COURSE NAME: OPERATING SYSTEMS

COURSE CODE: BCS303

SEMESTER: 3

MODULE: 4 & 5

NUMBER OF HOURS: 10

CONTENTS:

WEB RESOURCES:

https://www.geeksforgeeks.org/operating-systems/
https://www.tutorialspoint.com/operating_system/index.htm

MODULE 4

VIRTUAL MEMORYMANAGEMENT

- Virtual memory is a technique that allows for the execution of partially loaded process.
- Advantages:
  - A program will not be limited by the amount of physical memory that is available user can able to write in to large virtual space.
  - Since each program takes less amount of physical memory, more than one program could be run at the same time which can increase the throughput and CPU utilization.
  - Less i/o operation is needed to swap or load user program in to memory. So each user program could run faster.

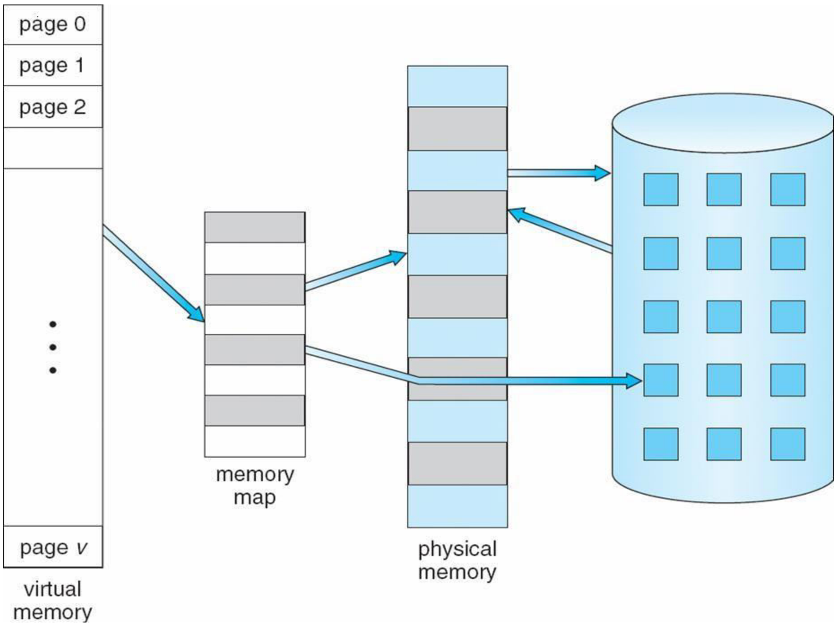


Fig: Virtual memory that is larger than physical memory.

- Virtual memory is the separation of users logical memory from physical memory. This separation allows an extremely large virtual memory to be provided when these is less physical memory.
- Separating logical memory from physical memory also allows files and memory to be shared by several different processes through page sharing.

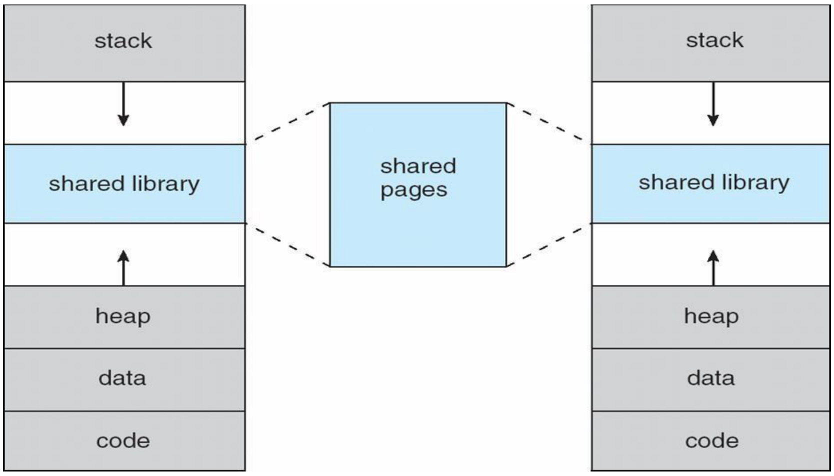


Fig: Shared Library using Virtual Memory

- Virtual memory is implemented using Demand Paging.
- Virtual address space: Every process has a virtual address space i.e used as the stack or heap grows in size.

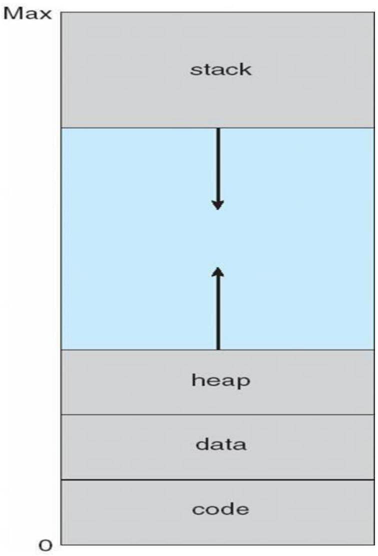


Fig: Virtual address space

DEMAND PAGING

- A demand paging is similar to paging system with swapping when we want to execute a process we swap the process the in to memory otherwise it will not be loaded in to memory.
- A swapper manipulates the entire processes, where as a pager manipulates individual pages of the process.
    - Bring a page into memory only when it is needed
    - Less I/O needed
    - Less memory needed
    - Faster response

- More users
- Page is needed ⇒ reference to it
- invalid reference ⇒abort
- not-in-memory ⇒ bring to memory
- ***Lazy swapper***– never swaps a page into memory unless page will be needed
- Swapper that deals with pages is a **pager.**

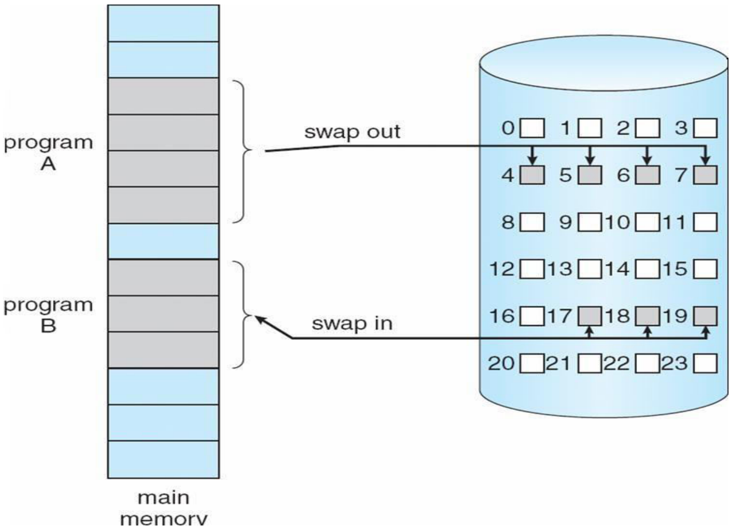
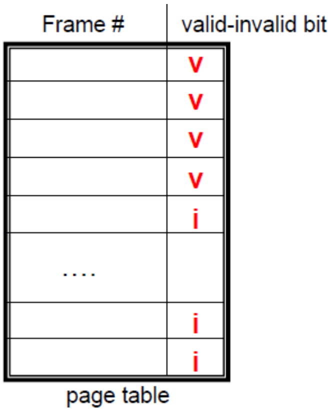


Fig: Transfer of a paged memory into continuous disk space

- **Basic concept:** Instead of swapping the whole process the pager swaps only the necessary pages in to memory. Thus it avoids reading unused pages and decreases the swap time and amount of physical memory needed.
- The valid-invalid bit scheme can be used to distinguish between the pages that are on the disk and that are in memory.
  - With each page table entry a valid–invalid bit is associated
  - (**v** ⇒ in-memory, **i**⇒not-in-memory)
  - Initially valid–invalid bit is set to **i**on all entries
  - Example of a page table snapshot:



Frame #	valid-invalid bit
	v
	v
	v
	v
	i
....	
	i
	i

page table

- During address translation, if valid–invalid bit in page table entry is **I** ⇒ page fault.
- If the bit is valid then the page is both legal and is in memory.
- If the bit is invalid then either page is not valid or is valid but is currently on the disk. Marking

a page as invalid will have no effect if the processes never access to that page. Suppose if it access the page which is marked invalid, causes a page fault trap. This may result in failure of OS to bring the desired page in to memory.

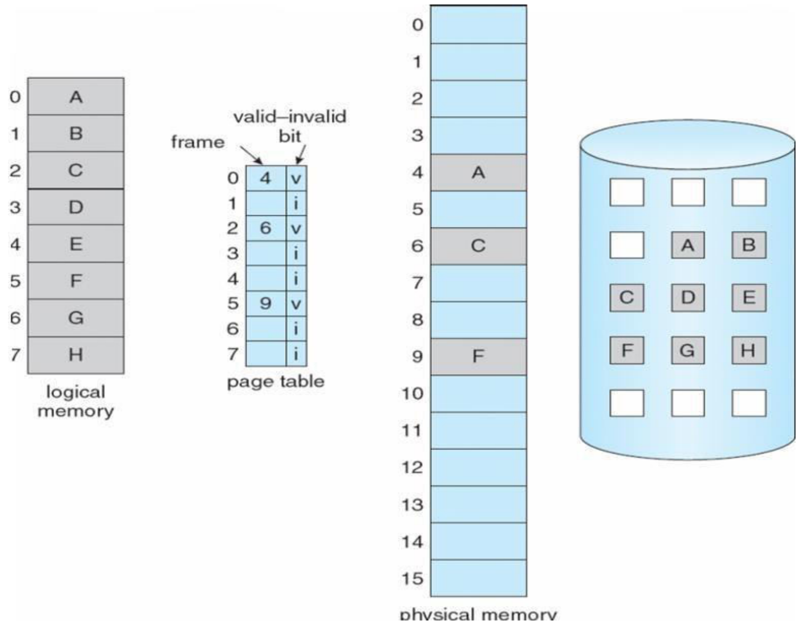


Fig: Page Table when some pages are not in main memory

Page Fault

If a page is needed that was not originally loaded up, then a ***page fault trap*** is generated.

Steps in Handling a Page Fault

1. The memory address requested is first checked, to make sure it was a valid memory request.
2. If the reference is to an invalid page, the process is terminated. Otherwise, if the page is not present in memory, it must be paged in.
3. A free frame is located, possibly from a free-frame list.
4. A disk operation is scheduled to bring in the necessary page from disk.
5. After the page is loaded to memory, the process's page table is updated with the new frame number, and the invalid bit is changed to indicate that this is now a valid page reference.
6. The instruction that caused the page fault must now be restarted from the beginning.

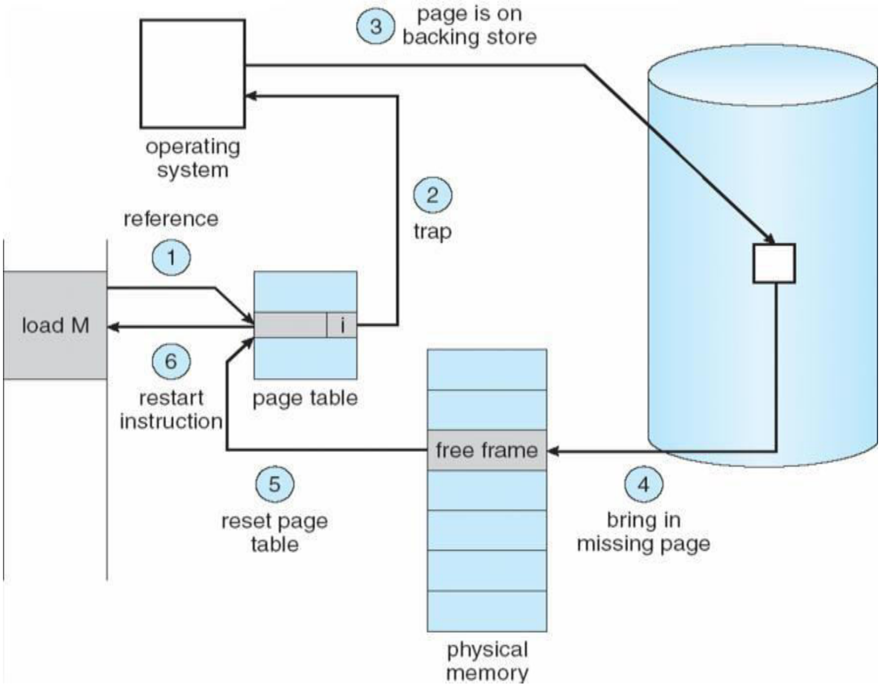


Fig: steps in handling page fault

Pure Demand Paging: Never bring a page into main memory until it is required.

- We can start executing a process without loading any of its pages into main memory.
- Page fault occurs for the non memory resident pages.
- After the page is brought into memory, process continues to execute.
- Again page fault occurs for the next page.

Hardware support: For demand paging the same hardware is required as paging and swapping.

1. Page table:-Has the ability to mark an entry invalid through valid-invalid bit.
2. Secondary memory:-This holds the pages that are not present in main memory.

Performance of Demand Paging: Demand paging can have significant effect on the performance of the computer system.

- Let P be the probability of the page fault (0<=P<=1)
- **Effective access time = (1-P) * ma + P * page fault.**
  - Where P = page fault and ma = memory access time.
- Effective access time is directly proportional to page fault rate. It is important to keep page fault rate low in demand paging.

Demand Paging Example

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8milliseconds
- EAT = (1 – p) x 200 + p (8milliseconds)

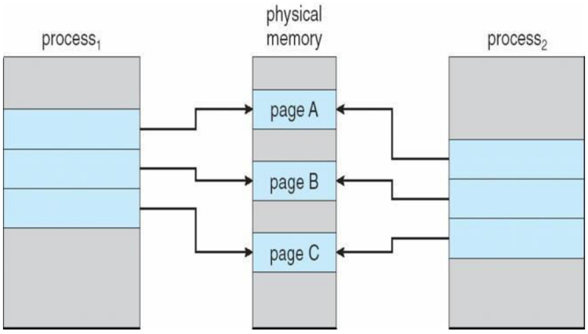
  = (1 – p x 200 + p x 8,000,000

  = **200 + p x 7,999,800**
- If one access out of 1,000 causes a page fault, then EAT = 8.2 microseconds. This is a slowdown by a factor of 40.

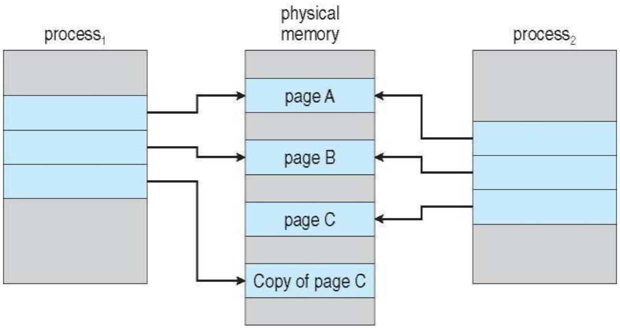
COPY-ON-WRITE

- Technique initially allows the parent and the child to share the same pages. These pages are marked as copy on- write pages i.e., if either process writes to a shared page, a copy of shared page is created.
- *Eg*:-If a child process try to modify a page containing portions of the stack; the OS recognizes them as a copy-on-write page and create a copy of this page and maps it on to the address space of the child process. So the child process will modify its copied page and not the page belonging to parent. The new pages are obtained from the pool of free pages.
- The previous contents of pages are erased before getting them into main memory. This is called **Zero – on fill demand.**

a) Before Process 1 modifies pageC



b) After process 1 modifies page C



PAGEREPLACEMENT

- Page replacement policy deals with the solution of pages in memory to be replaced by a new page that must be brought in. When a user process is executing a page fault occurs.
- The hardware traps to the operating system, which checks the internal table to see that this is a page fault and not an illegal memory access.
- The operating system determines where the derived page is residing on the disk, and this finds that there are no free frames on the list of free frames.
- When all the frames are in main memory, it is necessary to bring a new page to satisfy the page fault, replacement policy is concerned with selecting a page currently in memory to be replaced.
- The page i,e to be removed should be the page i,e least likely to be referenced in future.

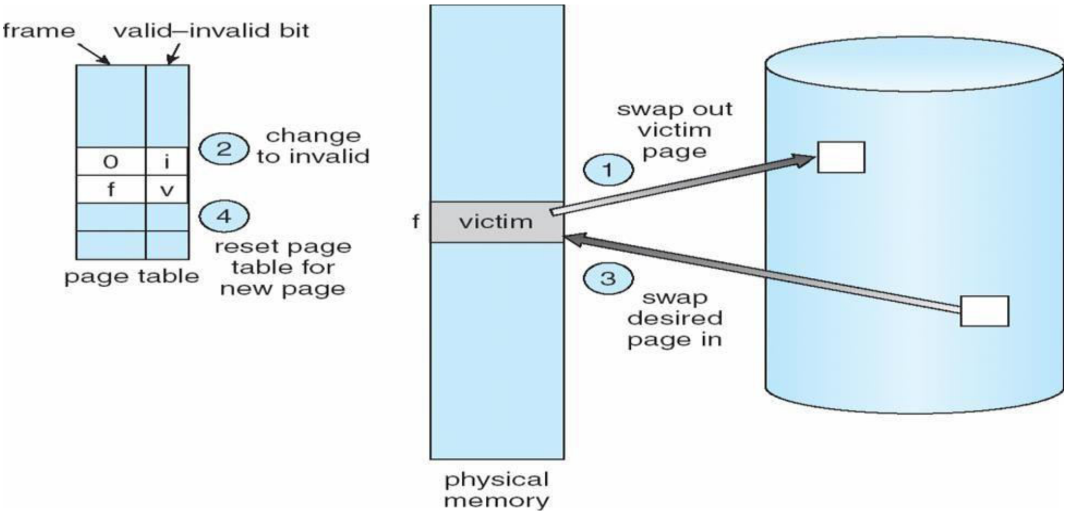


Fig: Page Replacement

Working of Page Replacement Algorithm

1. Find the location of derived page on the disk.
2. Find a free frame x If there is a free frame, use it. x Otherwise, use a replacement algorithm to select a victim.
   - Write the victim page to the disk.
   - Change the page and frame tables accordingly.
3. Read the desired page into the free frame; change the page and frame tables.
4. Restart the user process.

Victim Page

- The page that is supported out of physical memory is called victim page.
- If no frames are free, the two page transforms come (out and one in) are read. This will see the effective access time.
- Each page or frame may have a dirty (modify) bit associated with the hardware. The modify bit for a page is set by the hardware whenever any word or byte in the page is

written into, indicating that the page has been modified.

- When we select the page for replacement, we check its modify bit. If the bit is set, then the page is modified since it was read from the disk.
- If the bit was not set, the page has not been modified since it was read into memory. Therefore, if the copy of the page has not been modified we can avoid writing the memory page to the disk, if it is already there. Sum pages cannot be modified.

Modify bit/ Dirty bit :

- Each page/frame has a modify bit associated with it.
- If the page is not modified (read-only) then one can discard such page without writing it onto the disk**.** Modify bit of such page is set to0.
- Modify bit is set to 1, if the page has been modified. Such pages must be written to the disk.
- Modify bit is used to reduce overhead of page transfers – only modified pages are written to disk

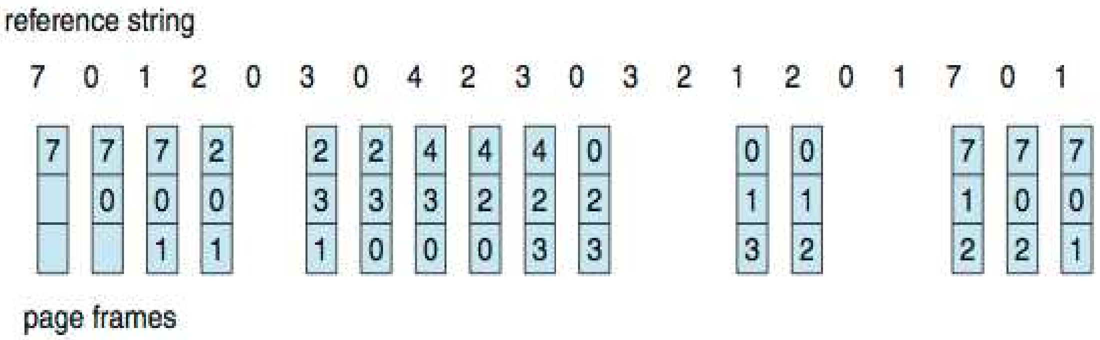
PAGE REPLACEMENT ALGORITHMS

- Want lowest page-fault rate
- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string
- In all our examples, the reference string is

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

FIFO Algorithm:

- This is the simplest page replacement algorithm. A FIFO replacement algorithm associates each page the time when that page was brought into memory.
- When a Page is to be replaced the oldest one is selected.
- We replace the queue at the head of the queue. When a page is brought into memory, we insert it at the tail of the queue.
- In the following example, a reference string is given and there are 3 free frames. There are 20 page requests, which results in 15 page faults



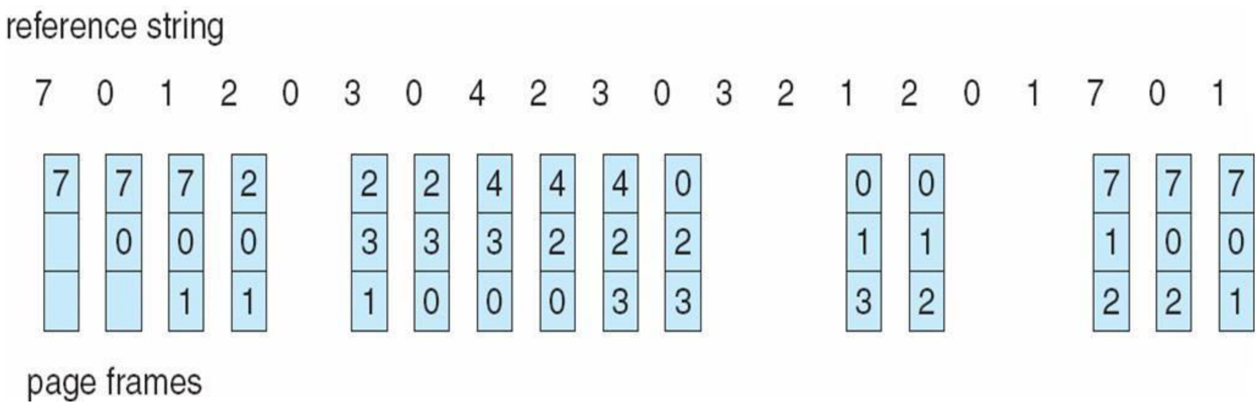
Belady’s Anomaly

- For some page replacement algorithm, the page fault may increase as the number of allocated frames increases. FIFO replacement algorithm may face this problem.

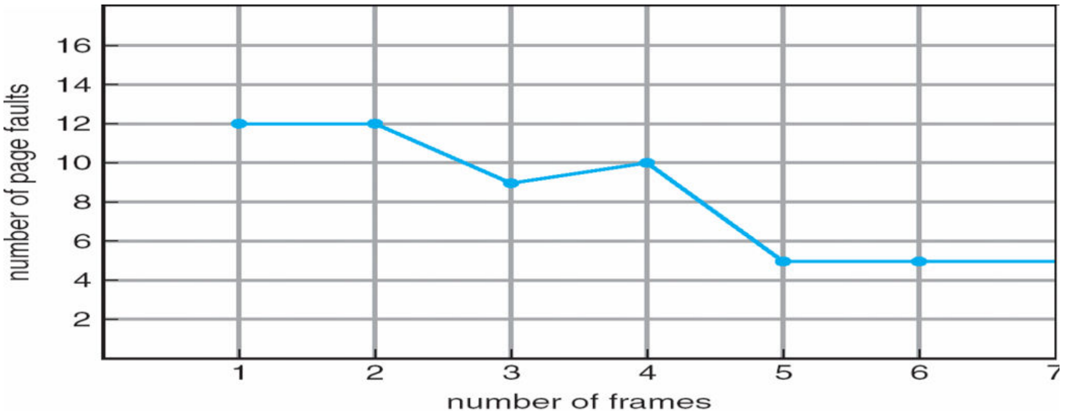
more frames ⇒ more page faults

Example: Consider the following references string with frames initially empty.

- The first three references (7,0,1) cases page faults and are brought into the empty frames.
- The next references 2 replaces page 7 because the page 7 was brought in first. x Since 0 is the next references and 0 is already in memory e has no page faults.
- The next references 3 results in page 0 being replaced so that the next references to 0 causer page fault. This will continue till the end of string. There are 15 faults all together.



FIFO Illustrating Belady’s Anomaly



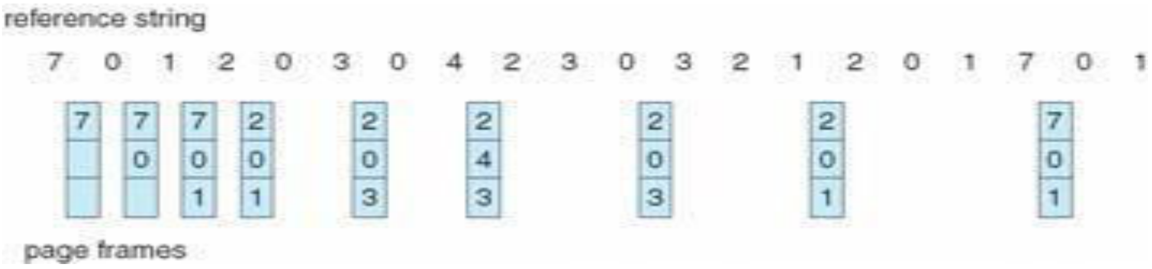
Optimal Algorithm

- Optimal page replacement algorithm is mainly to solve the problem of Belady's Anomaly.
- Optimal page replacement algorithm has the lowest page fault rate of all algorithms.
- An optimal page replacement algorithm exists and has been called OPT.

The working is simple "Replace the page that will not be used for the longest period of time"
Example: consider the following reference string

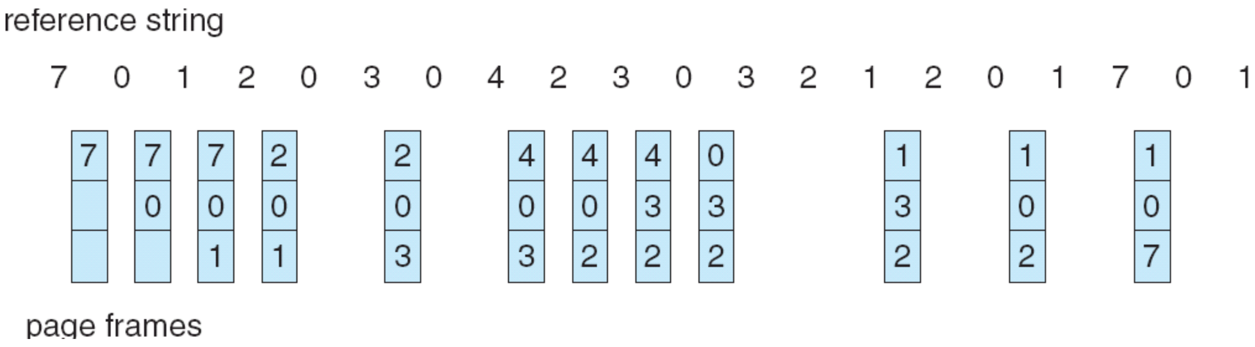
- The first three references cause faults that fill the three empty frames.
- The references to page 2 replaces page 7, because 7 will not be used until reference 18. x The page 0 will be used at 5 and page 1 at 14.
- With only 9 page faults, optimal replacement is much better than a FIFO, which had 15 faults. This algorithm is difficult t implement because it requires future knowledge of reference strings.
- Replace page that will not be used for longest period of time

Optimal Page Replacement



Least Recently Used (LRU) Algorithm

- The ***LRU (Least Recently Used)*** algorithm, predicts 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.
- Some view LRU as analogous to OPT, but here we look backwards in time instead of forwards.



The main problem to how to implement LRU is the LRU requires additional h/w assistance.

Two implementation are possible:

1. **Counters:** In this we associate each page table entry a time -of -use field, and add to the cpu a logical clock or counter. The clock is incremented for each memory reference. When a reference to a page is made, the contents of the clock register are copied to the time-of-use field in the page table entry for that page. In this way we have the time of last reference to each page we replace the page with smallest time value. The time must also be maintained when page tables are changed.
2. **Stack:** Another approach to implement LRU replacement is to keep a stack of page numbers when a page is referenced it is removed from the stack and put on to the top of stack. In this way the top of stack is always the most recently used page and the bottom in least recently used page. Since the entries are removed from the stack it is best implement by a doubly linked list. With a head and tail pointer.

Note: Neither optimal replacement nor LRU replacement suffers from Belady's Anamoly. These are called stack algorithms.

LRU-Approximation Page Replacement

- Many systems offer some degree of hardware support, enough to approximate LRU.
- In particular, many systems provide a ***reference bit*** for every entry in a page table, which is set anytime that page is accessed. Initially all bits are set to zero, and they can also all be cleared at any time. One bit distinguishes pages that have been accessed since the last clear from those that have not been accessed.

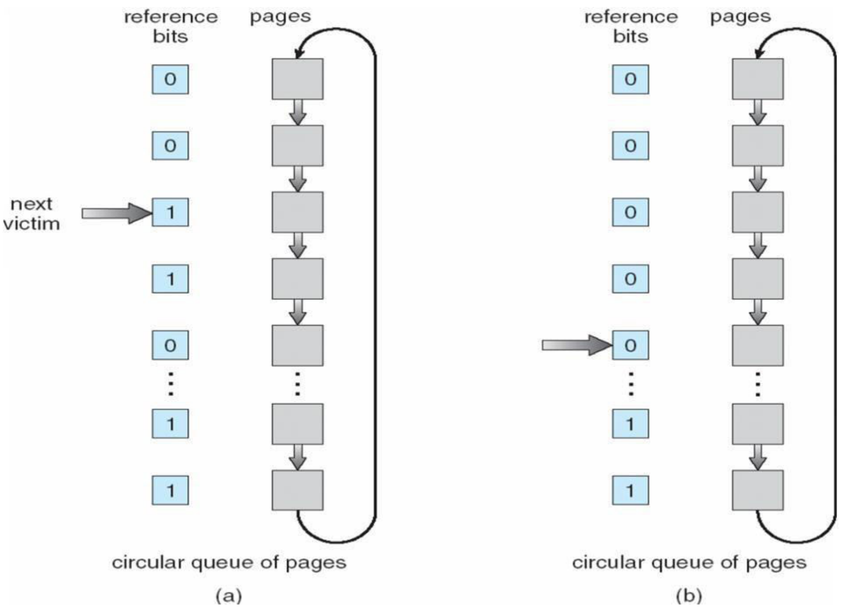
Additional-Reference-Bits Algorithm

- An 8-bit byte (reference bit) is stored for each page in a table in memory.
- At regular intervals (say, every 100 milliseconds), a timer interrupt transfers control to the operating system. The operating system shifts the reference bit for each page into the high-order bit of its 8-bit byte, shifting the other bits right by 1 bit and discarding the low-order bit.
- These 8-bit shift registers contain the history of page use for the last eight time periods.
- If the shift register contains 00000000, then the page has not been used for eight time periods.
- A page with a history register value of 11000100 has been used more recently than one with a value of 01110111.

Second- chance (clock) page replacement algorithm

- The ***second chance algorithm*** is a FIFO replacement algorithm, except the reference bit is used to give pages a second chance at staying in the page table.
- When a page must be replaced, the page table is scanned in a FIFO (circular queue) manner.
- If a page is found with its reference bit as '0', then that page is selected as the next victim.

- If the reference bitvalueis‘1’, then the page is given a second chance and its reference bit value is cleared (assigned as‘0’).
- Thus, a page that is given a second chance will not be replaced until all other pages have been replaced (or given second chances). In addition, if a page is used often, then it sets its reference bit again.
- This algorithm is also known as the ***clock*** algorithm.



Enhanced Second-Chance Algorithm

- The **enhanced second chance algorithm** looks at the reference bit and the modify bit (dirty bit) as an ordered page, and classifies pages into one of four classes:
  1. (0, 0) - Neither recently used nor modified.
  2. (0, 1) - Not recently used, but modified.
  3. (1, 0) - Recently used, but clean.
  4. (1, 1) - Recently used and modified.
- This algorithm searches the page table in a circular fashion, looking for the first page it can find in the lowest numbered category. i.e. it first makes a pass looking for a (0, 0), and then if it can't find one, it makes another pass looking for a(0,1),etc.
- The main difference between this algorithm and the previous one is the preference for replacing clean pages if possible.

Count Based Page Replacement

There is many other algorithms that can be used for page replacement, we can keep a counter of the number of references that has made to a page.

a) **LFU** (least frequently used):

This causes the page with the smallest count to be replaced. The reason for this selection is that actively used page should have a large reference count.

This algorithm suffers from the situation in which a page is used heavily during the initial phase of a process but never used again. Since it was used heavily, it has a large count and remains in memory even though it is no longer needed.

b)**MFU** Algorithm:

based on the argument that the page with the smallest count was probably just brought in and has yet to be used

ALLOCATION OF FRAMES

- The absolute minimum number of frames that a process must be allocated is dependent on system architecture.
- The maximum number is defined by the amount of available physical memory.

Allocation Algorithms

After loading of OS, there are two ways in which the allocation of frames can be done to the processes.

1. **Equal Allocation**- If there are m frames available and n processes to share them, each process gets m / n frames, and the left over's are kept in a free-frame buffer pool.
2. **Proportional Allocation** - Allocate the frames proportionally depending on the size of the process. If the size of process i is S_i, and S is the sum of size of all processes in the system, then the allocation for process P_i is $a_i = m * S_i / S$. where m is the free frames available in the system.

- Consider a system with a 1KB frame size. If a small student process of 10 KB and an interactive database of 127 KB are the only two processes running in a system with 62 free frames.
- with proportional allocation, we would split 62 frames between two processes, as follows

  m=62, S = (10+127)=137

  Allocation for process 1 = 62 X 10/137 ~ 4 Allocation for process 2 = 62 X 127/137 ~57

  Thus allocates 4 frames and 57 frames to student process and database respectively.
- Variations on proportional allocation could consider priority of process rather than just their size.

Global versus Local Allocation

- Page replacement can occur both at local or global level.
- With local replacement, the number of pages allocated to a process is fixed, and page replacement occurs only amongst the pages allocated to this process.
- With global replacement, any page may be a potential victim, whether it currently belongs to the process seeking a free frame or not.
- Local page replacement allows processes to better control their own page fault rates, and leads to more consistent performance of a given process over different system load levels.

- Global page replacement is over all more efficient, and is the more commonly used approach.

Non-Uniform Memory Access (New)

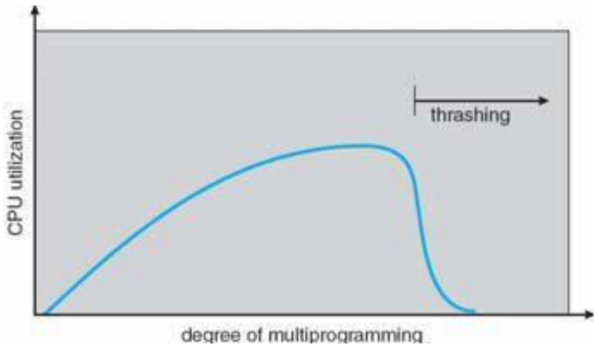
- Usually the time required to access all memory in a system is equivalent.
- This may not be the case in multiple-processor systems, especially where each CPU is physically located on a separate circuit board which also holds some portion of the overall system memory.
- In such systems, CPU s can access memory that is physically located on the same board much faster than the memory on the other boards.
- The basic solution is akin to processor affinity - At the same time that we try to schedule processes on the same CPU to minimize cache misses, we also try to allocate memory for those processes on the same boards, to minimize access times.

THRASHING

- If the number of frames allocated to a low-priority process falls below the minimum number required by the computer architecture then we suspend the process execution.
- A process is thrashing if it is spending more time in paging than executing.
- If the processes do not have enough number of frames, it will quickly page fault. During this it must replace some page that is not currently in use. Consequently it quickly faults again and again.
- The process continues to fault, replacing pages for which it then faults and brings back. This high paging activity is called thrashing. The phenomenon of excessively moving pages back and forth b/w memory and secondary has been called ***thrashing***.

Cause of Thrashing

- Thrashing results in severe performance problem.
- The operating system monitors the cpu utilization is low. We increase the degree of multi programming by introducing new process to the system.
- A global page replacement algorithm replaces pages with no regards to the process to which they belong.



The figure shows the thrashing

- As the degree of multi programming increases, more slowly until a maximum is reached. If the degree of multi programming is increased further thrashing sets in and the cpu utilization drops sharply.

- At this point, to increases CPU utilization and stop thrashing, we must increase degree of multiprogramming.
- we can limit the effect of thrashing by using a local replacement algorithm. To prevent thrashing, we must provide a process as many frames as it needs.

Locality of Reference:

- As the process executes it moves from locality to locality.
- A locality is a set of pages that are actively used.
- A program may consist of several different localities, which may overlap.
- Locality is caused by loops in code that find to reference arrays and other data structures by indices.
- The ordered list of page number accessed by a program is called reference string.
- Locality is of two types :

  1. spatial locality 2. temporal locality

Working set model

- Working set model algorithm uses the current memory requirements to determine the number of page frames to allocate to the process, an informal definition is “the collection of pages that a process is working with and which must be resident if the process to avoid thrashing”. The idea is to use the recent needs of a process to predict its future reader.
- The working set is an approximation of programs locality. Ex: given a sequence of memory reference, if the working set window size to memory references, then working set at time t1 is{1,2,5,6,7} and at t2 is changed to {3,4}
- At any given time, all pages referenced by a process in its last 4 seconds of execution are considered to compromise its working set.
- A process will never execute until its working set is resident in main memory.
- Pages outside the working set can be discarded at any movement.
- Working sets are not enough and we must also introduce balance set.
  - If the sum of the working sets of all the run able process is greater than the size of memory the refuse some process for a while.
  - Divide the run able process into two groups, active and inactive. The collection of active set is called the balance set. When a process is made active its working set is loaded.
  - Some algorithm must be provided for moving process into and out of the balance set. As a working set is changed, corresponding change is made to the balance set.
  - Working set presents thrashing by keeping the degree of multi programming as high as possible. Thus if optimizes the CPU utilization. The main disadvantage of this is keeping track of the working set.

Page-Fault Frequency

- When page- fault rate is too high, the process needs more frames and when it is too low, the process may have too many frames.
- The upper and lower bounds can be established on the page-fault rate. If the actual page- fault rate exceeds the upper limit, allocate the process another frame or suspend the process.
- If the page-fault rate falls below the lower limit, remove a frame from the process. Thus, we can directly measure and control the page-fault rate to prevent thrashing.

FILE CONCEPT

FILE:

- *A* file is a named collection of related information that is recorded on secondary storage.
- The information in a file is defined by its creator. Many different types of information may be stored in a file source programs, object programs, executable programs, numeric data, text, payroll records, graphic images, sound recordings, and so on.

A file has a certain defined which depends on its type.

- *A text* file is a sequence of characters organized into lines.
- *A source* file is a sequence of subroutines and functions, each of which is further organized as declarations followed by executable statements.
- An *object* file is a sequence of bytes organized into blocks understandable by the system's linker.
- An *executable* file is a series of code sections that the loader can bring into memory and execute.

File Attributes

- A file is named, for the convenience of its human users, and is referred to by its name. A name is usually a string of characters, such as *example*.c
- When a file is named, it becomes independent of the process, the user, and even the system that created it.

A file's attributes vary from one operating system to another but typically consist of these:

- **Name:** The symbolic file name is the only information kept in human readable form.
- **Identifier:** This unique tag, usually a number, identifies the file within the file system; it is the non-human-readable name for the file.
- **Type:** This information is needed for systems that support different types of files.
- **Location:** This information is a pointer to a device and to the location of the file on that device.
- **Size:** The current size of the file (in bytes, words, or blocks) and possibly the maximum allowed size are included in this attribute.
- **Protection:** Access-control information determines who can do reading, writing, executing, and so on.
- **Time, date, and user identification:** This information may be kept for creation, last modification, and last use. These data can be useful for protection, security, and usage monitoring.

The information about all files is kept in the directory structure, which also resides on secondary storage. Typically, a directory entry consists of the file's name and its unique identifier. The identifier in turn locates the other file attributes.

File Operations

A file is an abstract data type. To define a file properly, we need to consider the operations that can be performed on files.

1. **Creating a file:**Two steps are necessary to create a file,
    a) Space in the file system must be found for the file.
    b) An entry for the new file must be made in the directory.
2. **Writing a file:**To write a file, we make a system call specifying both the name of the file and the information to be written to the file. Given the name of the file, the system searches the directory to find the file's location. The system must keep a write pointer to the location in the file where the next write is to take place. The write pointer must be updated whenever a write occurs.
3. **Reading a file:**To read from a file, we use a system call that specifies the name of the file and where the next block of the file should be put. Again, the directory is searched for the associated entry, and the system needs to keep a read pointer to the location in the file where the next read is to take place. Once the read has taken place, the read pointer is updated. Because a process is usually either reading from or writing to a file, the current operation location can be kept as a per-process current file-position pointer.
4. **Repositioning within a file:**The directory is searched for the appropriate entry, and the current-file-position pointer is repositioned to a given value. Repositioning within a file need not involve any actual I/0. This file operation is also known as files seek.
5. **Deleting a file:**To delete a file, search the directory for the named file. Having found the associated directory entry, then release all file space, so that it can be reused by other files, and erase the directory entry.
6. **Truncating a file:**The user may want to erase the contents of a file but keep its attributes. Rather than forcing the user to delete the file and then recreate it, this function allows all attributes to remain unchanged but lets the file be reset to length zero and its file space released.

- Other common operations include appending new information to the end of an existing file and renaming an existing file.
- Most of the file operations mentioned involve searching the directory for the entry associated with the named file.
- To avoid this constant searching, many systems require that an open () system call be made before a file is first used actively.
- The operating system keeps a small table, called the **open file table** containing information about all open files. When a file operation is requested, the file is specified via an index into this table, so no searching is required.

- The implementation of the open() and close() operations is more complicated in an environment where several processes may open the file simultaneously
- The operating system uses two levels of internal tables:
    1. A per-process table
    2. A system-wide table

The per-process table:

- Tracks all files that a process has open. Stored in this table is information regarding the use of the file by the process.
- Each entry in the per-process table in turn points to a system-wide open-file table.

The system-wide table

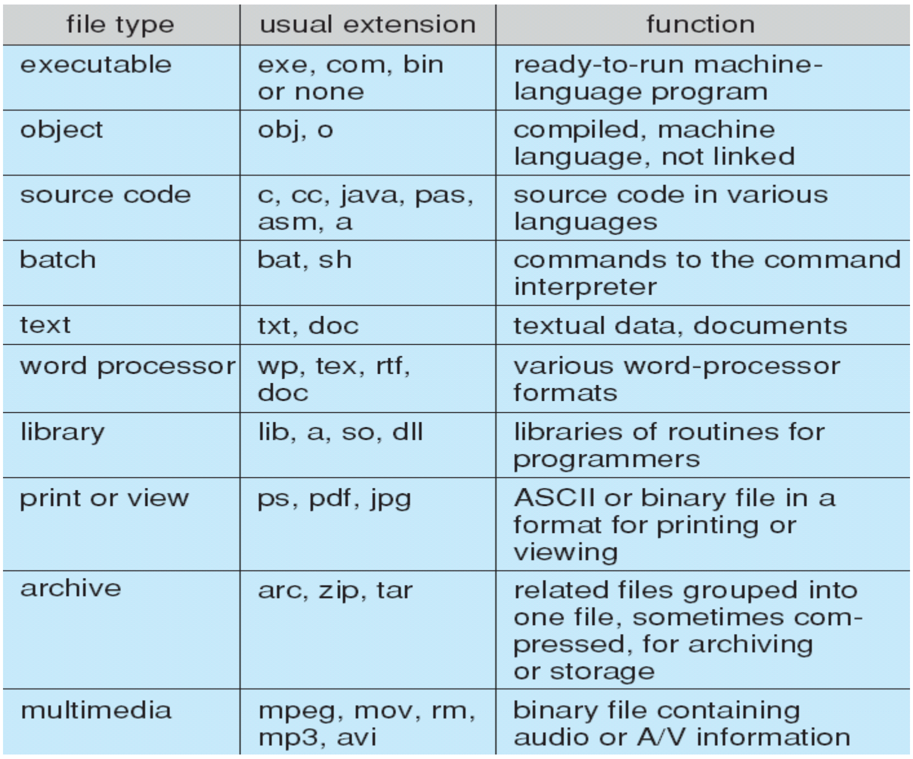
- contains process-independent information, such as the location of the file on disk, access dates, and file size. Once a file has been opened by one process, the system-wide table includes an entry for the file.

Several pieces of information are associated with an open file.

1. **File pointer:** On systems that do not include a file offset as part of the read() and write() system calls, the system must track the last read write location as a current-file-position pointer. This pointer is unique to each process operating on the file and therefore must be kept separate from the on-disk file attributes.
2. **File-open count:** As files are closed, the operating system must reuse its open- file table entries, or it could run out of space in the table. Because multiple processes may have opened a file, the system must wait for the last file to close before removing the open-file table entry. The file-open counter tracks the number of opens and closes and reaches zero on the last close. The system can then remove the entry.
3. **Disk location of the file:**Most file operations require the system to modify data within the file. The information needed to locate the file on disk is kept in memory so that the system does not have to read it from disk for each operation.
4. **Access rights:** Each process opens a file in an access mode. This information is stored on the per-process table so the operating system can allow or deny subsequent I/0 requests.

File Types

- The operating system should recognize and support file types. If an operating system recognizes the type of a file, it can then operate on the file in reasonable ways.
- A common technique for implementing file types is to include the type as part of the file name. The name is split into two parts-**a name and an extension**, usually separated by a period character
- The system uses the extension to indicate the type of the file and the type of operations that can be done on that file.



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 compressed, for archiving or storage
multimedia	mpeg, mov, rm, mp3, avi	binary file containing audio or A/V information

File Structure

- File types also can be used to indicate the internal structure of the file. For instance source and object files have structures that match the expectations of the programs that read them. Certain files must conform to a required structure that is understood by the operating system.

  For example: the operating system requires that an executable file have a specific structure so that it can determine where in memory to load the file and what the location of the first instruction is.
- The operating system support multiple file structures: the resulting size of the operating system also increases. If the operating system defines five different file structures, it needs to contain the code to support these file structures.
- It is necessary to define every file as one of the file types supported by the operating system. When new applications require information structured in ways not supported by

the operating system, severe problems may result.

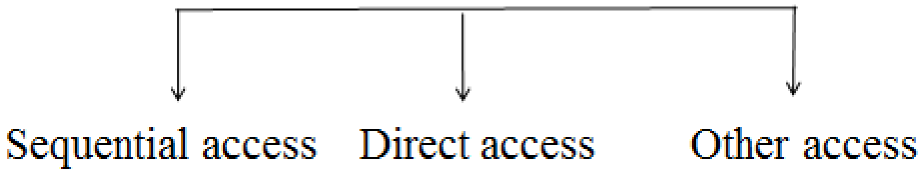
- Example: The Macintosh operating system supports a minimal number of file structures. It expects files to contain two parts: a resource fork and data fork.
    - **The resource fork** contains information of interest to the user.
    - **The data fork** contains program code or data

Internal File Structure

- Internally, locating an offset within a file can be complicated for the operating system. Disk systems typically have a well-defined block size determined by the size of a sector.
- All disk I/0 is performed in units of one block (physical record), and all blocks are the same size. It is unlikely that the physical record size will exactly match the length of the desired logical record.
- Logical records may even vary in length. Packing a number of logical records into physical blocks is a common solution to this problem.

ACCESS METHODS

- Files store information. When it is used, this information must be accessed and read into computer memory. The information in the file can be accessed in several ways.
- Some of the common methods are:



1. Sequential methods

- The simplest access method is sequential methods. Information in the file is processed in order, one record after the other.
- Reads and writes make up the bulk of the operations on a file.
- A read operation (next-reads) reads the next portion of the file and automatically advances a file pointer, which tracks the I/O location
- The write operation (write next) appends to the end of the file and advances to the end of the newly written material.
- A file can be reset to the beginning and on some systems, a program may be able to skip forward or backward n records for some integer n-perhaps only for n =1.

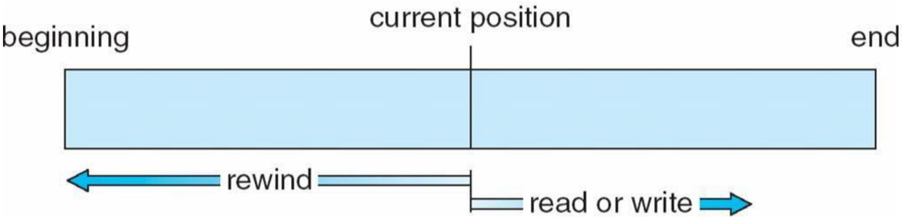
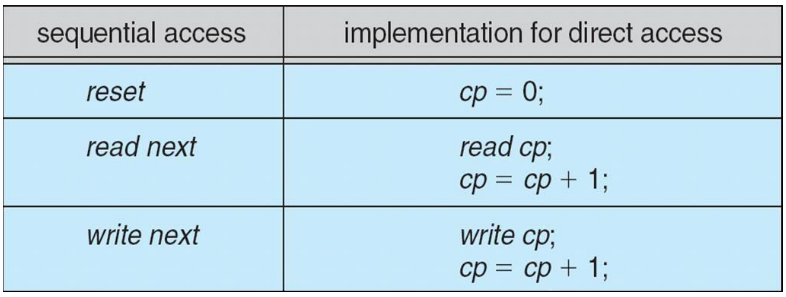


Figure: Sequential-access file.

2. Direct Access

- A file is made up of fixed length logical records that allow programs to read and write records rapidly in no particular order.
- The direct-access method is based on a disk model of a file, since disks allow random access to any file block. For direct access, the file is viewed as a numbered sequence of blocks or records.
- Example: if we may read block 14, then read block 53, and then write block 7. There are no restrictions on the order of reading or writing for a direct-access file.
- Direct-access files are of great use for immediate access to large amounts of information such as Databases, where searching becomes easy and fast.
- For the direct-access method, the file operations must be modified to include the block number as a parameter. Thus, we have read n, where n is the block number, rather than read next, and ·write n rather than write next.
- An alternative approach is to retain read next and write next, as with sequential access, and to add an operation position file to n, where n is the block number. Then, to affect a read n, we would position to n and then read next.



sequential access	implementation for direct access
reset	*cp* = 0;
read next	*read cp*; *cp* = *cp* + 1;
write next	*write cp*; *cp* = *cp* + 1;

Figure: Simulation of sequential access on a direct-access file.

3. Other Access Methods:

- Other access methods can be built on top of a direct-access method. These methods generally involve the construction of an index for the file.
- The **Index**, is like an index in the back of a book contains pointers to the various blocks. To find a record in the file, we first search the index and then use the pointer to access the file directly and to find the desired record.

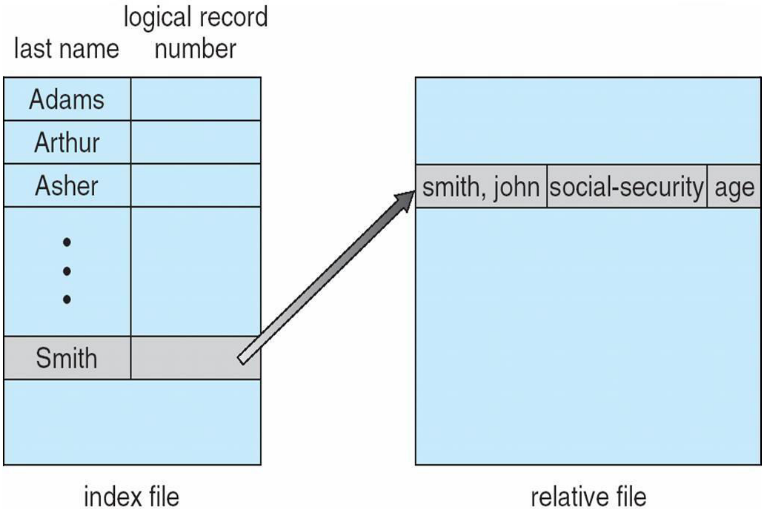


Figure: Example of index and relative files

DIRECTORY AND DISK STRUCTURE

- Files are stored on random-access storage devices, including hard disks, optical disks, and solid state (memory-based) disks.
- A storage device can be used in its entirety for a file system. It can also be subdivided for finer-grained control
- Disk can be subdivided into partitions. Each disks or partitions can be RAID protected against failure.
- Partitions also known as minidisks or slices. Entity containing file system known as a volume. Each volume that contains a file system must also contain information about the files in the system. This information is kept in entries in a **device directory** or **volume table of contents.**

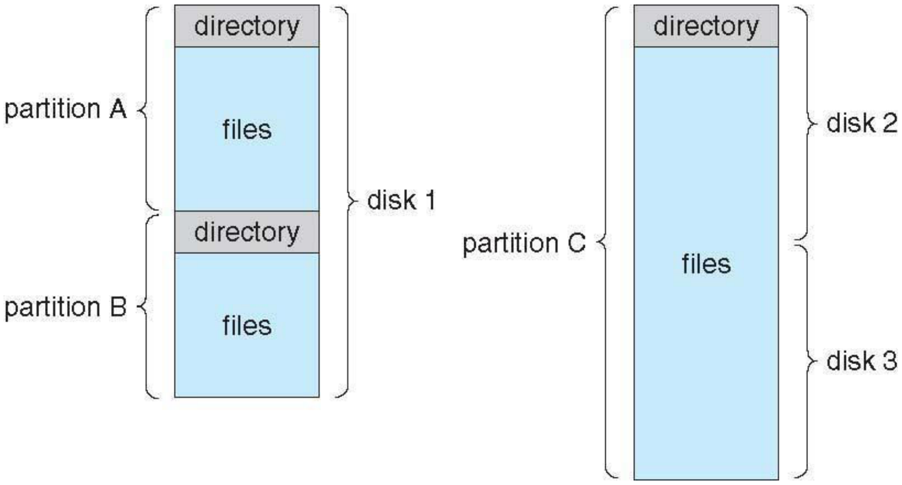


Figure: A Typical File-system Organization

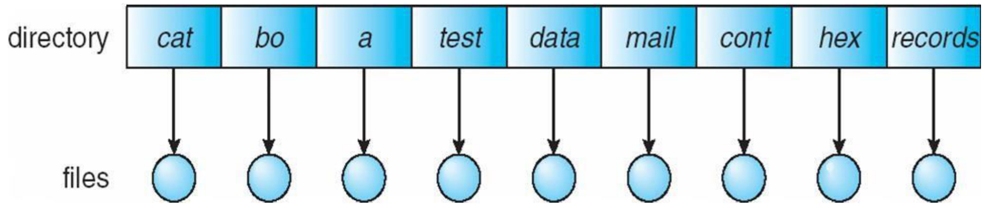
Directory Overview

- The directory can be viewed as a symbol table that translates file names into their directory entries. A directory contains information about the files including attributes location and ownership. To consider a particular directory structure, certain operations on the directory have to be considered:
  - **Search for a file**: Directory structure is searched for a particular file in directory. Files have symbolic names and similar names may indicate a relationship between files. Using this similarity it will be easy to find all whose name matches a particular pattern.
  - **Create a file**: New files needed to be created and added to the directory.
  - **Delete a file:**When a file is no longer needed, then it is able to remove it from thedirectory.
  - **List a directory**: It is able to list the files in a directory and the contents of the directory entry for each file in the list.
  - **Rename a file**: Because the name of a file represents its contents to its users, It is possible to change the name when the contents or use of the file changes. Renaming a file may also allow its position within the directory structure to be changed.
  - **Traverse the file system**: User may wish to access every directory and every file within a directory structure. To provide reliability the contents and structure of the entire file system is saved at regular intervals.

- The most common schemes for defining the logical structure of a directory are described below
  1. Single-level Directory
  2. Two-Level Directory
  3. Tree-Structured Directories
  4. Acyclic-Graph Directories
  5. General Graph Directory

1. Single-level Directory

- The simplest directory structure is the single-level directory. All files are contained in the same directory, which is easy to support and understand

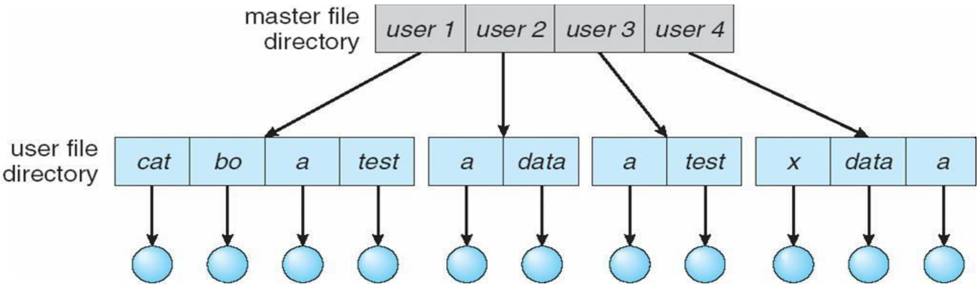


- A single-level directory has significant limitations, when the number of files increases or when the system has more than one user.

- As directory structure is single, uniqueness of file name has to be maintained, which is difficult when there are multiple users.
- Even a single user on a single-level directory may find it difficult to remember the names of all the files as the number of files increases.
- It is not uncommon for a user to have hundreds of files on one computer system and an equal number of additional files on another system. Keeping track of so many files is a daunting task.

2. Two-Level Directory

- In the two-level directory structure, each user has its own **user file directory** (UFD). The UFDs have similar structures, but each lists only the files of a single user.
- When a user refers to a particular file, only his own UFD is searched. Different users may have files with the same name, as long as all the file names within each UFD are unique.
- To create a file for a user, the operating system searches only that user's UFD to ascertain whether another file of that name exists. To delete a file, the operating system confines its search to the local UFD thus; it cannot accidentally delete another user's file that has the same name.
- When a user job starts or a user logs in, the system's Master file directory (MFD) is searched. The MFD is indexed by user name or account number, and each entry points to the UFD for that user.



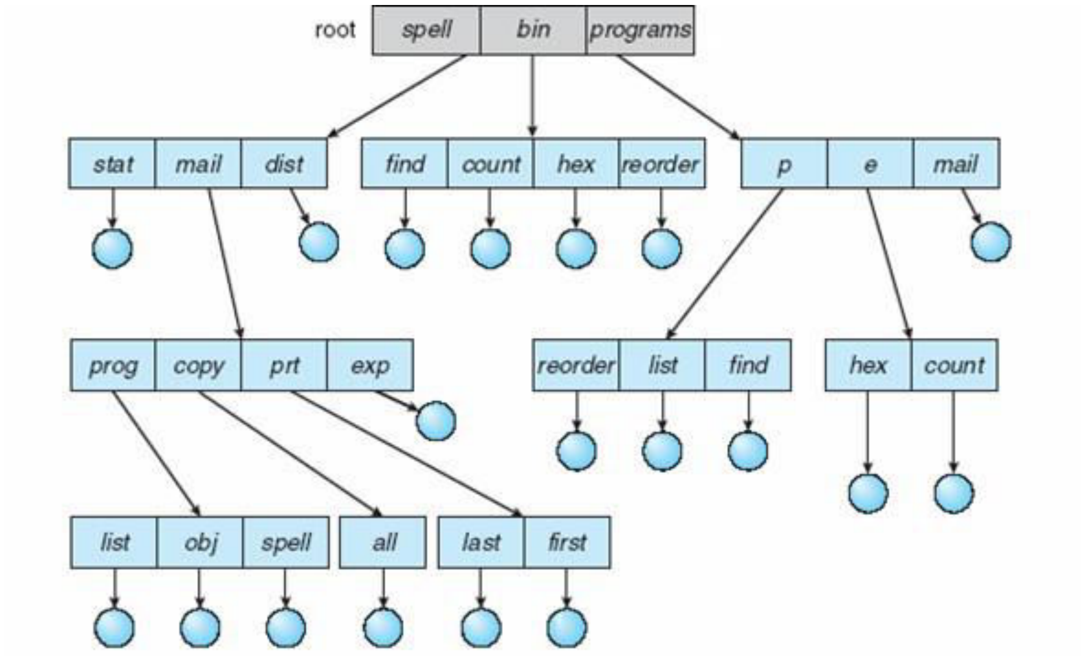
- **Advantage:**
  - No file name-collision among different users.
  - Efficient searching.
- **Disadvantage**
  - Users are isolated from one another and can’t cooperate on the same task.

3. Tree Structured Directories

- A tree is the most common directory structure.
- The tree has a root directory, and every file in the system has a unique path name.
- A directory contains a set of files or subdirectories. A directory is simply another file, but it is treated in a special way.
- All directories have the same internal format. One bit in each directory entry defines the entry as a file (0) or as a subdirectory (1). Special system calls are used to create and

delete directories.

- **Two types of path-names:**
  1. Absolute path-name: begins at the root.
  2. Relative path-name: defines a path from the current directory.



How to delete directory?

1. To delete an empty directory: Just delete the directory.
2. To delete a non-empty directory:
   - First, delete all files in the directory.
   - If any subdirectories exist, this procedure must be applied recursively to them.

Advantage:

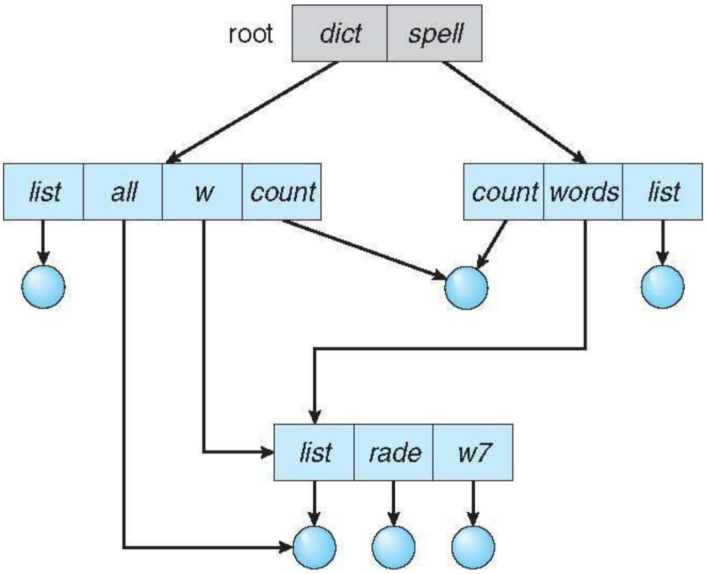
- Users can be allowed to access the files of other users.

Disadvantages:

- A path to a file can be longer than a path in a two-level directory.
- Prohibits the sharing of files (or directories).

4. Acyclic Graph Directories

- The common subdirectory should be shared. A shared directory or file will exist in the file system in two or more places at once. A tree structure prohibits the sharing of files or directories.
- An acyclic graph is a graph with no cycles. It allows directories to share subdirectories and files.



- The same file or subdirectory may be in two different directories. The acyclic graph is a natural generalization of the tree-structured directory scheme.

Two methods to implement shared-files (or subdirectories):

1. Create a new directory-entry called a link. A link is a pointer to another file (or subdirectory).
2. Duplicate all information about shared-files in both sharing directories.

Two problems:

1. A file may have multiple absolute path-names.
2. Deletion may leave dangling-pointers to the non-existent file.

Solution to deletion problem:

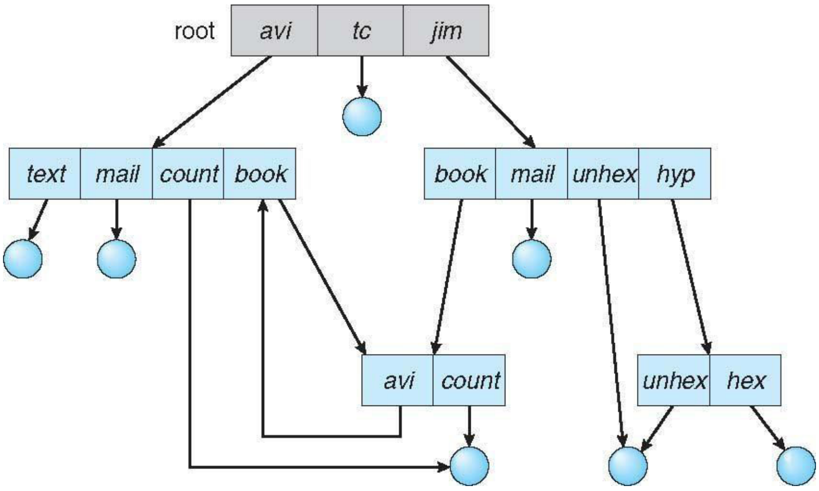
1. Use back-pointers: Preserve the file until all references to it are deleted.
2. With symbolic links, remove only the link, not the file. If the file itself is deleted, the link can be removed.

5. General Graph Directory

- **Problem:** If there are cycles, we want to avoid searching components twice.
- **Solution:** Limit the no. of directories accessed in a search.
- **Problem:** With cycles, the reference-count may be non-zero even when it is no longer possible to refer to a directory (or file). (A value of 0 in the reference count means that there are no more references to the file or directory, and the file can be deleted).
- **Solution:** Garbage-collection scheme can be used to determine when the last reference has been deleted.

Garbage collection involves

- First pass traverses the entire file-system and marks everything that can be accessed.
- A second pass collects everything that is not marked onto a list of free-space



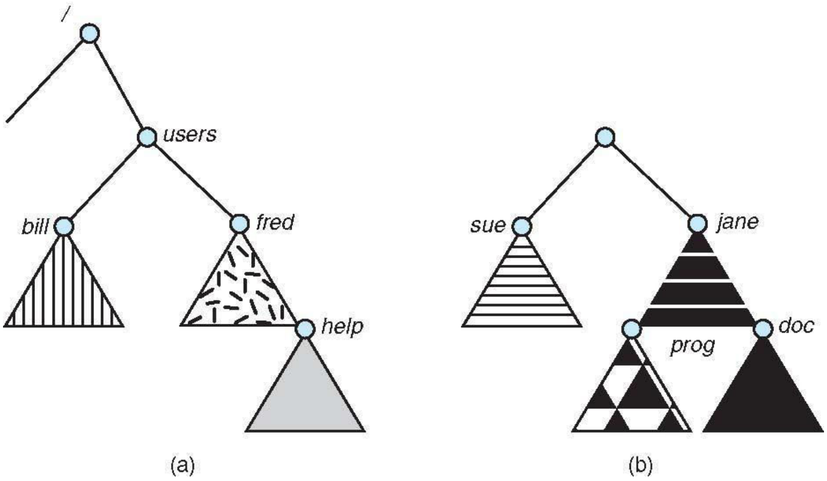
FILE SYSTEM MOUNTING

- A file must be *opened* before it is used, a file system must be *mounted* before it can be available to processes on the system
- **Mount Point:** The location within the file structure where the file system is to be attached.

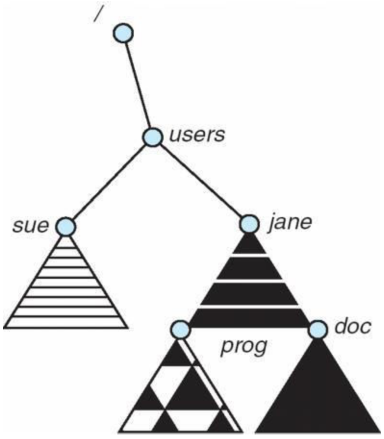
The mounting procedure:

- The operating system is given the name of the device and the mount point.
- The operating system verifies that the device contains a valid file system. It does so by asking the device driver to read the device directory and verifying that the directory has the expected format
- The operating system notes in its directory structure that a file system is mounted at the specified mount point.

To illustrate file mounting, consider the file system shown in figure. The triangles represent sub-trees of directories that are of interest



- Figure (a) shows an existing file system,
- while Figure 1(b) shows an un-mounted volume residing on */device/dsk.* At this point, only the files on the existing file system can be accessed.



- Above figure shows the effects of mounting the volume residing on */device/dsk* over */users.*
- If the volume is un-mounted, the file system is restored to the situation depicted in first Figure.

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.

Multiple Users

File-sharing can be done in 2 ways:

1. The system can allow a user to access the files of other users by default or
2. The system may require that a user specifically grant access.

To implement file-sharing, the system must maintain more file- & directory-attributes than on a single-user system.

Most systems use concepts of file owner and group.

1. Owner

- The user who may change attributes & grant access and has the most control over the file (or directory).
- Most systems implement owner attributes by managing a list of user-names and user IDs

2. Group

- The group attribute defines a subset of users who can share access to the file.
- Group functionality can be implemented as a system-wide list of group-names and group IDs.
- Exactly which operations can be executed by group-members and other users is definable by the file's owner.
- The owner and group IDs of files are stored with the other file-attributes and can be used to allow/deny requested operations.

Remote File Systems

It allows a computer to mount one or more file-systems from one or more remote- machines.

There are three methods:

1. Manually transferring files between machines via programs like ftp.
2. Automatically DFS (Distributed file-system): remote directories are visible from a local machine.
3. Semi-automatically via www (World Wide Web): A browser is needed to gain access to the remote files, and separate operations (a wrapper for ftp) are used to transfer files.

ftp is used for both anonymous and authenticated access. **Anonymous access** allows a user to transfer files without having an account on the remote system.

Client Server Model

- Allows clients to mount remote file-systems from servers.
- The machine containing the files is called the **server**. The machine seeking access to the files is called the **client**.
- A server can serve multiple clients, and a client can use multiple servers.
- The server specifies which resources (files) are available to which clients.
- A client can be specified by a network-name such as an IP address.

Disadvantage:

- Client identification is more difficult.
- In UNIX and its NFS (network file-system), authentication takes place via the client networking information by default.
- Once the remote file-system is mounted, file-operation requests are sent to the server via the DFS protocol.

Distributed Information Systems

- Provides unified access to the information needed for remote computing.
- The DNS (domain name system) provides hostname-to-network address translations.
- Other distributed info. systems provide username/password space for a distributed facility

Failure Modes

- Local file-systems can fail for a variety of reasons such as failure of disk (containing the file-system), corruption of directory-structure & cable failure.
- Remote file-systems have more failure modes because of the complexity of network-systems.
- The network can be interrupted between 2 hosts. Such interruptions can result from hardware failure, poor hardware configuration or networking implementation issues.
- DFS protocols allow delaying of file-system operations to remote-hosts, with the hope that the remote-host will become available again.
- To implement failure-recovery, some kind of state information may be maintained on both the client and the server.

Consistency Semantics

- These represent an important criterion of evaluating file-systems that supports file-sharing. These specify how multiple users of a system are to access a shared-file simultaneously.
- In particular, they specify when modifications of data by one user will be observed by other users.
- These semantics are typically implemented as code with the file-system.

- These are directly related to the process-synchronization algorithms.
- A successful implementation of complex sharing semantics can be found in the Andrew file-system (AFS).

UNIX Semantics

- UNIX file-system (UFS) uses the following consistency semantics:
    1. Writes to an open-file by a user are visible immediately to other users who have this file opened.
    2. One mode of sharing allows users to share the pointer of current location into a file. Thus, the advancing of the pointer by one user affects all sharing users.
- A file is associated with a single physical image that is accessed as an exclusive resource.
- Contention for the single image causes delays in user processes.

Session Semantics

The AFS uses the following consistency semantics:

1. Writes to an open file by a user are not visible immediately to other users that have the same file open.
2. Once a file is closed, the changes made to it are visible only in sessions starting later. Already open instances of the file do not reflect these changes.

- A file may be associated temporarily with several (possibly different) images at the same time.
- consequently, multiple users are allowed to perform both read and write accesses concurrently on their images of the file, without delay.
- Almost no constraints are enforced on scheduling accesses.

Immutable Shared Files Semantics

- Once a file is declared as shared by its creator, it cannot be modified.
- An immutable file has 2 key properties:
    1. File-name may not be reused
    2. File-contents may not be altered.
- Thus, the name of an immutable file signifies that the contents of the file are fixed.
- The implementation of these semantics in a distributed system is simple, because the sharing is disciplined

PROTECTION

- When information is stored in a computer system, we want to keep it safe from physical damage (reliability) and improper access (protection).
- Reliability is generally provided by duplicate copies of files.
- For a small single-user system, we might provide protection by physically removing the floppy disks and locking them in a desk drawer.
- File owner/creator should be able to control what can be done and by whom.

Types of Access

- Systems that do not permit access to the files of other users do not need protection. This is too extreme, so controlled-access is needed.
- Following operations may be controlled:
  1. *Read:* Read from the file.
  2. *Write:* Write or rewrite the file.
  3. *Execute:* Load the file into memory and execute it.
  4. *Append*: Write new information at the end of the file.
  5. *Delete:* Delete the file and tree its space for possible reuse.
  6. *List:* List the name and attributes of the file.

Access Control

- Common approach to protection problem is to make access dependent on identity of user.
- Files can be associated with an ACL (access-control list) which specifies username and types of access for each user.

Problems:

1. Constructing a list can be tedious.
2. Directory-entry now needs to be of variable-size, resulting in more complicated space management.

Solution:

- These problems can be resolved by combining ACLs with an 'owner, group, universe' access control scheme
- To reduce the length of the ACL, many systems recognize 3 classifications of users:
  1. *Owner:* The user who created the file is the owner.
  2. *Group:* A set of users who are sharing the file and need similar access is a group.
  3. *Universe:* All other users in the system constitute the universe.

Other Protection Approaches

- A password can be associated with each file.
- **Disadvantages:**
    1. The no. of passwords you need to remember may become large.
    2. If only one password is used for all the files, then all files are accessible if it is discovered.
    3. Commonly, only one password is associated with all of the user's files, so protection is all-or nothing.

- In a multilevel directory-structure, we need to provide a mechanism for directory protection.
- The directory operations that must be protected are different from the File-operations:
    1. Control creation & deletion of files in a directory.
    2. Control whether a user can determine the existence of a file in a directory.

IMPLEMENTATION OF FILE SYSTEM

FILE SYSTEM STRUCTURE

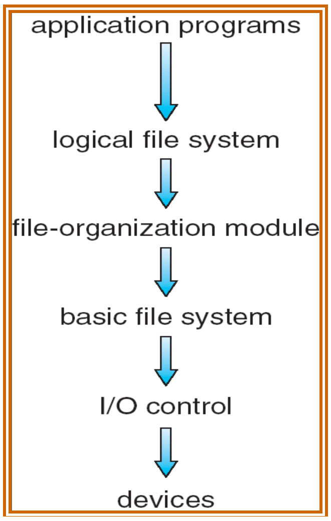
- Disks provide the bulk of secondary-storage on which a file-system is maintained.

The disk is a suitable medium for storing multiple files.

- This is because of two characteristics
  1. A disk can be rewritten in place; it is possible to read a block from the disk, modify the block, and write it back into the same place.
  2. A disk can access directly any block of information it contains. Thus, it is simple to access any file either sequentially or randomly, and switching from one file to another requires only moving the read-write heads and waiting for the disk to rotate.
- To improve I/O efficiency, I/O transfers between memory and disk are performed in units of blocks. Each block has one or more sectors. Depending on the disk drive, sector-size varies from 32 bytes to 4096 bytes. The usual size is 512 bytes.
- File-systems provide efficient and convenient access to the disk by allowing data to be stored, located, and retrieved easily
- Design problems of file-systems:
  1. Defining how the file-system should look to the user.
  2. Creating algorithms & data-structures to map the logical file-system onto the physical secondary-storage devices.

Layered File Systems:

- The file-system itself is generally composed of many different levels. Every level in design uses features of lower levels to create new features for use by higher levels.



- File system provide efficient and convenient access to the disk by allowing data to be stored, located, and retrieved easily.

A file system poses two quite different design problems.

1. The first problem is defining how the file system should look to the user. This task involves defining a file and its attributes, the operations allowed on a file, and the directory structure for organizing files.
2. The second problem is creating algorithms and data structures to map the logical file system onto the physical secondary-storage devices.

The file system itself is generally composed of many different levels. The structure shown in Figure is an example of a layered design. Each level in the design uses the features of lower levels to create new features for use by higher levels.

- The lowest level, **the *I/O control***, consists of **device drivers** and interrupts handlers to transfer information between the main memory and the disk system.

A device driver can be thought of as a translator. Its input consists of high-level commands such as "retrieve block 123."

Its output consists of low level, hardware-specific instructions that are used by the hardware controller, which interfaces the I/0 device to the rest of the system.

- The device driver usually writes specific bit patterns to special locations in the I/0 controller's memory to tell the controller which device location to act on and what actions to take.
- The **basic file system** needs only to issue generic commands to the appropriate device driver to read and write physical blocks on the disk. Each physical block is identified by its numeric disk address (for example, drive 1, cylinder 73, track 2, sector10).

This layer also manages the memory buffers and caches that hold various file-system, directory, and data blocks.

A block in the buffer is allocated before the transfer of a disk block can occur. When the buffer is full, the buffer manager must find more buffer memory or free up buffer space to allow a requested I/O to complete.

File organization

- Module knows about files and their logical blocks, as well as physical blocks. By knowing the type of file allocation used and the location of the file, the file- organization module can translate logical block addresses to physical block addresses for the basic file system to transfer.
- Each file's logical blocks are numbered from 0 (or 1) through *N*. Since the physical blocks containing the data usually do not match the logical numbers, a translation is needed to locate each block.
- The file-organization module also includes the free-space manager, which tracks unallocated blocks and provides these blocks to the file-organization module when requested.

Logical file system

- Manages metadata information. Metadata includes all of the file- system structure except the actual *data* (or contents of the files). The logical file system manages the directory structure to provide the file organization module with the information the latter needs, given a symbolic file name. It maintains file structure via **file-control blocks (FCB)**.
- FCB contains information about the file, including ownership, permissions, and location of the file contents.

File System Implementation

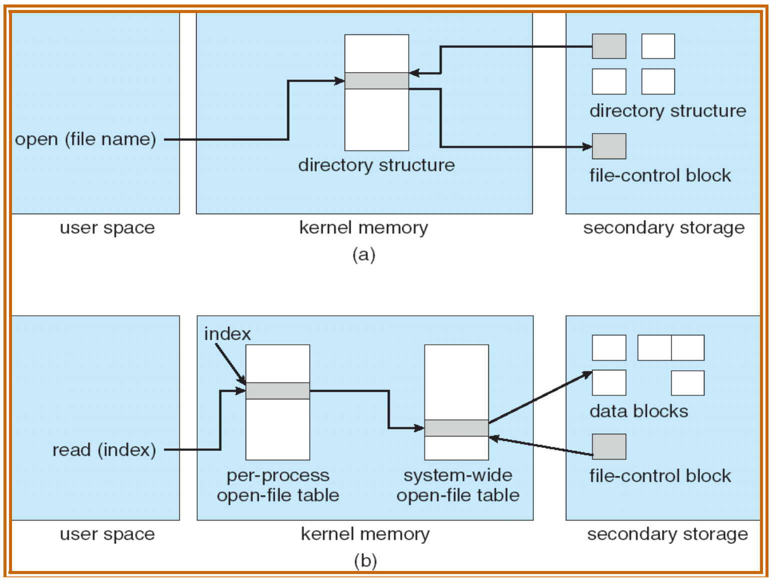
On disk, the file system may contain information about how to boot an operating system stored there, the total number of blocks, the number and location of free blocks, the directory structure, and individual files.

- **Boot Control Block**: On disk, the file system may contain information about how to boot an operating system stored there, the total number of blocks, the number and location of free blocks, the directory structure, and individual files.
- **Volume Control Block** : (per volume) contains volume (or partition) details, such as the number of blocks in the partition, the size of the blocks, a free-block count and free-block pointers and a free-FCB count and FCB pointers.
- **A directory structure** (per file system) is used to organize the files.
- **A per-file FCB** contains many details about the file. It has a unique identifier number to allow association with a directory entry.

The in-memory information is used for both file-system management and performance improvement via caching. The data are loaded at mount time, updated during file-system operations, and discarded at dismount. Several types of structures may be included.

- An in-memory mount table contains information about each mounted volume.
- An in-memory directory-structure cache holds the directory information of recently accessed directories.

- The system wide open file table contains a copy of the FCB of each open file, as well as other information.
- The per -process open file table contains a pointer to the appropriate entry in the system- wide open-file table, as well as other information.



- Buffers hold file-system blocks when they are being read from disk or written to disk.

Steps for creating a file:

1) An application program calls the logical file system, which knows the format of the directory structures
2) The logical file system allocates a new file control block(FCB)
    - If all FCBs are created at file-system creation time, an FCB is allocated from the free list
3) The logical file system then
    - Reads the appropriate directory into memory
    - Updates the directory with the new file name and FCB
    - Writes the directory back to the disk

UNIX treats a directory exactly the same as a file by means of a type field in the i node Windows NT implements separate system calls for files and directories and treats directories as entities separate from files.

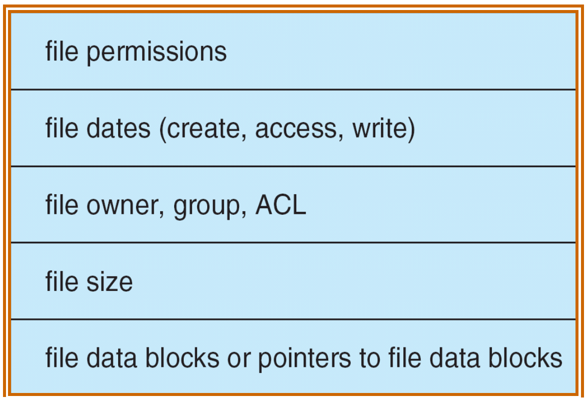
Steps for opening a file:

1) The function first searches the system-wide open-file table to see if the file is already in use by another process
    - If it is, a per-process open-file table entry is created pointing to the existing system-wide open-file table
    - This algorithm can have substantial overhead; consequently, parts of the directory structure are usually cached in memory to speed operations

2) Once the file is found, the FCB is copied into a system-wide open-file table in memory
    - This table also tracks the number of processes that have the file open
3) Next, an entry is made in the per-process open-file table, with a pointer to the entry in the system-wide open-file table
4) The function then returns a pointer/index to the appropriate entry in the per-process file-system table
    - All subsequent file operations are then performed via this pointer
    - UNIX refers to this pointer as the file descriptor
    - Windows refers to it as the file handle Steps for closing a file:

1) The per-process table entry is removed
2) The system-wide entry's open count is decremented
3) When all processes that have opened the file eventually close it

Any updated metadata is copied back to the disk-based directory structure. The system-wide open-file table entry is removed



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

Partitions and Mounting

- Each partition can be either "raw," containing no file system, or "cooked," containing a file system.
- Raw disk is used where no file system is appropriate.
- UNIX swap space can use a raw partition, for example, as it uses its own format on disk and does not use a file system.
- Boot information can be stored in a separate partition. Again, it has its own format, because at boot time the system does not have the file-system code loaded and therefore cannot interpret the file-system format.
- Boot information is a sequential series of blocks, loaded as an image into memory.
- Execution of the image starts at a predefined location, such as the first byte. This boot loader in turn knows about the file-system structure to be able to find and load the kernel and start it executing.
- The root partition which contains the operating-system kernel and sometimes other system files, is mounted at boot time.

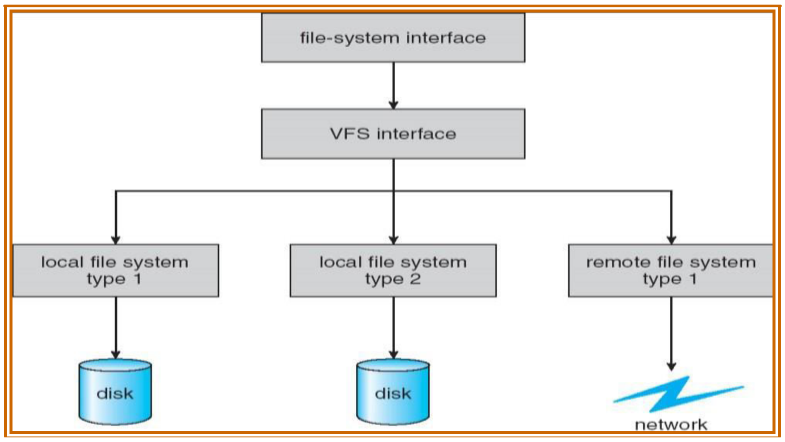
- Other volumes can be automatically mounted at boot or manually mounted later, depending on the operating system.
- After successful mount operation, the operating system verifies that the device contains a valid file system.
- It is done by asking the device driver to read the device directory and verifying that the directory has the expected format.
- If the format is invalid, the partition must have its consistency checked and possibly corrected, either with or without user intervention.
- Finally, the operating system notes in its in-memory mount table that a file system is mounted, along with the type of the file system.
- The root partition is mounted at boot time
    - It contains the operating-system kernel and possibly other system files
- Other volumes can be automatically mounted at boot time or manually mounted later
- As part of a successful mount operation, the operating system verifies that the storage device contains a valid file system
    - It asks the device driver to read the device directory and verify that the directory has the expected format
    - If the format is invalid, the partition must have its consistency checked and possibly corrected
    - Finally, the operating system notes in its in-memory mount table structure that a file system is mounted along with the type of the file system

Virtual file Systems

The file-system implementation consists of three major layers, as depicted schematically in Figure. The first layer is the file-system interface, based on the open(), read(), write(), and close() calls and on file descriptors.

The second layer is called the virtual file system (vfs) layer. The VFS layer serves two important functions:

- It separates file-system-generic operations from their implementation by defining a clean VFS interface. Several implementations for the VFS interface may coexist on the same machine, allowing transparent access to different types of file systems mounted locally.
- It provides a mechanism for uniquely representing a file throughout a network. The VFS is based on a file-representation structure, called a vnode that contains a numerical designator for a network-wide unique file. This network-wide uniqueness is required for support of network file systems.
- The kernel maintains one vnode structure for each active node.
- Thus, the VFS distinguishes local files from remote ones, and local files are further distinguished according to their file-system types.



Directory Implementation

Selection of directory allocation and directory management algorithms significantly affects the efficiency, performance, and reliability of the file system

One Approach: Direct indexing of a linear list

- Consists of a list of file names with pointers to the data blocks
- Simple to program
- Time-consuming to search because it is a linear search.
- Sorting the list allows for a binary search; however, this may complicate creating and deleting files
- To create a new file, we must first search the directory to be sure that no existing file has the same name.
- Add a new entry at the end of the directory.
- To delete a file, we search the directory for the named file and then release the space allocated to it.
- To reuse the directory entry, we can do one of several things. Mark the entry as unused.
- An alternative is to copy the last entry in the directory into the freed location and to decrease the length of the directory.
- Directory information is used frequently, and users will notice if access to it is slow.

Another Approach: List indexing via a hash function

- Takes a value computed from the file name and returns a pointer to the file name in the linear list
- Greatly reduces the directory search time
- **Can result in collisions** – situations where two file names hash to the same location
- A hash table are its generally fixed size and the dependence of the hash function on that size. (i.e., fixed number of entries).
- Each hash entry can be a linked list instead of an individual value, and we can resolve collisions by adding the new entry to the linked list.

ALLOCATION METHODS

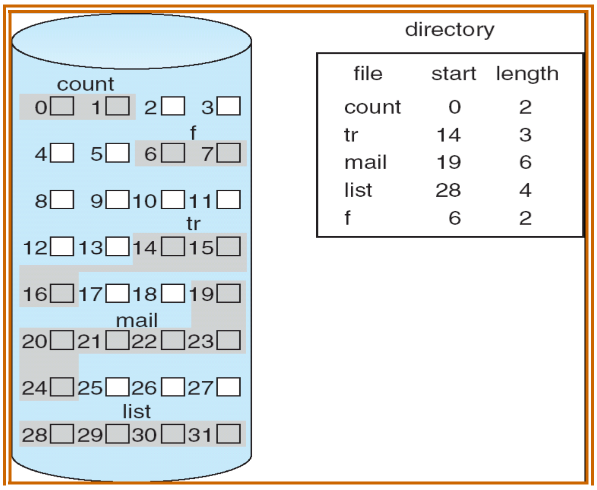
Allocation methods address the problem of allocating space to files so that disk space is utilized effectively and files can be accessed quickly.

Three methods exist for allocating disk space

- **Contiguous allocation**
- **Linked allocation**
- **Indexed allocation**

Contiguous allocation:

- Requires that each file occupy a set of contiguous blocks on the disk
- Accessing a file is easy – only need the starting location (block #) and length (number of blocks)
- Contiguous allocation of a file is defined by the disk address and length (in block units) of the first block. If the file is *n* blocks long and starts at location *b,* then it occupies blocks *b, b* + 1, *b* + 2, ... ,*b* + *n* - 1. The directory entry for each file indicates the address of the starting block and the length of the area allocated for this file.
- Accessing a file that has been allocated contiguously is easy. For sequential access, the file system remembers the disk address of the last block referenced and when necessary, reads the next block. For direct access to block *i* of a file that starts at block *b,* we can immediately access block *b* + *i.* Thus, both sequential and direct access can be supported by contiguous allocation.



directory

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

Disadvantages:

1. Finding space for a new file is difficult. The system chosen to manage free space determines how this task is accomplished. Any management system can be used, but some are slower than others.
2. Satisfying a request of size *n* from a list of free holes is a problem. First fit and best fit are the most common strategies used to select a free hole from the set of available

holes.

3. The above algorithms suffer from the problem of external fragmentation.
   - As files are allocated and deleted, the free disk space is broken into pieces.
   - External fragmentation exists whenever free space is broken into chunks.
   - It becomes a problem when the largest contiguous chunk is insufficient for a request; storage is fragmented into a number of holes, none of which is large enough to store the data.
   - Depending on the total amount of disk storage and the average file size, external fragmentation may be a minor or a major problem.

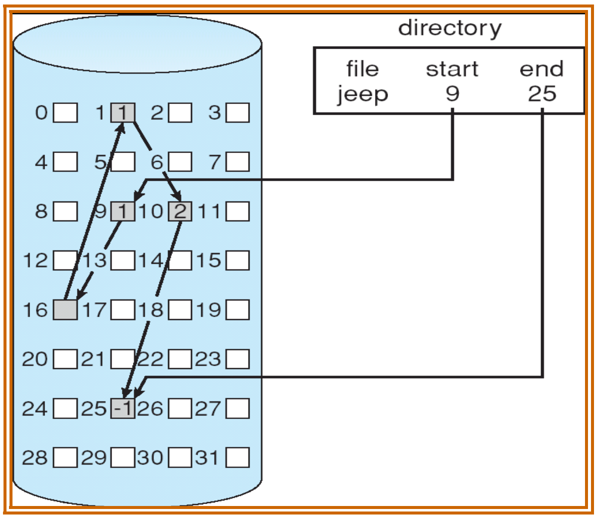
Linked Allocation:

- Solves the problems of contiguous allocation
- Each file is a linked list of disk blocks: blocks may be scattered anywhere on the disk
- The directory contains a pointer to the first and last blocks of a file
- Creating a new file requires only creation of a new entry in the directory
- Writing to a file causes the free-space management system to find a free block

- ➢ This new block is written to and is linked to the end of the file
- ➢ Reading from a file requires only reading blocks by following the pointers from block to block.

Advantages

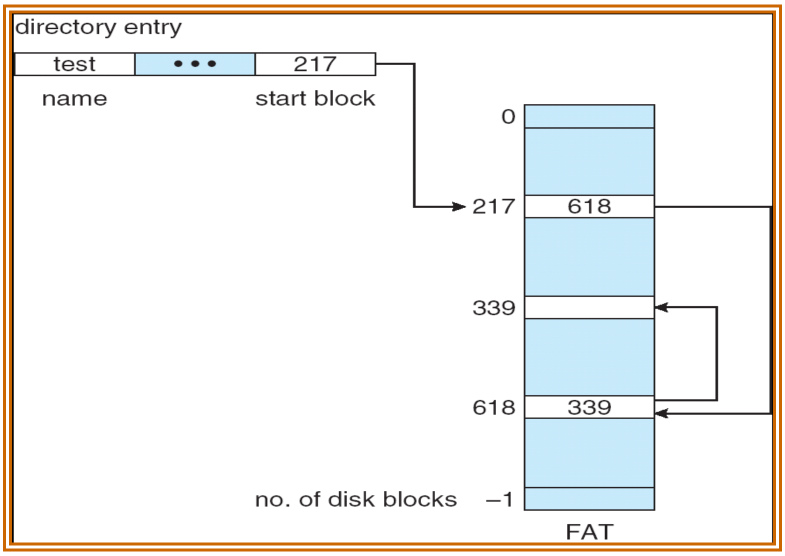
- There is no external fragmentation
- Any free blocks on the free list can be used to satisfy a request for disk space
- The size of a file need not be declared when the file is created
- A file can continue to grow as long as free blocks are available
- It is never necessary to compact disk space for the sake of linked allocation (however, file access efficiency may require it)



- Each file is a linked list of disk blocks; the disk blocks may be scattered anywhere on the disk. The directory contains a pointer to the first and last blocks of the file.
- For example, a file of five blocks might start at block 9 and continue at block 16, then block 1, then block 10, and finally block 25. Each block contains a pointer to the next block. These pointers are not made available to the user. A disk address (the pointer) requires 4 bytes in the disk.
- To **create** a new file, we simply create a new entry ile the directory. With linked allocation, each directory entry has a pointer to the first disk block of the file. This pointer is initialized to *nil* (the end-of-list pointer value) to signify an empty file. The size field is also set to 0.
- A **write** to the file causes the free-space management system to filed a free block, and this new block is written to and is linked to the end of the file.
- To **read** a file, we simply read blocks by following the pointers from block to block. There is no external fragmentation with linked allocation, and any free block on the free-space list can be used to satisfy a request. The size of a file need not be declared when that file is created.
- A file can continue to grow as long as free blocks are available. Consequently, it is never necessary to compact disk space.
- **Disadvantages**:
    1. The major problem is that it can be used effectively only for sequential-access files. To filed the i th block of a file, we must start at the beginning of that file and follow the pointers until we get to the ith block.
    2. Space required for the pointers. Solution is clusters. Collect blocks into multiples and allocate clusters rather than blocks
    3. Reliability - the files are linked together by pointers scattered all over the disk and if a pointer were lost or damaged then all the links are lost.

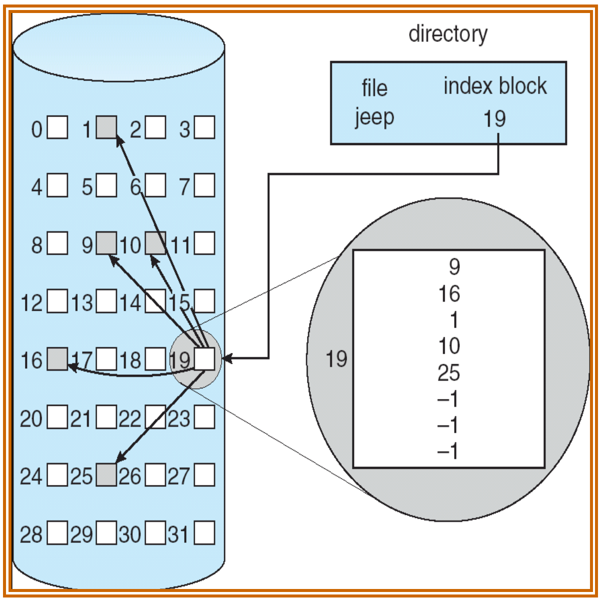
File Allocation Table:

- A section of disk at the beginning of each volume is set aside to contain the table. The table has one entry for each disk block and is indexed by block number.
- The FAT is used in much the same way as a linked list. The directory entry contains the block number of the first block of the file.
- The table entry indexed by that block number contains the block number of the next block in the file.
- The chain continues until it reaches the last block, which has a special end-of-file value as the table entry.
- An unused block is indicated by a table value of 0.
- Consider a FAT with a file consisting of disk blocks 217, 618, and 339.



Indexed allocation:

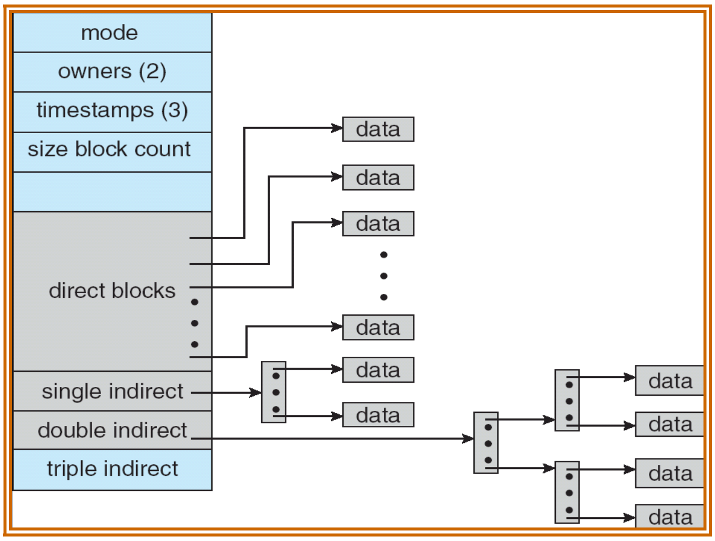
- Brings all the pointers together into one location called index block.
- Each file has its own index block, which is an array of disk-block addresses.
- The *ith* entry in the index block points to the *ith* block of the file. The directory contains the address of the index block. To find and read the *ith* block, we use the pointer in the *ith* index- block entry.
- When the file is created, all pointers in the index block are set to *nil*. When the ith block is first written, a block is obtained from the free-space manager and its address is put in the ith index- block entry.
- Indexed allocation supports direct access, without suffering from external fragmentation, because any free block on the disk can satisfy a request for more space.
- Disadvantages :
  - Suffers from some of the same performance problems as linked allocation
  - Index blocks can be cached in memory; however, data blocks may be spread all over the disk volume.
  - Indexed allocation does suffer from wasted space.
  - The pointer overhead of the index block is generally greater than the pointer overhead of linked allocation.



If the index block is too small, however, it will not be able to hold enough pointers for a large file, and a mechanism will have to be available to deal with this issue. Mechanisms for this purpose include the following:

a) **Linked scheme.** An index block is normally one disk block. Thus, it can be read and written directly by itself. To allow for large files, we can link together several index blocks. For example, an index block might contain a small header giving the name of the file and a set of the first 100 disk-block addresses. The next address (the last word in the index block) is *nil* (for a small file) or is a pointer to another index block (for a large file).

b) **Multilevel index.** A variant of linked representation uses a first-level index block to point to a set of second-level index blocks, which in turn point to the file blocks. To access a block, the operating system uses the first-level index to find a second-level index block and then uses that block to find the desired data block. This approach could be continued to a third or fourth level, depending on the desired maximum file size



c) **Combined scheme.** For eg. 15 pointers of the index block is maintained in the file's i node. The first 12 of these pointers point to **direct blocks**; that is, they contain addresses of blocks that contain data of the file. Thus, the data for small files (of no more than 12 blocks) do not need a separate index block. If the block size is 4 KB, then up to 48 KB of data can be accessed directly. The next three pointers point to indirect blocks. The first points to a **single indirect block**, which is an index block containing not data but the addresses of blocks that do contain data. The second points to a **double indirect block**, which contains the address of a block that contains the addresses of blocks that contain pointers to the actual data blocks. The last pointer contains the address of a **triple indirect block**.

Performance

- Contiguous allocation requires only one access to get a disk block. Since we can easily keep the initial address of the file in memory, we can calculate immediately the disk address of the *ith* block and read it directly.
- For linked allocation, we can also keep the address of the next block in memory and read it directly. This method is fine for sequential access. Linked allocation should not be used for an application requiring direct access.
- Indexed allocation is more complex. If the index block is already in memory, then the access can be made directly. However, keeping the index block in memory requires considerable space. If this memory space is not available, then we may have to read first the index block and then the desired data block.

Free Space Management

The space created after deleting the files can be reused. Another important aspect of disk management is keeping track of free space in memory. The list which keeps track of free space in memory is called the free-space list. To create a file, search the free-space list for the required amount of space and allocate that space to the new file. This space is then removed from the free-space list. When a file is deleted, its disk space is added to the free-space list. The free-space list, is implemented in different ways as explained below.

a) Bit Vector

- Fast algorithms exist for quickly finding contiguous blocks of a given size
- 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.

For example, consider a disk where blocks 2,3,4,5,8,9, 10,11, 12, 13, 17and 18 are free, and the rest of the blocks are allocated. The free-space bit map would be
0011110011111100011

- Easy to implement and also very efficient in finding the first free block or ‘n’ consecutive free blocks on the disk.

- The down side is that a 40GB disk requires over 5MB just to store the bitmap.

b) Linked List

a. A linked list can also be used to keep track of all free blocks.
b. 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.
c. The FAT table keeps track of the free list as just one more linked list on the table.

c) Grouping

a. A variation on linked list free lists. It stores the addresses of n free blocks in the first free block. The first n-1 blocks are actually free. The last block contains the addresses of another n free blocks, and so on.
b. The address of a large number of free blocks can be found quickly.

d) Counting

a. 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.
b. Rather than keeping al list of n free disk addresses, we can keep the address of first free block and the number of free contiguous blocks that follow the first block.
c. Thus the overall space is shortened. It is similar to the extent method of allocating blocks.

e) Space Maps

a. Sun's ZFS file system was designed for huge numbers and sizes of files, directories, and even file systems.
b. The resulting data structures could be 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.
c. ZFS uses a combination of techniques, starting with dividing the disk up into (hundreds of) ***Meta slabs*** of a manageable size, each having their own space map.
d. 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.
e. An in-memory space map is constructed using a balanced tree data structure, constructed from the log data.
f. 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.

QUESTION BANK

1. What is a file? Distinguish between contiguous and linked allocation methods with the neat diagram.
2. Explain file allocation methods by taking an example with the neat diagram. Write the advantages and disadvantages.
3. Explain free space management. Explain typical file control block, with a neat sketch.
4. Distinguish between single level directory structure and two level directory structures. What are its advantages and disadvantages?
5. Explain the access matrix model of implementing protection in operating system.
6. For the following page reference string **1,2,3,4,1,2,5,1,2,3,4,5**. Calculate the page faults using FIFO, Optimal and LRU using 3 and 4 frames.
7. Explain Demand paging in detail.
8. For the following page reference string **7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1**. Calculate the page faults using FIFO, Optimal and LRU using 3 and 4 frames.
9. Explain copy-on-write process in virtual memory.
10. What is a page fault? with the supporting diagram explain the steps involved in handling page fault.
11. Illustrate how paging affects the system performance.
12. Explain the various types of directory structures.
13. Explain the various file attributes.
14. Explain the various file operations.
15. Explain the various mechanism of implementing file protection.