking to allow file system access between systems

- Manually via programs like FTP
- Automatically, seamlessly using distributed file systems
- Semi automatically via the world wide web

Client-server model allows clients to mount remote file systems from servers

- Server can serve multiple clients
- Client and user-on-client identification is insecure or complicated
- NFS is standard UNIX client-server file sharing protocol
- Standard operating system file calls are translated into remote calls..

Protection

- File owner/creator should be able to control:
- what can be done
- by whom
- Types of access
- **Read**
- **Write**
- **Execute**
- **Append**
- **Delete**
- **List**

Chapter -11
FILE SYSTEM IMPLEMENTATION

TOPICS:

- **File System Structure**
- **File System Implementation**
- **Directory Implementation**
- **Allocation Methods**
- **Free space Management**
- **Efficiency and Performance**
- **Recovery**

File-System Structure

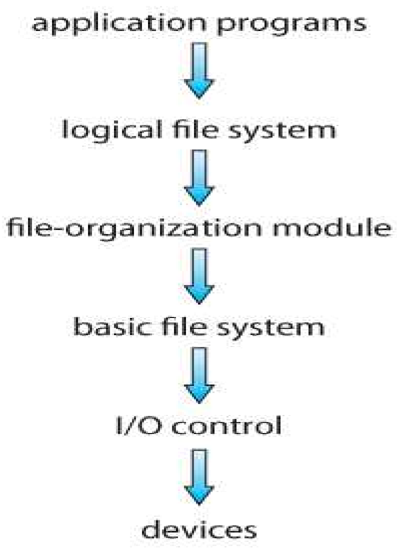
- Hard disks have two important properties that make them suitable for secondary storage of files in file systems:

(1) Blocks of data can be rewritten in place, and

(2) they are direct access, allowing any block of data to be accessed with only (relatively) minor movements of the disk heads and rotational latency.

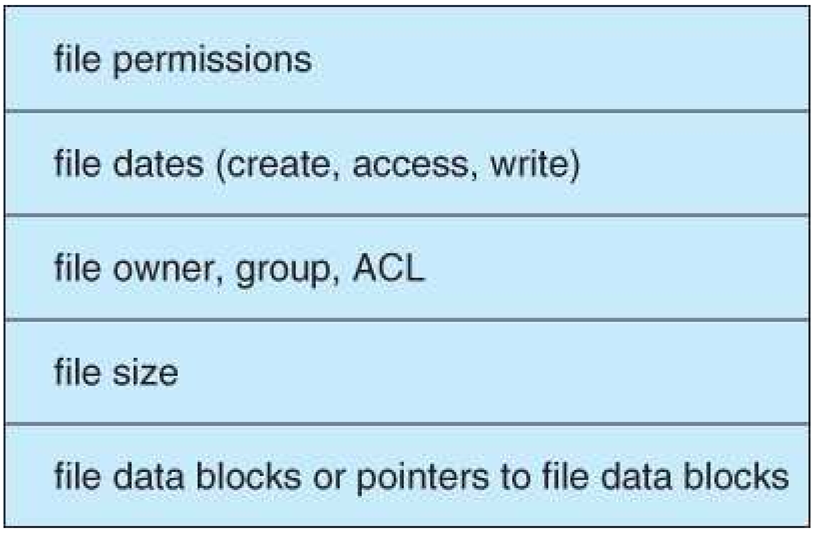
- Disks are usually accessed in physical blocks, rather than a byte at a time. Block sizes may range from 512 bytes to 4K or larger.
- File systems organize storage on disk drives, and can be viewed as a layered design:
- At the lowest layer are the physical devices, consisting of the magnetic media, motors & controls, and the electronics connected to them and controlling them. Modern disk put more and more of the electronic controls directly on the disk drive itself, leaving relatively little work for the disk controller card to perform.
- **I/O Control** consists of **device drivers**, special software programs (often written in assembly) which communicate with the devices by reading and writing special codes directly to and from memory addresses corresponding to the controller card's registers.
- Each controller card (device) on a system has a different set of addresses (registers, a.k.a. ports) that it listens to, and a unique set of command codes and results codes that it understands.

- **The basic file system** level works directly with the device drivers in terms of retrieving and storing raw blocks of data, without any consideration for what is in each block.
- Depending on the system, blocks may be referred to with a single block number, (e.g. block # 234234), or with head-sector-cylinder combinations.
- The file organization module knows about files and their logical blocks, and how they map to physical blocks on the disk.
- In addition to translating from logical to physical blocks, the file organization module also maintains the list of free blocks, and allocates free blocks to files as needed.
- The logical file system deals with all of the meta data associated with a file (UID, GID, mode, dates, etc), i.e. everything about the file except the data itself.
- This level manages the directory structure and the mapping of file names to file control blocks, FCBs, which contain all of the meta data as well as block number information for finding the data on the disk.
- The layered approach to file systems means that much of the code can be used uniformly for a wide variety of different file systems, and only certain layers need to be filesystem specific.
- Common file systems in use include the UNIX file system, UFS, the Berkeley Fast File System, FFS, Windows systems FAT, FAT32, NTFS, CD-ROM systems ISO 9660, and for Linux the extended file systems ext2 and ext3 (among 40 others supported.)



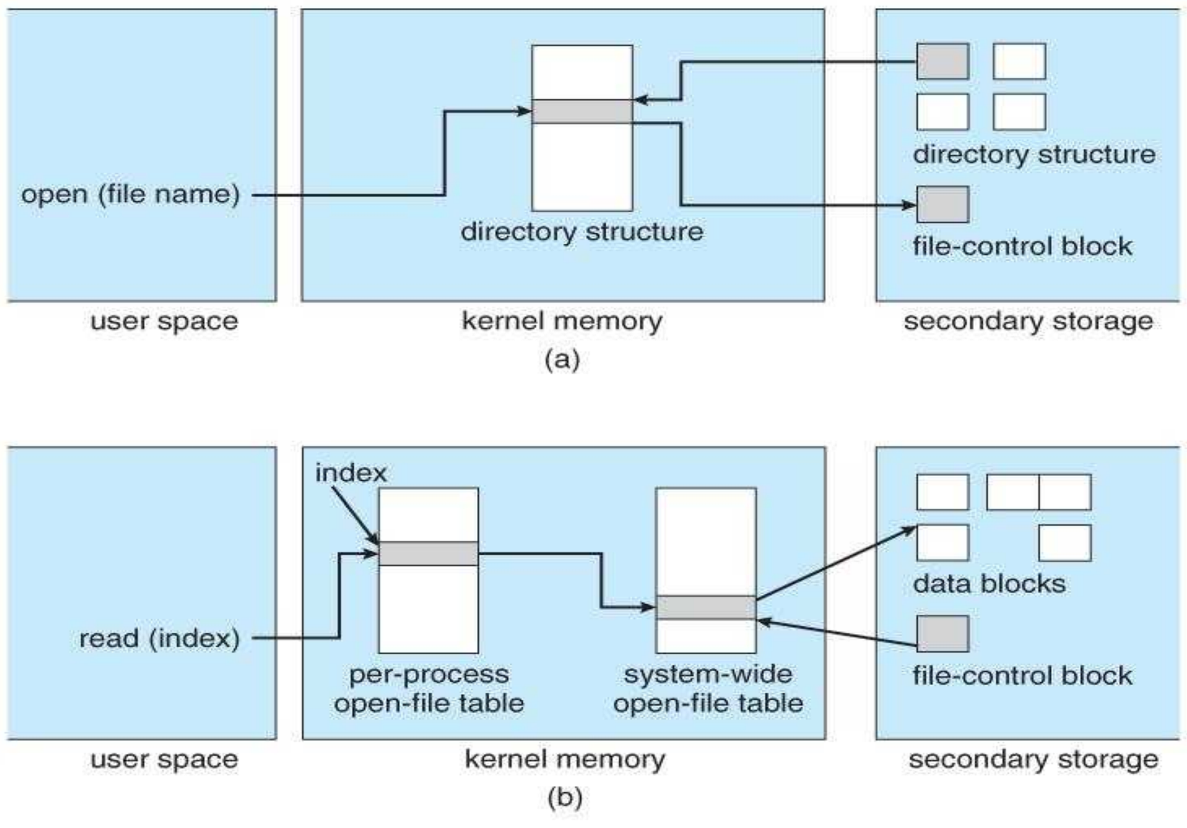
File-System Implementation

- File systems store several important data structures on the disk:
    - A ***boot-control block***, (per volume) a.k.a. the ***boot block*** in UNIX or the ***partition boot sector*** in Windows contains information about how to boot the system off of this disk. This will generally be the first sector of the volume if there is a bootable system loaded on that volume, or the block will be left vacant otherwise.
    - A ***volume control block,*** (per volume) a.k.a. the ***master file table*** in UNIX or the ***superblock*** in Windows, which contains information such as the partition table, number of blocks on each filesystem, and pointers to free blocks and free FCB blocks.
    - A directory structure (per file system), containing file names and pointers to corresponding FCBs. UNIX uses inode numbers, and NTFS uses a ***master file table.***
    - The ***File Control Block, FCB,*** (per file) containing details about ownership, size, permissions, dates, etc. UNIX stores this information in inodes, and NTFS in the master file table as a relational database structure.


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

- There are also several key data structures stored in memory:
    - An in-memory mount table.
    - An in-memory directory cache of recently accessed directory information.
    - ***A system-wide open file table***, containing a copy of the FCB for every currently open file in the system, as well as some other related information.
    - ***A per-process open file table,*** containing a pointer to the system open file table as well as some other information. (For example the current file position pointer may be either here or in the system file table, depending on the implementation and whether the file is being shared or not.)
- Figure 12.3 illustrates some of the interactions of file system components when files are created and/or used:

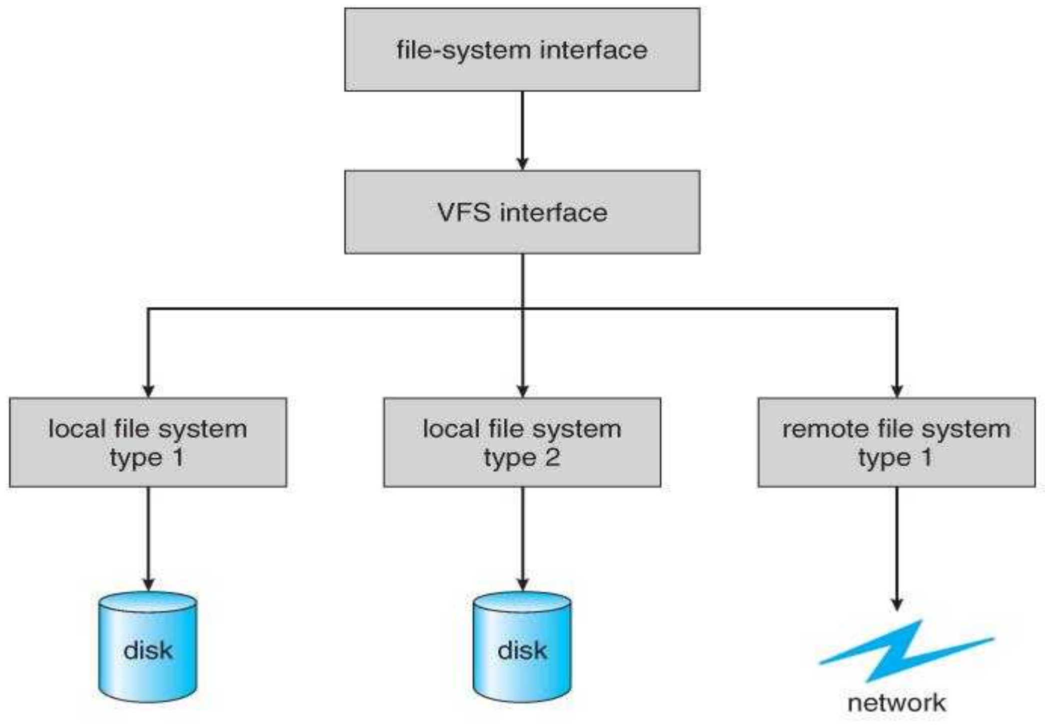
- When a new file is created, a new FCB is allocated and filled out with important information regarding the new file. The appropriate directory is modified with the new file name and FCB information.
- When a file is accessed during a program, the open( ) system call reads in the FCB information from disk, and stores it in the system-wide open file table. An entry is added to the per-process open file table referencing the system-wide table, and an index into the per-process table is returned by the open( ) system call. UNIX refers to this index as a ***file descriptor***, and Windows refers to it as a ***file handle***.
- If another process already has a file open when a new request comes in for the same file, and it is sharable, then a counter in the system-wide table is incremented and the per-process table is adjusted to point to the existing entry in the system-wide table.
- When a file is closed, the per-process table entry is freed, and the counter in the system-wide table is decremented. If that counter reaches zero, then the system wide table is also freed. Any data currently stored in memory cache for this file is written out to disk if necessary.



Virtual File Systems

- ***Virtual File Systems, VFS***, provide a common interface to multiple different filesystem types. In addition, it provides for a unique identifier (vnode) for files across the entire space, including across all filesystems of different types. (UNIX inodes are unique only across a single filesystem, and certainly do not carry across networked file systems.)

- The VFS in Linux is based upon four key object types:
    - The ***inode*** object, representing an individual file
    - The ***file*** object, representing an open file.
    - The ***superblock*** object, representing a filesystem.
    - The ***dentry*** object, representing a directory entry.
- Linux VFS provides a set of common functionalities for each filesystem, using function pointers accessed through a table. The same functionality is accessed through the same table position for all filesystem types, though the actual functions pointed to by the pointers may be filesystem-specific. See /usr/include/linux/fs.h for full details. Common operations provided include open( ), read( ), write( ), and mmap( ).



Directory Implementation

- Directories need to be fast to search, insert, and delete, with a minimum of wasted disk space.

Linear List

- A linear list is the simplest and easiest directory structure to set up, but it does have some drawbacks.

- Finding a file (or verifying one does not already exist upon creation) requires a linear search.
- Deletions can be done by moving all entries, flagging an entry as deleted, or by moving the last entry into the newly vacant position.
- Sorting the list makes searches faster, at the expense of more complex insertions and deletions.
- A linked list makes insertions and deletions into a sorted list easier, with overhead for the links.
- More complex data structures, such as B-trees, could also be considered.

Hash Table

- A hash table can also be used to speed up searches.
- Hash tables are generally implemented ***in addition to*** a linear or other structure

Allocation Methods

- There are three major methods of storing files on disks: contiguous, linked, and indexed.

Contiguous Allocation

- Contiguous Allocation requires that all blocks of a file be kept together contiguously.

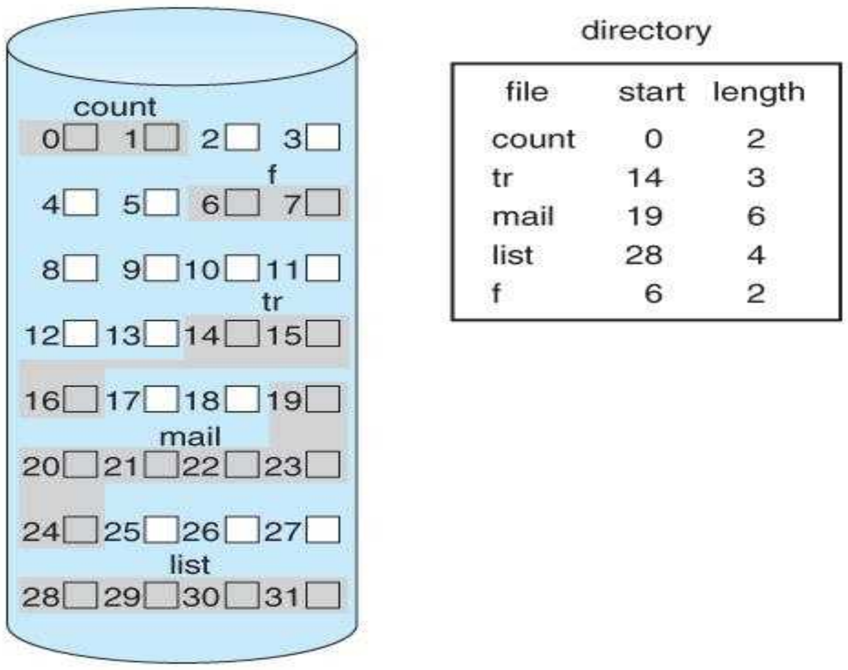
- Performance is very fast, because reading successive blocks of the same file generally requires no movement of the disk heads, or at most one small step to the next adjacent cylinder.

- Storage allocation involves the same issues discussed earlier for the allocation of contiguous blocks of memory (first fit, best fit, fragmentation problems, etc.)

- The distinction is that the high time penalty required for moving the disk heads from spot to spot may now justify the benefits of keeping files contiguously when possible.

- (Even file systems that do not by default store files contiguously can benefit from certain utilities that compact the disk and make all files contiguous in the process.)

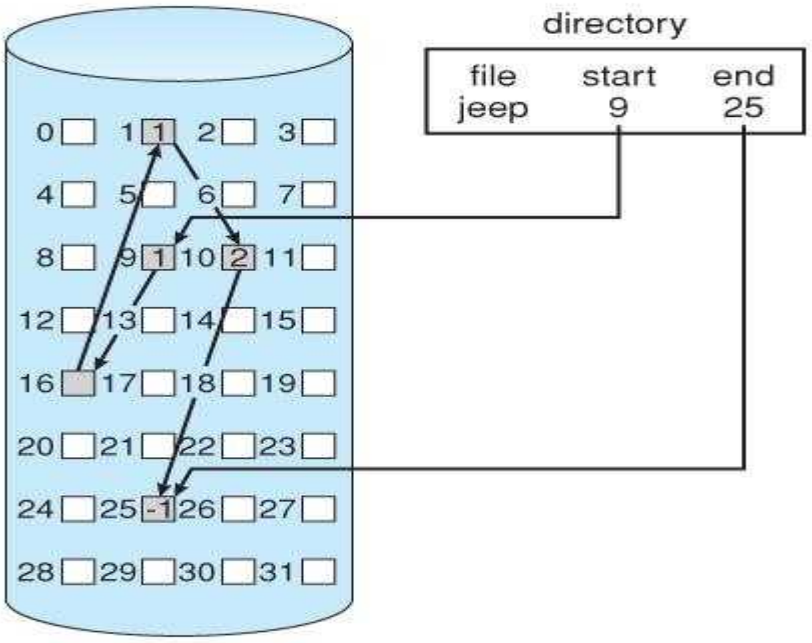
- Problems can arise when files grow, or if the exact size of a file is unknown at creation time:
    - Over-estimation of the file's final size increases external fragmentation and wastes disk space.
    - Under-estimation may require that a file be moved or a process aborted if the file grows beyond its originally allocated space.
    - If a file grows slowly over a long time period and the total final space must be allocated initially, then a lot of space becomes unusable before the file fills the space.
    - A variation is to allocate file space in large contiguous chunks, called extents. When a file outgrows its original extent, then an additional one is allocated. (For example an extent may be the size of a complete track or even cylinder, aligned on an appropriate track or cylinder boundary.) The high-performance files system Veritas uses extents to optimize performance.



file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

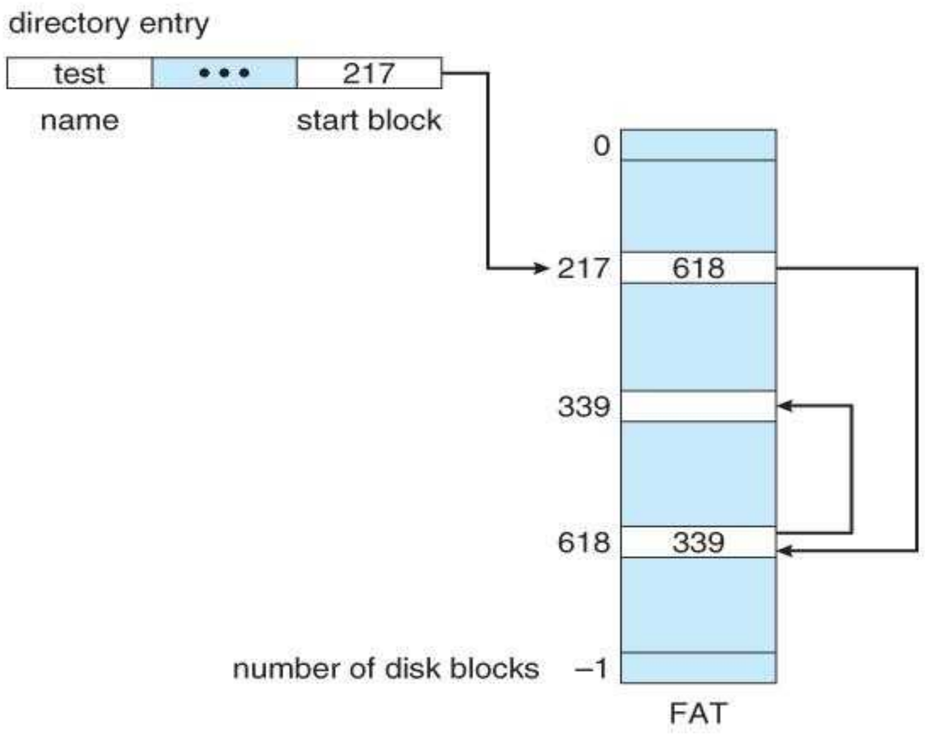
Linked Allocation

- Disk files can be stored as linked lists, with the expense of the storage space consumed by each link. (E.g. a block may be 508 bytes instead of 512.)
- Linked allocation involves no external fragmentation, does not require pre-known file sizes, and allows files to grow dynamically at any time.
- Unfortunately linked allocation is only efficient for sequential access files, as random access requires starting at the beginning of the list for each new location access.
- Allocating clusters of blocks reduces the space wasted by pointers, at the cost of internal fragmentation.
- Another big problem with linked allocation is reliability if a pointer is lost or damaged. Doubly linked lists provide some protection, at the cost of additional overhead and wasted space.



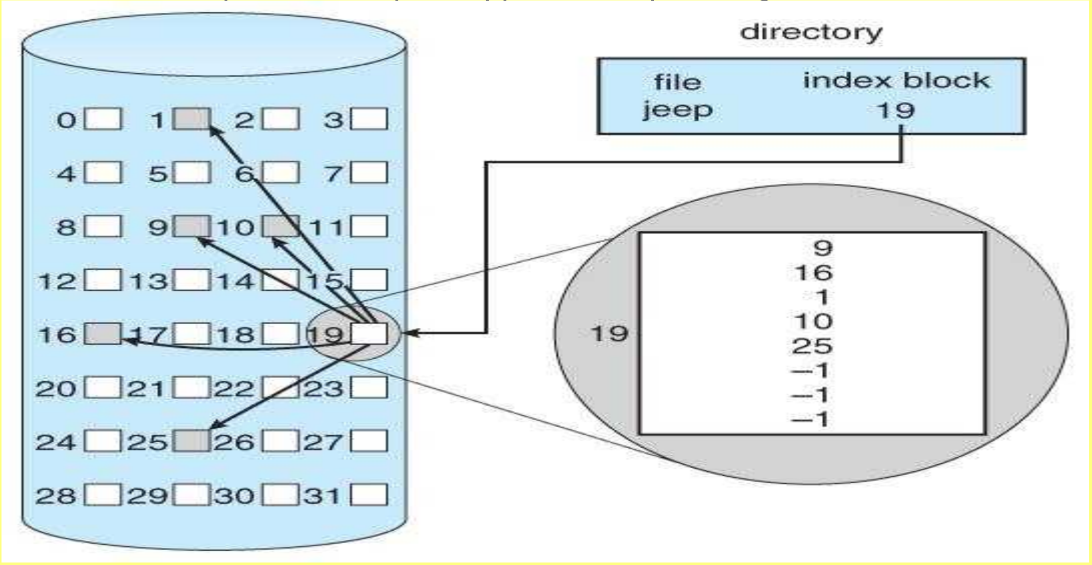
Linked allocation of disk space.

The File Allocation Table, FAT, used by DOS is a variation of linked allocation, where all the links are stored in a separate table at the beginning of the disk. The benefit of this approach is that the FAT table can be cached in memory, greatly improving random access speeds.



Indexed Allocation

- ***Indexed Allocation*** combines all of the indexes for accessing each file into a common block (for that file), as opposed to spreading them all over the disk

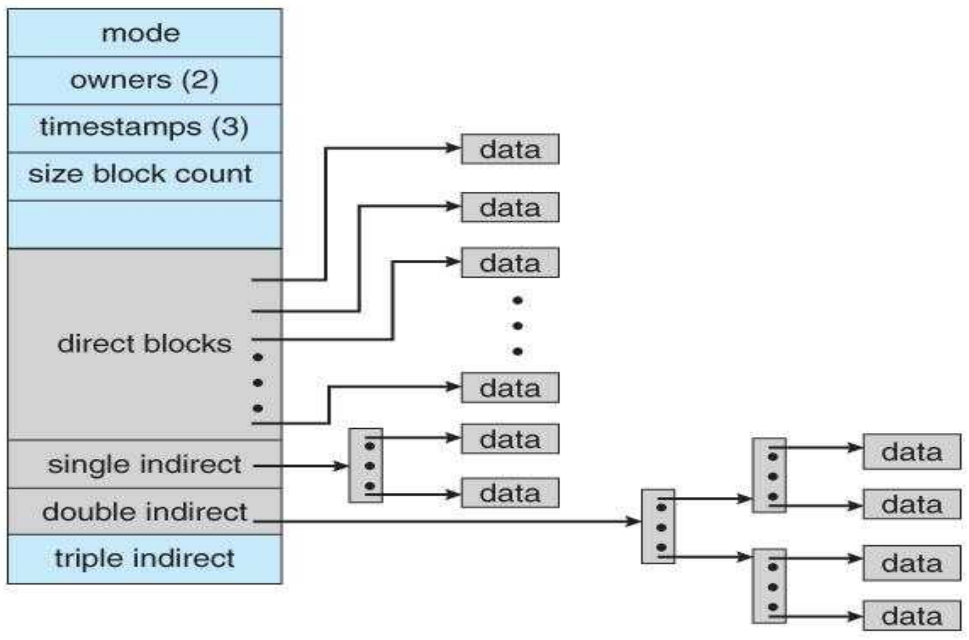


or storing them in a FAT table.

Indexed allocation of disk space.

- Some disk space is wasted (relative to linked lists or FAT tables) because an entire index block must be allocated for each file, regardless of how many data blocks the file contains. This leads to questions of how big the index block should be, and how it should be implemented. There are several approaches:
  - **Linked Scheme -** An index block is one disk block, which can be read and written in a single disk operation. The first index block contains some header information, the first N block addresses, and if necessary a pointer to additional linked index blocks.
  - **Multi-Level Index -** The first index block contains a set of pointers to secondary index blocks, which in turn contain pointers to the actual data blocks.
  - **Combined Scheme -** This is the scheme used in UNIX inodes, in which the first 12 or so data block pointers are stored directly in the inode, and then singly, doubly, and triply indirect pointers provide access to more data blocks as needed. (See below.) The advantage of this scheme is that for small files

(which many are), the data blocks are readily accessible (up to 48K with 4K block sizes); files up to about 4144K (using 4K blocks) are accessible with only a single indirect block (which can be cached), and huge files are still accessible using a relatively small number of disk accesses (larger in theory than can be addressed by a 32-bit address, which is why some systems have moved to 64-bit file pointers.)



Performance

- The optimal allocation method is different for sequential access files than for random access files, and is also different for small files than for large fi
- Some systems support more than one allocation method, which may require specifying how the file is to be used (sequential or random access) at the time it is allocated. Such systems also provide conversion utilities.
- Some systems have been known to use contiguous access for small files, and automatically switch to an indexed scheme when file sizes surpass a certain threshold.
- And of course some systems adjust their allocation schemes (e.g. block sizes) to best match the characteristics of the hardware for optimum performance.

Free-Space Management

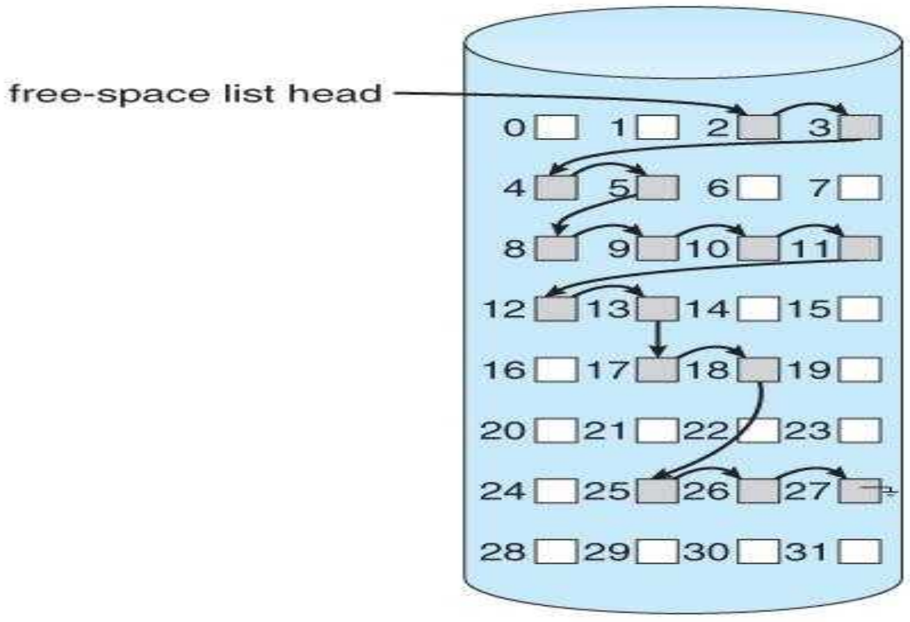
- Another important aspect of disk management is keeping track of and allocating free space.

Bit Vector

- One simple approach is to use a bit vector, in which each bit represents a disk block, set to 1 if free or 0 if allocated.
- Fast algorithms exist for quickly finding contiguous blocks of a given size
- The down side is that a 40GB disk requires over 5MB just to store the bitmap. (For example.)

Linked List

- A linked list can also be used to keep track of all free blocks.
- Traversing the list and/or finding a contiguous block of a given size are not easy, but fortunately are not frequently needed operations. Generally the system just adds and removes single blocks from the beginning of the list.
- The FAT table keeps track of the free list as just one more linked list on the table.



G**rouping**

- A variation on linked list free lists is to use links of blocks of indices of free blocks. If a block holds up to N addresses, then the first block in the linked-list contains up to N-1 addresses of free blocks and a pointer to the next block of free addresses.

Counting

- When there are multiple contiguous blocks of free space then the system can keep track of the starting address of the group and the number of contiguous free blocks.
- As long as the average length of a contiguous group of free blocks is greater than two this offers a savings in space needed for the free list. (Similar to

compression techniques used for graphics images when a group of pixels all the same color is encountered.)

Space Maps

- Sun's ZFS file system was designed for HUGE numbers and sizes of files, directories, and even file systems.
- The resulting data structures could be VERY inefficient if not implemented carefully. For example, freeing up a 1 GB file on a 1 TB file system could involve updating thousands of blocks of free list bit maps if the file was spread across the disk.
- ZFS uses a combination of techniques, starting with dividing the disk up into (hundreds of) metaslabs of a manageable size, each having their own space map.
- Free blocks are managed using the counting technique, but rather than write the information to a table, it is recorded in a log-structured transaction record. Adjacent free blocks are also coalesced into a larger single free block.
- An in-memory space map is constructed using a balanced tree data structure, constructed from the log data.
- The combination of the in-memory tree and the on-disk log provide for very fast and efficient management of these very large files and free blocks.

Efficiency and Performance

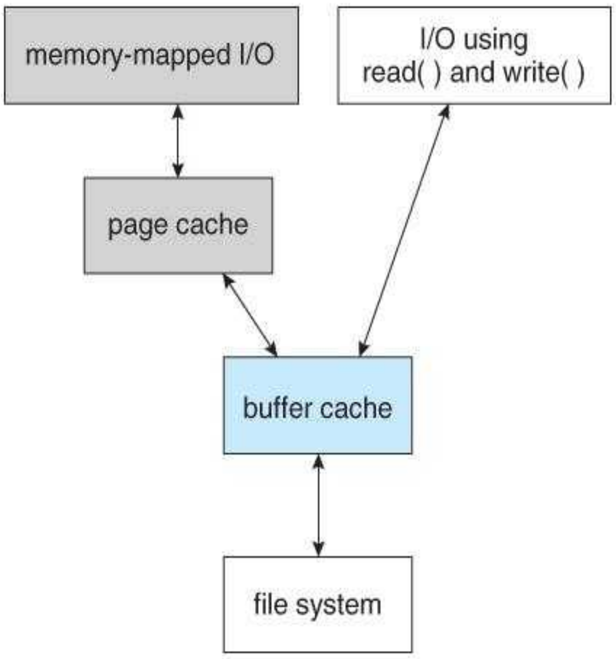
Efficiency

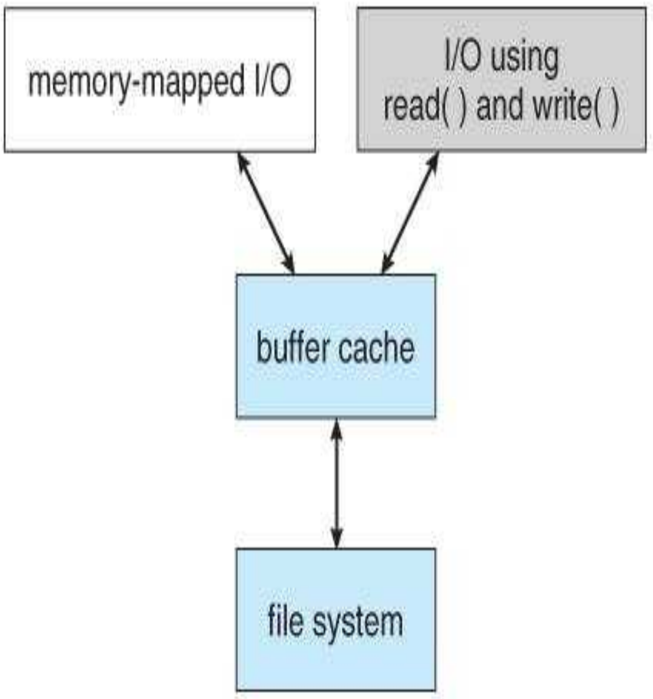
- UNIX pre-allocates inodes, which occupies space even before any files are created
- UNIX also distributes inodes across the disk, and tries to store data files near their inode, to reduce the distance of disk seeks between the inodes and the data
- Some systems use variable size clusters depending on the file size.
- The more data that is stored in a directory (e.g. last access time), the more often the directory blocks have to be re-written.
- As technology advances, addressing schemes have had to grow as well.
- Sun's ZFS file system uses 128-bit pointers, which should theoretically never need to be expanded. (The mass required to store 2^128 bytes with atomic storage would be at least 272 trillion kilograms!)
- Kernel table sizes used to be fixed, and could only be changed by rebuilding the kernels. Modern tables are dynamically allocated, but that requires more complicated algorithms for accessing them.

Performance

- Disk controllers generally include on-board caching. When a seek is requested, the heads are moved into place, and then an entire track is read, starting from whatever sector is currently under the heads (reducing latency.)
- The requested sector is returned and the unrequested portion of the track is cached in the disk's electronics.
- Some OS cache disk blocks they expect to need again in a buffer cache.
- A page cache connected to the virtual memory system is actually more efficient as memory addresses do not need to be converted to disk block addresses and back again.
- Some systems (Solaris, Linux, Windows 2000, NT, XP) use page caching for both process pages and file data in a unified virtual memory.
- Figures show the advantages of the unified buffer cache found in some versions of UNIX and Linux - Data does not need to be stored twice, and problems of inconsistent buffer information are avoided.

\





- Page replacement strategies can be complicated with a unified cache, as one needs to decide whether to replace process or file pages, and how many pages to guarantee to each category of pages.
- Solaris, for example, has gone through many variations, resulting in priority paging giving process pages priority over file I/O pages, and setting limits so that neither can knock the other completely out of memory.
- Another issue affecting performance is the question of whether to implement synchronous writes or asynchronous writes.
- Synchronous writes occur in the order in which the disk subsystem receives them, without caching;
- Asynchronous writes are cached, allowing the disk subsystem to schedule writes in a more efficient order. Metadata writes are often done synchronously.
- Some systems support flags to the open call requiring that writes be synchronous, for example for the benefit of database systems that require their writes be performed in a required order.
- The type of file access can also have an impact on optimal page replacement policies.
- For example, LRU is not necessarily a good policy for sequential access files.
- For these types of files progression normally goes in a forward direction only, and the most recently used page will not be needed again until after the file has been rewound and re-read from the beginning, (if it is ever needed at all.) On the other hand, we can expect to need the next page in the file fairly soon.
- For this reason sequential access files often take advantage of two special policies:
    - **Free-behind** frees up a page as soon as the next page in the file is requested, with the assumption that we are now done with the old page and won't need it again for a long time.
    - **Read-ahead** reads the requested page and several subsequent pages at the same time, with the assumption that those pages will be needed in the near future. This is similar to the track caching that is already performed by the disk controller, except it saves the future latency of

transferring data from the disk controller memory into motherboard main memory.

- The caching system and asynchronous writes speed up disk writes considerably, because the disk subsystem can schedule physical writes to the disk to minimize head movement and disk seek times.
- Reads, on the other hand, must be done more synchronously in spite of the caching system, with the result that disk writes can counter-intuitively be much faster on average than disk reads.

Recovery

Consistency Checking

- ➢ The storing of certain data structures (e.g. directories and inodes) in memory and the caching of disk operations can speed up performance, but what happens in the result of a system crash? All volatile memory structures are lost, and the information stored on the hard drive may be left in an inconsistent state.
- ➢ A Consistency Checker (fsck in UNIX, chkdsk or scandisk in Windows) is often run at boot time or mount time, particularly if a file system was not closed down properly. Some of the problems that these tools look for include:
    - Disk blocks allocated to files and also listed on the free list.
    - Disk blocks neither allocated to files nor on the free list.
    - Disk blocks allocated to more than one file.
    - The number of disk blocks allocated to a file inconsistent with the file's stated size.
    - Properly allocated files / inodes which do not appear in any directory entry.
    - Link counts for an inode not matching the number of references to that inode in the directory structure.
    - Two or more identical file names in the same directory.
    - Illegally linked directories, e.g. cyclical relationships where those are not allowed, or files/directories that are not accessible from the root of the directory tree.
    - Consistency checkers will often collect questionable disk blocks into new files with names such as chk00001.dat. These files may contain

valuable information that would otherwise be lost, but in most cases they can be safely deleted, (returning those disk blocks to the free list.)

- UNIX caches directory information for reads, but any changes that affect space allocation or metadata changes are written synchronously, before any of the corresponding data blocks are written to.

Log-Structured File Systems

Log-based transaction-oriented (a.k.a. journaling) filesystems borrow techniques developed for databases, guaranteeing that any given transaction either completes successfully or can be rolled back to a safe state before the transaction commenced:

- All metadata changes are written sequentially to a log.
- A set of changes for performing a specific task (e.g. moving a file) is a transaction.
- As changes are written to the log they are said to be committed, allowing the system to return to its work.
- In the meantime, the changes from the log are carried out on the actual filesystem, and a pointer keeps track of which changes in the log have been completed and which have not yet been completed.
- When all changes corresponding to a particular transaction have been completed, that transaction can be safely removed from the log.
- At any given time, the log will contain information pertaining to uncompleted transactions only, e.g. actions that were committed but for which the entire transaction has not yet been completed.
- From the log, the remaining transactions can be completed,

or if the transaction was aborted, then the partially completed changes can be undone.

NFS

- NFS is an abbreviation of the **Network File System**. It is a protocol of a distributed file system. This protocol was developed by the **Sun Microsystems** in the year of 1984.

- It is an architecture of the client/server, which contains a client program, server program, and a protocol that helps for communication between the client and server.

- It is that protocol which allows the users to access the data and files remotely over the network. Any user can easily implement the NFS protocol because it is an open standard.

- Any user can manipulate files as same as if they were on like other protocols. This protocol is also built on the ONC RPC system.

- This protocol is mainly implemented on those computing environments where the centralized management of resources and data is critical.
- It uses the Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) for accessing and delivering the data and files.

- Network File System is a protocol that works on all the networks of IP-based. It is implemented in that client/server application in which the server of NFS manages the authorization, authentication, and clients.
- This protocol is used with Apple Mac OS, Unix, and Unix-like operating systems such as Solaris, Linux, FreeBSD, AIX.

The NFS Protocol

Implemented as a set of remote procedure calls (RPCs):

- Searching for a file in a directory
- REading a set of directory entries
- Manipulating links and directories
- Accessing file attributes
- Reading and writing files