Module 4
File System and Implementation of File System

- File concept;
- Access methods;
- Directory structure;
- File system mounting;
- File sharing;
- Protection:
- **Implementing File system**: File system structure;
- File system implementation;
- Directory implementation;
- Allocation methods;
- Free space management

4.1 Introduction:

- A file is a named collection of related information that is recorded on secondary storage such as magnetic disks, magnetic tapes and optical disks. In general, a file is a sequence of bits, bytes, lines or records whose meaning is defined by the files creator and user.
- The file system is the most visible aspect of an operating system. It provides the **mechanism for on-line storage** of and access to both data and programs of the operating system and all the users of the computer system.
- The file system consists of **two distinct parts,** a collection of **files** each storing related data, and a **directory structure** which organizes and provides information about all the files in the system.

4.2 File Concept

- Computers can store information on various storage media.
- Operating system provides a uniform logical view of information storage. This **logical storage unit** is called as a **file**.
- Files are **mapped by operating system** onto physical devices. These storage devices are **nonvolatile**, so contents are **persistent** through power failures and system reboots.
- **File** is a named collection of **related information** that is recorded on secondary storage.
- Files represent both the **program and the data**. Data can be numeric, alphanumeric, alphabetic or binary.
- Many different types of information like source programs, object programs, executable programs, numeric data, payroll recorder, graphic images, and sound recordings and so on can be stored on a file.
- A file has a **certain defined structures** according to its type.
    - **Text file:** Text file is a sequence of characters organized in to lines.
    - **Object file:** Object file is a sequence of bytes organized in to blocks understandable by the systems linker.

- **Executable file:** Executable file is a series of code section that the loader can bring in to memory and execute.

- **Source File:** Source file is a sequence of subroutine and function, each of which are further organized as declaration followed by executable statements.

4.2.1 File Attributes :

File attributes varies from one OS to other. The common file attributes are,

- **Name:** The symbolic file name is the only information kept in human readable form

- **Identifier:** The unique tag, usually a number, identifies the file within the file system. It is the non-readable name for a file.

- **Type:** This information is needed for systems that support different types.

- **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 and possibly the maximum allowed size are included in this attribute.

- **Protection:** Access control information determines who can do reading, writing, execute and so on.

- **Time, data and User Identification:** This information must be kept for creation, last modification and last use. These data are useful for protection, security and usage monitoring.

4.2.2 File Operations

File is an abstract data type. To define a file we need to consider the operation that can be performed on the file. Basic operations of files are:

- **Creating a file: Two steps** are necessary to create a file. First **space** in the file system for file is found. Second an **entry** for the new file must be made in the directory.

- **Writing a file: System call** is mainly used for writing in to the file. System call specifies the **name** of the file and the **information** to be written to the file. Given the name the system search the entire directory for the file. The system must keep a write pointer to the location in the file where the next write to be taken place.

- **Reading a file:** To read a file, system call is used. It requires the **name** of the file and the **memory address** where the next block of the file should be put. Again the directory is searched for the associated directory and system must maintain a read pointer to the location in the file where next read is to take place.

- **Delete a file:** System will search for the directory for which file to be deleted. If entry is found it releases all free space. That free space can be reused by another file.

- **Truncating the file:** User may want to erase the contents of the file but keep its attributes. Rather than forcing the user to delete a file and then recreate it, truncation allows all attributes to remain unchanged except for file length.

- **Repositioning within a file:** The directory is searched for appropriate entry and the current file position is set to a given value. Repositioning within a file does not need to involve actual I/O. The file operation is also known as **seek.**

In addition to this basis 6 operations the **other two operations** include **appending** new information to the end of the file and **renaming** the existing file. Most of the file operation involves searching the entire directory for the entry associated with the file. To avoid this OS 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 index into this table. So searching is not required. Several **piece of information** are associated with an **open file are:**

- **File pointer:** on systems that does not include offset as part of the read and write system calls, the system must track the last read-write location as current file position pointer. This pointer is unique to each process operating on a file.

- **File open count:** As the files are closed, the OS must reuse its open file table entries, or it could run out of space in the table. Because multiple processes may open a file, the system must wait for the last file to close before removing the open file table entry. The counter tracks the number of copies of open and closes and reaches zero to last close.

- **Disk location of the file:** The information needed to locate the file on the disk is kept in memory to avoid having to read it from the disk for each operation.

- **Access rights:** Each process opens a file in an access mode. This information is stored on per-process table the OS can allow or deny subsequent i/o request.

Some operating systems provide facilities for **locking an open file (or sections of a file)**. File locks allow one process to lock a file and prevent other processes from gaining access to it. File locks are useful for files that are shared by several processes. A **shared lock** is similar to a **reader lock** in that several processes can acquire the lock concurrently. An **exclusive lock** behaves like a **writer lock**; only one process at a time can acquire such a lock. Operating systems provide both types of locks; some systems provide only exclusive file locking.

Operating systems may also provide either **mandatory or advisory** file-locking mechanisms. If a lock is mandatory, then once a process acquires an exclusive lock, the operating system will prevent any other process from accessing the locked file. For advisory locking, it is up to software developers to ensure that locks are appropriately acquired and released. Windows operating systems adopt mandatory locking, and UNIX systems employ advisory locks.

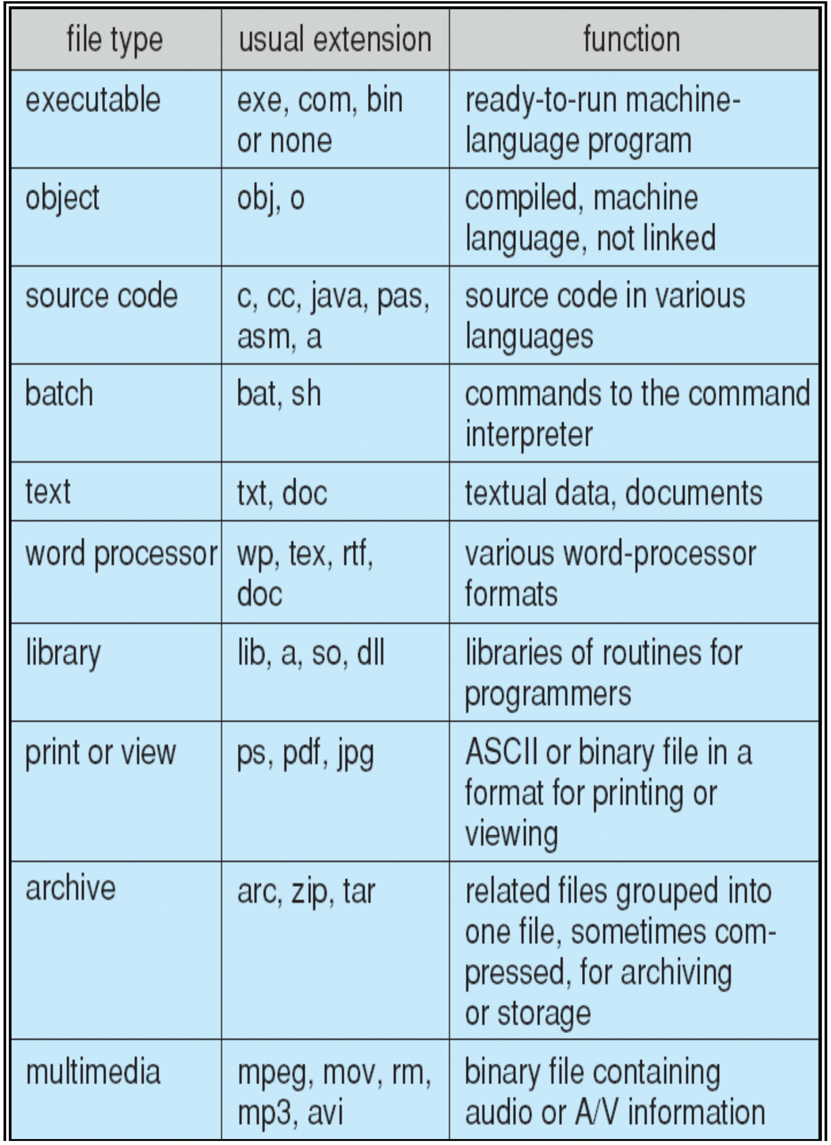
4.2.3 File Types

- File type is included as a part of the filename. File name is split into **two parts-** a **name** and **extension se**parated 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.

- **For example,** only a file with a .com, .exe, or .bat extension can be executed. The **.com** and **.exe** files are two forms of binary executable files, whereas **.bat** file is a **batch file** containing commands in ASCII format to the operating system.

- The table shown below gives the file type with extension and function.



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

4.2.4 File Structure

- File types can be used to indicate the internal structure of the file. Certain files must match 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.
- Some operating systems extend this idea into a **set of system-supported file structures,** with sets of special operations for manipulating files with those structures. This is one disadvantage where the resulting size of the operating system is cumbersome.
- Some operating systems impose and support a **minimal number** of file structures.

- Too few structures make programming inconvenient, whereas too many cause operating-system to expand and makes programmer to confuse.

4.2.5 Internal File Structure

- Locating an **offset** within a file can be complicated for the operating system.
- Disk systems have a well-defined block size determined by the size of a sector. All disk I/0 is performed in units of one block and all blocks are the same size. It is not sure 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. The logical record size, physical block size, and packing technique determine how many logical records are in each physical block. The packing can be done either by the user's application program or by the operating system. In either case, the file may be considered a sequence of blocks.
- All the basic I/O functions operate in terms of blocks. The conversion from logical records to physical blocks is a relatively simple software problem.
- Because disk space is always allocated in blocks, some portion of the last block of each file is wasted. The waste incurred to keep everything in units of blocks is **internal fragmentation.**
- All file systems suffer from internal fragmentation; the larger the block size, the greater the internal fragmentation.

4.3 Access Methods: The information in the file can be accessed in several ways. The different file access methods are,

4.3.1 Sequential Access:

- ✓ Sequential access is the **simplest** access method. Information in the file is processed in order, one record after another. **Editors and compilers** access the files in this fashion.
- ✓ A read operation **read next** reads the next portion of the file and automatically advances a file pointer, which tracks next 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.
- ✓ Such 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.
- ✓ Sequential access, which is depicted in **figure**, is based on a **tape model** of a file and works well on sequential-access devices as it does on random-access ones.

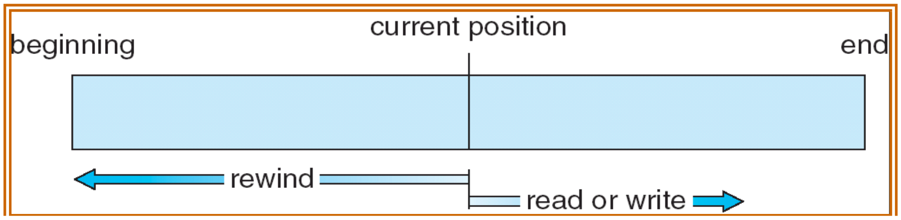
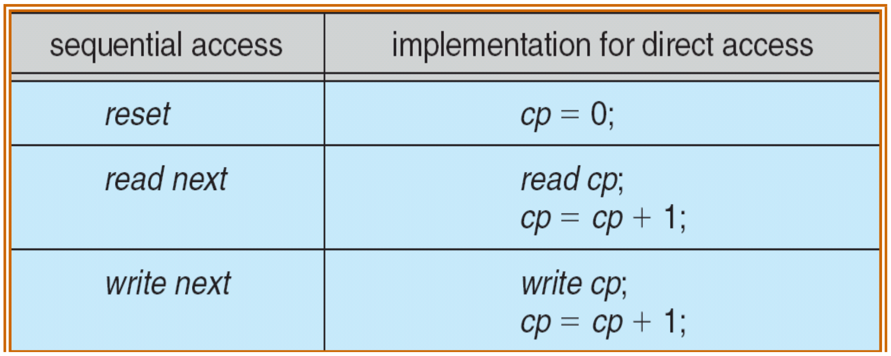


Figure: Sequential access file

4.3.2 Direct Access (Relative Access)

- ✓ A file is made up of fixed length logical records. It allows the program to read and write records rapidly in any order.
- ✓ Direct access allows **random access** to any file block. This method is based on **disk model** of a file.
- ✓ For direct access, the file is viewed as a **numbered sequence of blocks or records.** Thus, we may read block 14, then read block 53, and then write block 7.
- ✓ The direct access method is suitable for searching the records in large amount of information. **For example,** on an **airline-reservation system,** we might store all the information about a particular flight (for example, flight 713) in the block identified by the flight number. Thus, the number of available seats for flight 713 is stored in block 713 of the reservation file.
- ✓ 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.
- ✓ The block number provided by the user to the operating system is normally a **relative block number.** A relative block number is an index relative to the beginning of the file.
- ✓ We can easily **simulate** sequential access on a direct-access file by simply keeping a **variable cp** that defines **current position**, as shown in below **figure**. Simulating a direct-access file on a sequential-access file is extremely inefficient and clumsy.



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 over direct access

4.3.3 Other Access Methods

- ✓ These methods generally involve the **construction of an index** for the file.
- ✓ The index is like an index at the end of a book which contains pointers to various blocks.
- ✓ To find a record in a file, we search the index and then use the pointer to access the file directly and to find the desired record.
- ✓ With large files index file itself can be very large to be kept in memory. One **solution** is to create an index for the index files itself. The primary index file would contain pointer to secondary index files which would point to the actual data items.
- ✓ **For example**, IBM's **indexed sequential-access method (ISAM)** uses a small master index that points to disk blocks of a secondary index. The secondary index blocks point to the actual file blocks. The file is kept sorted on a **defined key**. To find a particular item, we first make a binary search of the master index, which provides the block number of the secondary index. This block is read in, and again a binary search is used to find the block containing the desired record. Finally, this block is searched sequentially. In this way, any record can be located from its key by at most two direct-access reads. The below **figure** shows a similar situation as implemented by VMS index and relative files.

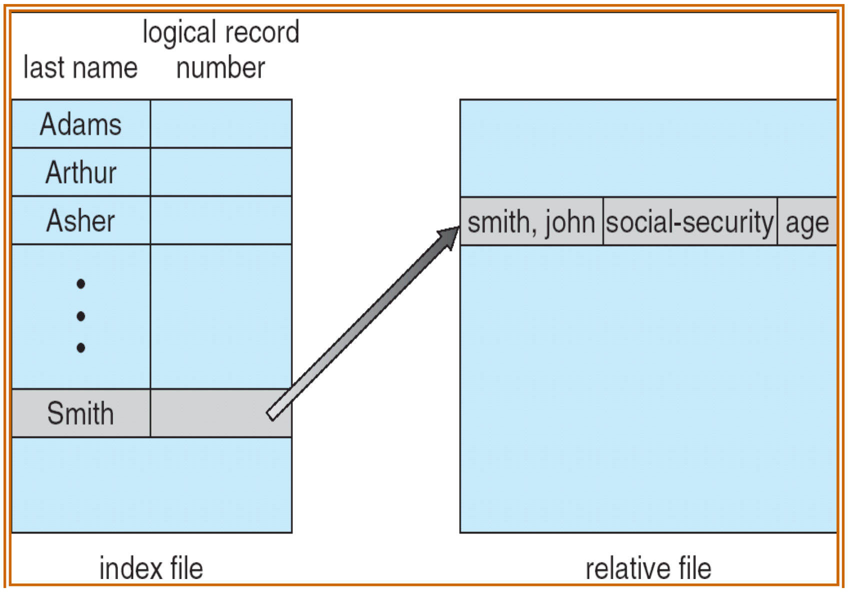


Fig: Example of index and relative files

4.4 Directory Structure

The files systems can be very large. Some systems stores millions of files on the disk. To manage all this data we need to organize them. This organization involves the use of **directories.**

4.4.1 Storage structure

- A disk can be used in its entirety for a file system. Sometimes it is desirable to place multiple file systems on a disk or to use parts of the disk for a file system. These parts are known as **partitions, slices or minidisks**.
- These parts can be combined to form larger structures known as **volumes.**
- Each volume contains information about files within it. This information is kept in entries in a **device directory (directory) or volume table of contents.**
- The **directory** records the information such as name, location, size, type for all files on that volume.
- The below **figure** shows a typical file-system organization.

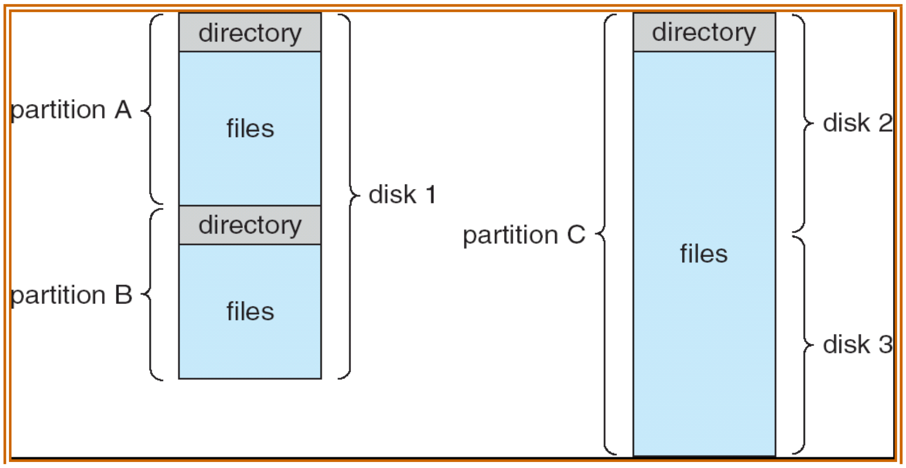


Figure: Typical file system organisation

4.4.2 Directory Overview

- The directory can be viewed as a **symbol table** that translates the file names into their directory entries. The directory itself can be organized in many ways.
- When considering a particular directory structure, several operations can be performed on a directory.
  - **Search for a file:** Directory structure is searched for finding particular file in the directory. Files have symbolic names and similar names may indicate a relationship between files, we must be able to find all the files whose name matches a particular pattern.
  - **Create a file:** New files can be created and added to the directory.
  - **Delete a file:** when a file is no longer needed, we can remove it from the directory.
  - **List a directory:** We need to be able to list the files in directory and the contents of the directory entry for each file in the list.
  - **Rename a file:** Name of the file must be changeable when the contents of the file are changed. Renaming allows the position within the directory structure to be changed.
  - **Traverse the file:** It is always good to keep the backup copy of the file so that it can be used when the system fails or when the file system is not in use.
- There are **different** types of **logical structures of a directory** as discussed below.

4.4.3 Single-level directory

- This is the simplest directory structure. All the files are contained in the same directory which is easy to support and understand as shown in below **figure.**

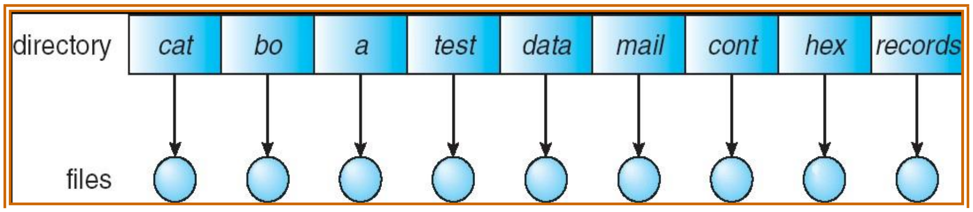


Figure: single level directory

- The disadvantages are,
  - Not suitable for a large number of files and more than one user.
  - Because of single directory files, files require unique file names.
  - Difficult to remember names of all the files as the number of files increases.

4.4.4 Two-level directory

- A single level directory often leads to the confusion of file names between different users. The solution here is to create **separate directory for each user**.
- In two level directories each user has its own directory. It is called **User File Directory (UFD)**. Each UFD has a similar structure, but lists only the files of a single user.
- When a user job starts or users logs in, the system's **Master File Directory (MFD)** is searched. The MFD is indexed by the user name or account number and each entry points to the UFD for that user as shown in below **figure.**

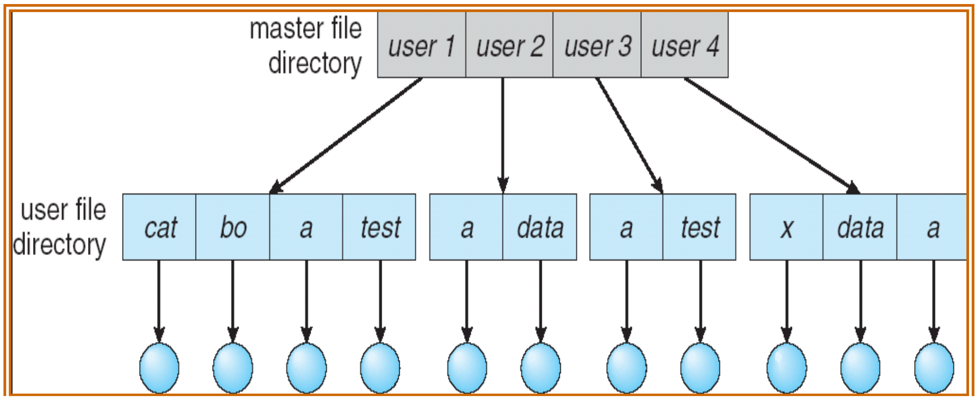
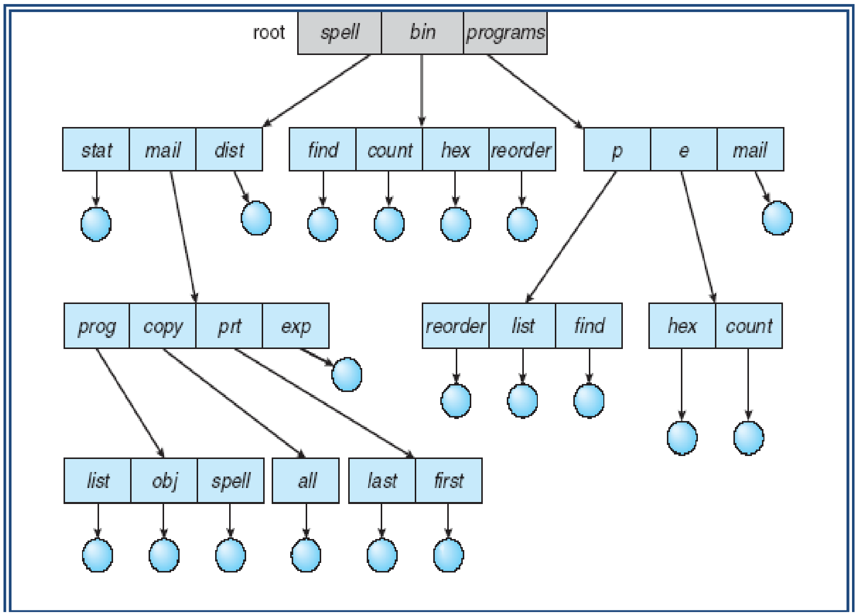


Figure: Two level directory

- When a user refers to a particular file, only his own UFD is searched. Thus different users may have files with the same name.
- To create a file for a user, OS searches only that user UFD to check whether another file of that name exists.
- To delete a file OS checks in the local UFD, so that it cannot accidentally delete another user’s file with the same name.
- Although two-level directories **solve the name collision problem** but it still has some **disadvantages**.
- This structure isolates one user from another and this isolation is an advantage when the users are independent, but disadvantage when some users want to co-operate on some task and to access one another’s file.
- If access is to be permitted, one user must have the ability to name a file in another user's directory. To name a particular file we must give both the user name and the file name.
- A two-level directory is like a **tree, or an inverted tree of height 2.**
  The root of the tree is the MFD. Its direct descendants are the UFDs. The descendants of the UFDs are the files. The **files** are the leaves of the **tree**. A user name and a file name define a **path name.**
- To access the **system files** the appropriate commands are given to the operating system and these files are read by the loader and executed. This file name would be searched in the current UFD. The sequence of directories searched when a file is named is called the **search path.**

4.4.5 Tree-structured directories

- The two level directory structures can be extended to a tree of **arbitrary height** as shown in **figure below.**

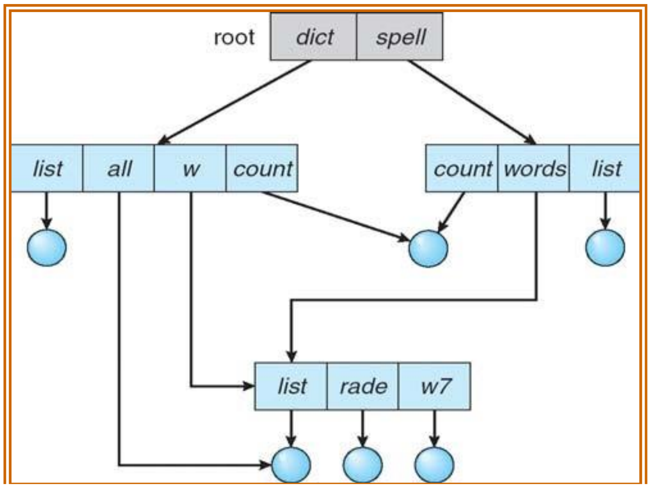


- Each process has a **current directory** and it should contain most of the files that are of the current interest to the process. When a reference is made to a file the current directory is searched. If needed file is not in the current directory the user must specify the path name or change the current directory.

- Path name can be of two types.
  - **Absolute path name:** Begins at the root and follows a path down to the specified file, giving the directory names on the path.
  - **Relative path name:** Defines a path from the current directory.
- One important task is how to handle the deletion of a directory. If a directory is empty, its entry can simply be deleted. If a **directory is not empty, one of the two approaches** can be used.
  - In MS-DOS, the directory is not deleted until it becomes empty.
  - In UNIX, **rm** command is used, where all directory's files and subdirectories are also deleted before deleting a directory.
- With a tree-structured directory system, users can be allowed to access their files, and also the **files of other users**. Alternatively **user can change the current directory** to other user's directory and access the file by its file names.
- A **path** to a file in a tree-structured directory can be **longer** than a path in a two-level directory.

4.4.6 Acyclic graph directories

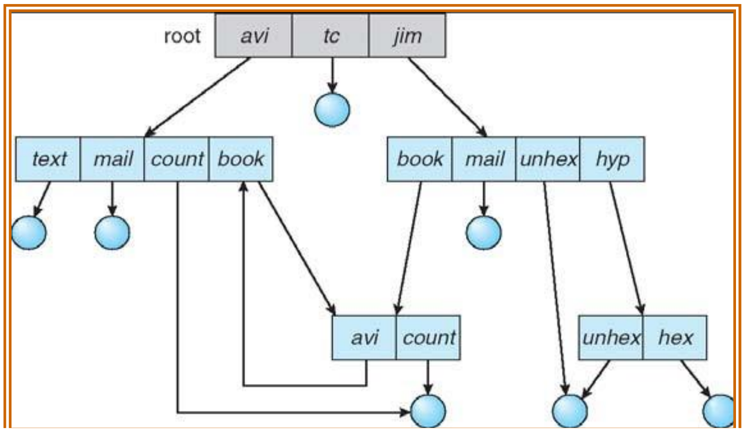
- A tree structure prohibits the sharing of files or directories.
- An acyclic graph that is, a **graph with no cycles** allows directories to **share subdirectories and files** as shown in **figure.** The same file or subdirectory may be in two different directories.



- The acyclic graph is a natural generalization of the tree-structured directory scheme. With a shared file, only one actual file exists, so any changes made by one person are immediately visible to the other.
- **Sharing** is mainly **important for subdirectories**; a new file created by one person will automatically appear in all the shared subdirectories.
- Shared files and subdirectories can be **implemented** by using **links.** A link is a **pointer** to another file or a subdirectory. **Another approach** to implementing shared files is simply to **duplicate all information** about them in both sharing directories. Thus, both entries are identical and equal.
- An acyclic graph directory structure is **more flexible** than a simple tree structure but sometimes it is **more complex.**
- Some of the **problems** are, a file may now have **multiple absolute path names** and distinct file names may refer to the same file.
- **Another problem** involves in **deletion**. There are **two approaches** to decide when the space allocated to a shared file can be deallocated and reused.
- **One possibility is to remove the file** whenever anyone deletes it, but this action may leave dangling pointers to the nonexistent file.
- **Another approach is to preserve the file** until all references to it are deleted. To **implement this approach**, we could keep a list of all references to a file (directory entries or symbolic links). When a link or a copy of the directory entry is established, a new entry is added to the **file-reference list.** When a link or directory entry is deleted, we remove its entry on the list. The file is deleted when its **file-reference list is empty**. The **trouble with this approach** is the variable and potentially large size of the file-reference list. So we need to keep only the **count of number of references,** and when the **count is 0,** the file can be deleted.

4.4.7 General Graph Directory

- The **problem with using an acyclic-graph structure** is ensuring that there are no cycles. When we add links, the tree structure is destroyed, resulting in a simple graph structure as shown in below **figure 10.10.**
- The **advantage of an acyclic graph** is the **simplicity** of the algorithms to traverse the graph and to determine when there are **no more references** to a file.



- A poorly designed algorithm might result in an infinite loop continually searching through the cycle and never terminating. One **solution** is to limit the number of directories that will be accessed during a search.
- A **problem exists** when we are trying to determine when a file can be **deleted.** When cycles exist, the reference count may not be 0 even when it is no longer possible to refer to a directory or file. So a **garbage-collection scheme** is used.
- Garbage collection involves **traversing** the entire file system, **marking everything that can be accessed** but it is **time consuming**. Thus, an **acyclic-graph structure is much easier** to work than general graph structure. The **difficulty** is to **avoid cycles** as new links are added to the structure. There are algorithms to detect cycles in graphs but they are **computationally expensive**, hence **cycles must be avoided.**

4.5 File System Mounting

- The file system must be mounted before it can be available to processes on the system
- The **procedure** for mounting the file is,
  - The OS is given the name of the device and the location within the file structure at which to attach the file system (**mount point**).A mount point will be an **empty directory** at which the mounted file system will be attached. **For example:** On UNIX, a file system containing **user's home directory** might be mounted as **/home** then to access the directory structure within that file system we must precede the directory names as **/home/Jane.**
  - Then OS verifies that the device contains this valid file system. OS uses device drivers for this verification.
  - Finally the OS mounts the file system at the specified mount point.
- consider the file system depicted in **figure 10.11**, where the triangles represent subtrees of directories. **Figure 10.11(a)** shows an existing file system, while **figure 10.11(b)** shows an unmounted volume residing on **/device/dsk**. At this point, only the files on the existing file system can be accessed. **Figure 10.12** shows the effects of mounting the volume residing on **/device/dsk** over **/users.**

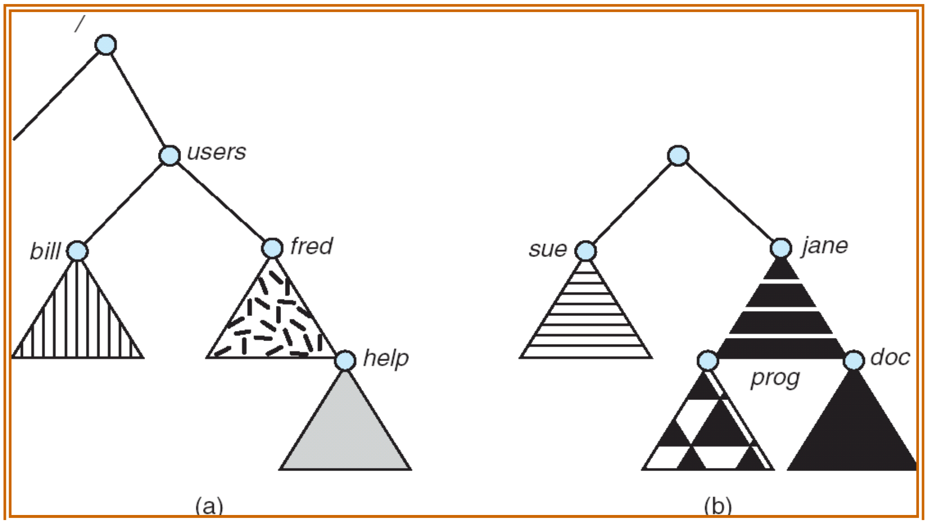


Figure: a)Existing b) Unmounted volume

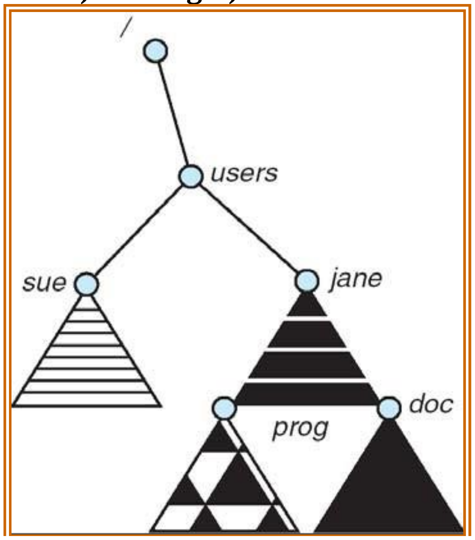


Figure: Mounted point

- Windows operating systems automatically discover all devices and mount all located file systems at boot time. In UNIX systems, the mount commands are explicit.
- A system configuration file contains a list of devices and mount points for automatic mounting at boot time, but other mounts may be executed manually.

4.6 File Sharing

4.6.1 Multiple Users

- ✓ Given a directory structure that allows files to be shared by users, the operating system must mediate the file sharing.
- ✓ The system either can allow a user to access the files of other users by default or it may require that a user specifically grant access to the files.
- ✓ To implement sharing and protection, the system maintain more file and directory attributes than on a single user system, most systems support the concept of file **owner and group.**

- ✓ When a user requests an operation on a file, the user ID can be compared to the owner attribute to determine if the requesting user is the owner of the file. Likewise the group ID's can be compared. The result indicates which permissions are applicable.

4.6.2 Remote File Systems

- ✓ Remote sharing of the file system is implemented by using **network.**
- ✓ **Network** allows the sharing of resources. **File Transfer Protocol (FTP)** is one of the methods used for remote sharing. Other methods are **distributed file system (DFS) and World Wide Web**.
- ✓ DFS involves a much tighter integration between the machine that is accessing the remote files and the machine providing the files. This integration adds complexity as discussed in below DFSs.
  - ▪ **Client-server Model**
    - ✓ System containing the files is the server and the system requesting access to the files is a client. Files are specified on a partition or subdirectory level. A server can serve multiple clients and a client can use multiple servers.
    - ✓ A client can be specified network name or other identifier, such as an IP address, but these can be but these can be **spoofed** or imitated. As a result of spoofing, an unauthorized client could be allowed to access the server. More secure solutions include secure authentication of the client by using **encrypted keys**.
  - ▪ **Distributed Information systems**
    - ✓ For managing client server services, distributed information system is used to provide a unified access to the information needed for remote computing. **UNIX** systems have a wide variety of distributed information methods. The **domain name system (DNS) provides host-name-to-network-address translations** for the entire internet.
    - ✓ Before DNS became widespread, files containing the same information were sent via e-mail or ftp between all networked hosts. This methodology was not scalable.
    - ✓ Sun Microsystems introduced **yellow pages** which are called as **Network Information Service (NIS).** It centralizes storage of user names, host names, printer information, and others. But it uses unsecure authentication methods, like sending user passwords unencrypted (in clear text) and identifying hosts by IP address.
    - ✓ **NIS+** is a much more secure replacement for NIS but is also much more complicated and has not been widely adopted.
    - ✓ Microsoft introduced **Common Internet File System (CIFS).** Here network information is used in conjunction with user authentication (user name and password) to create a **network login** that the server uses to decide whether to allow or deny access to a requested file system. For this authentication to be valid, the user names must match between the machines.
    - ✓ Microsoft uses **two distributed naming structures** to provide a single name space for users. The older naming technology is **domains.** The newer technology, available in Windows XP and Windows 2000, is **active directory.**
    - ✓ **Light Weight Directory Access Protocol (LDAP)** introduced by Sun Microsystems is a secure **distributed naming mechanism.** It is a **secure single sign-on** for users, who would enter their authentication information once for access to all computers within the organization. This **reduces system-administration efforts** by combining the information

that is currently scattered in various files on each system or in different distributed information services.

- **Failure Modes**

  - ✓ **Local file systems can fail** for a variety of reasons, including failure of the
  - ✓ disk containing the file system, corruption of the directory structure or other **disk-management information (metadata)** disk-controller failure, cable failure, and host-adapter failure.
  - ✓ User or system-administrator failure can also cause files to be lost or entire directories or volumes to be deleted. Many of these failures will cause a host to crash and an error condition to be displayed, and human intervention will be required to repair the damage.
  - ✓ **Remote file systems have even more failure modes.** Because of the complexity of network systems and the required interactions between remote
  - ✓ machines, many more problems can interfere with the proper operation of
  - ✓ remote file systems.
  - ✓ In the case of networks, the network can be interrupted between two hosts. Such interruptions can result from hardware failure, poor hardware configuration, or networking implementation issues.
  - ✓ To recover from failure, some kind of **state information** may be maintained on both the client and the server.
  - ✓ NFS takes a simple approach, implementing a stateless **DFS**. It assumes that a client request for a file read or write would not have occurred unless the file system had been **remotely mounted** and the file had been previously open. The NFS protocol carries all the information needed to locate the appropriate file and perform the requested operation.

4.6.3 Consistency Semantics

- ✓ **Consistency Semantics** represent an important criterion for evaluating any file system that supports file sharing. These semantics specify how multiple users of a system access a shared file simultaneously and they specify when modifications of data by one user will be observable by other users. These semantics are typically implemented as code with the file system.
- ✓ A **series of file accesses (**that is, reads and writes) attempted by a user to the same file is always enclosed between the open() and close() operations. The series of accesses between the open() and close() operations makes up a **file session.**
- ✓ Several **examples** of consistency semantics are,

- **UNIX Semantics**

  - ✓ The UNIX file system uses the **following consistency semantics**,
    - o Writes to an open file by a user are visible immediately to other users who have this file open.
    - o One mode of sharing allows users to share the pointer of current location into the file. Thus, the advancing of the pointer by one user affects all sharing users. Here, a file has a single image that interleaves all accesses, regardless of their origin.
  - ✓ In the UNIX semantics, a file is associated with a single physical image that is accessed as an exclusive resource. Contention for this single image causes delays in user processes.

- **Session Semantics**

  - ✓ The **Andrew file system (AFS)** uses the following **consistency semantics,**
    - o Writes to an open file by a user are not visible immediately to other users that have the same file open.
    - o 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.
  - ✓ According to these semantics, a file may be associated temporarily with several images at the same time. Multiple users are allowed to perform both read and write accesses concurrently on their images of the file, without delay.

- **Immutable-Shared-Files Semantics**

  - ✓ A unique approach is that **immutable shared files**. Once a file is declared as shared by its creator, it cannot be modified. An immutable file has **two key properties** i.e., its name may not be **reused,** and its contents may not be **altered**.

4.7 Protection

- ✓ Information must be protected from a physical damage and improper access i.e., reliability and protection.
- ✓ Protection can be provided in many ways. For a small single-user system,
- ✓ we might provide protection by physically removing the floppy disks and
- ✓ Locking them in a desk drawer or file cabinet. In a multiuser system other mechanisms are needed.

4.7.2 Types of Access

- ✓ Protection mechanisms provide **controlled access** by limiting the types of file access that can be made. Access is **permitted or denied** depending on several factors, one of which is the type of access requested. Several different types of operations may be controlled. They are ,
  - o **Read.** Read from the file.
  - o **Write**. Write or rewrite the file.
  - o **Execute**. Load the file into memory and execute it.
  - o **Append**. Write new information at the end of the file.
  - o **Delete**. Delete the file and tree its space for possible reuse.
  - o **List.** List the name and attributes of the file.
- ✓ Other operations, such as renaming, copying, and editing the file, may also be controlled. For many systems these higher-level functions may be implemented by a system program that makes lower-level system calls.

4.7.3 Access Control

- ✓ Different users may need different types of access to a file or directory.
- ✓ The most general scheme to implement identity dependent access is to associate with each file and directory an **Access Control List (ACL)** specifying user names and the types of access allowed for each user.
- ✓ When a user requests access to a particular file, the operating system checks the **access list** associated with that file. If that user is listed for the requested access, the access is allowed. Otherwise, a protection violation occurs, and the user job is denied access to the file.

- ✓ This approach has the **advantage** of enabling complex access methodologies.
- ✓ The main **problem** with access lists is their **length**. If we want to allow everyone to read a file, we must list all users with read access. This technique has **two undesirable consequences**,
  - o Constructing such a list may be a tedious and unrewarding task, especially if we do not know in advance the list of users in the system.
  - o The directory entry, previously of fixed size, now must be of variable size, resulting in more complicated space management.
- ✓ These problems can be **resolved** by use of a **condensed version** of the access list.
- ✓ To condense the length of the access-control list, many systems recognize **three classifications of users** in connection with each file as follows,
  - o **Owner**. The user who created the file is the owner.
  - o **Group.** A set of users who are sharing the file and need similar access is a group, or work group.
  - o **Universe.** All other users in the system constitute the universe.

4.7.4 ***Other Protection Approaches***

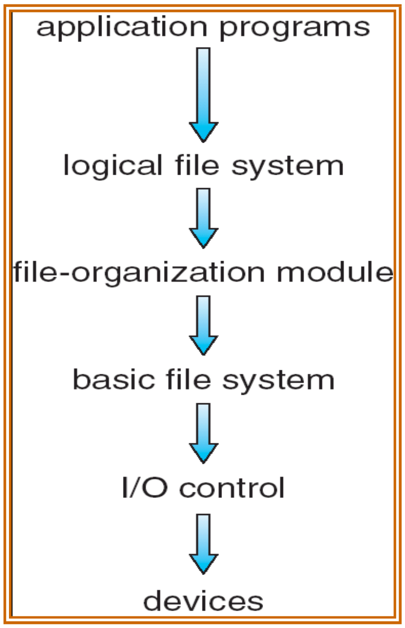
- ✓ Another approach to the protection problem is to associate a **password** with each file. If the passwords are chosen randomly and changed often, this scheme may be effective.
- ✓ The use of passwords has a few **disadvantages. First,** the number of passwords that a user needs to remember may become large. **Second,** if only one password is used for all the files, then once it is discovered, all files are accessible.
- ✓ Some systems allow a user to associate a password with a subdirectory, rather than with an individual file, to deal with this problem.
- ✓ In a multilevel directory structure, we need to protect not only individual files but also collections of files in subdirectories. We want to control the creation and deletion of files in a directory.

File System Implementation

4.8 ***File System Structure***

- ✓ Disks provide bulk of secondary storage on which the file system is maintained. Disks have **two characteristics:**
  - o They can be rewritten in place i.e., it is possible to read a block from the disk to modify the block and to write back in to same place.
  - o They can access any given block of information on the disk. Thus it is simple to access any file either sequentially or randomly and switching from one file to another.
- ✓ To provide efficient and convenient access to the disks, OS imposes one or more **file system** to allow the data to be **stored, located and retrieved easily**.
- ✓ A file system poses **two different design problems.** 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. The **second problem** is creating algorithms and data structures to map the logical file system onto the physical secondary-storage devices.
- ✓ File system itself is composed of many different levels. Layered design is shown in below **figure**.
- ✓ 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.

- ✓ The **basic file system** issue generic commands to the appropriate device driver to read and write physical blocks on the disk.
- ✓ The **file-organization module** knows about files and their logical blocks, as well as physical blocks. It also includes free-space manager.
- ✓ **Logical file system** manages **metadata information.** Metadata includes all of the file-system structure except actual data.
- ✓ FCB contains information about the file including the ownership permission and location of the file contents. Most operating system supports more than one type of file system.

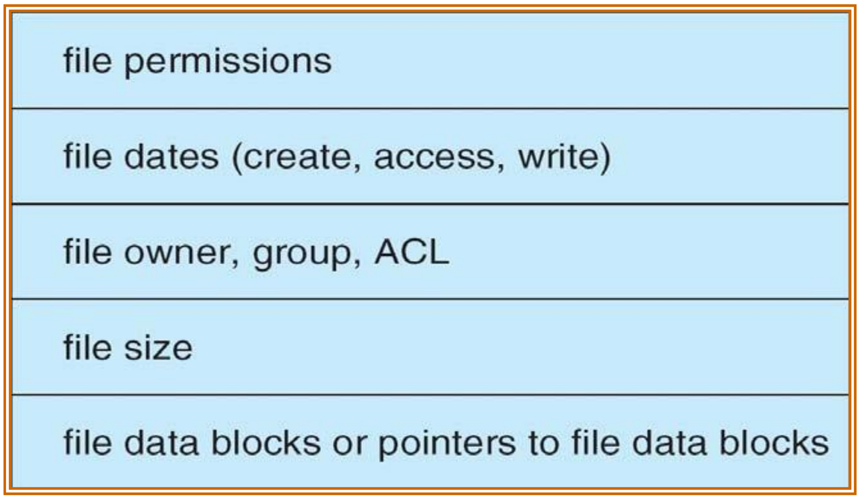


4.9 File System Implementation

4.9.1 Overview

- ✓ File system is implemented on the disk and the memory.
- ✓ The implementation of the file system varies according to the OS and the file system, but there are some general principles.
- ✓ If the file system is implemented on the disk it contains the following information:
  - **Boot Control Block** can contain information needed by the system to boot an OS from that partition. If the disk has no OS, this block is empty. It is the first block of the partition. In **UFS** it is called **boot block** and in **NTFS** it is **partition boot sector**.
  - **Partition control Block** contains volume or partition details such as the number of blocks in partition, size of the blocks, and number of free blocks, free block pointer, free FCB count and FCB pointers. In **NTFS** it is stored in **master file tables**, In **UFS** this is called **super block**.
  - Directory structure is used to organize the files. In UFS, this includes file names and associated **inode** numbers. In NTFS, it is stored in the **master file table**.
  - An FCB contains many of the files details, including file permissions, ownership, size, location of the data blocks. In UFS this is called **inode**, In NTFS this information is actually stored within **master file table**.

- ✓ The **In-memory information** is used for both file-system management and performance improvement via caching. The data are loaded at mount time and discarded at dismount. The structures includes,
  - An in-memory mount table containing information about each mounted information.
  - An in-memory directory structure that 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.
- ✓ To create a new file, an application program calls the logical file system. The logical file system knows the format of the directory structures. To create a new file, it allocates a new FCB. The system then reads the appropriate directory into memory, updates it with the new file name and FCB, and writes it back to the disk. A typical **FCB** is shown in below **figure.**



file permissions

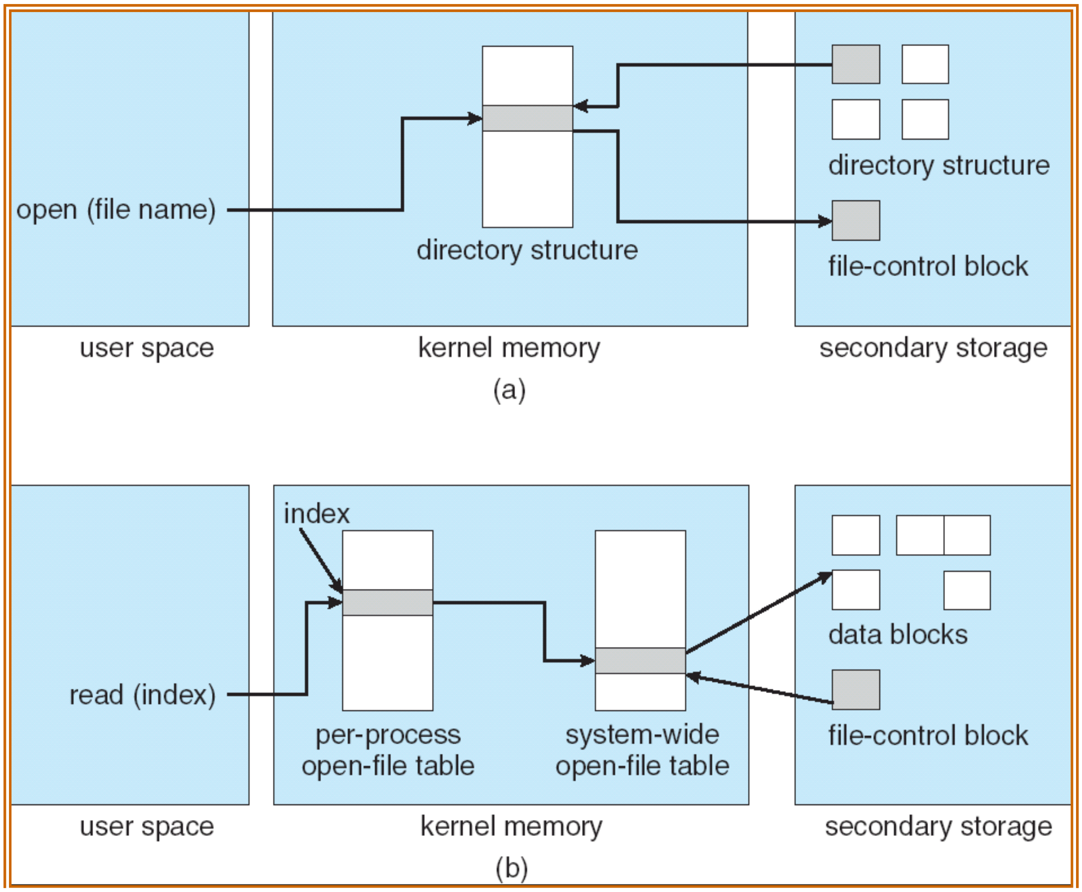
file dates (create, access, write)

file owner, group, ACL

file size

file data blocks or pointers to file data blocks

- ✓ File must be opened before using it for I/O. The open() call passes a file name to the file system. The open() system call searches the system-wide open-file table to find the file name given by the user. If it is open, a per-process open-file table entry is created pointing to the existing system-wide open-file table.
- ✓ When a file is opened, the directory structure is searched for the given file name
- ✓ The open() call returns a pointer to the appropriate entry in the per-process file-system table. All file operations are performed via this pointer.
- ✓ The name given to the entry varies. UNIX systems refer to it as a **file descriptor**, Windows refers to it as a **file handle.**
- ✓ When a process closes the file, the per-process table entry is removed, and system-wide entry's open count is decremented.
- ✓ The below **figure** illustrates the necessary file system structures provided by the operating systems.

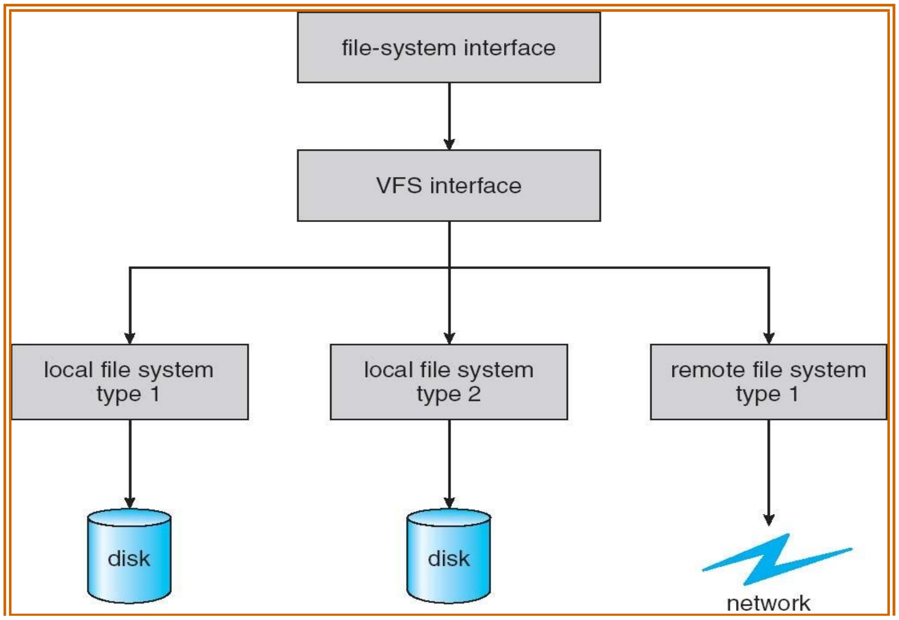


4.9. 2 Partition and Mounting

- ✓ A disk can be divided in to multiple partitions. Each partition can be either raw i.e., containing no file system or cooked i.e., containing a file system.
- ✓ Raw disk is used where no file system is appropriate. UNIX swap space can use a raw partition and do not use file system.
- ✓ Some db uses raw disk and format the data to suit their needs. Raw disks can hold information need by disk RAID (Redundant Array of Independent Disks) system.
- ✓ Boot information can be stored in a separate partition. Boot information will have their own format. At the booting time system does not load any device driver for the file system. Boot information is a sequential series of blocks, loaded as an image in to memory.
- ✓ **Dual booting** is also possible on some PCs where more than one OS are loaded on a system.
- ✓ A boot loader understands multiple file system and multiple OS can occupy the boot space once loaded it can boot one of the OS available on the disk. The disks can have multiple partitions each containing different types of file system and different types of OS.
- ✓ Boot partition contains the OS kernel and is mounted at a boot time.
- ✓ The operating system notes in its in-memory mount table that a file system is mounted, along with the type of the file system. Microsoft window based systems mount each partition in a separate **name space** denoted by a letter and a colon. On UNIX, file system can be mounted at any directory.

4.9.3 Virtual File Systems

- ✓ Modern operating systems must concurrently support multiple types of file systems.
- ✓ Data structures and procedures are used to isolate the basic system call functionality from the implementation details. The file-system implementation consists of **three major layers** as shown in below **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** layer. The VFS layer serves **two important functions**.
- ✓ The virtual file system (VFS) layer, serves **two important functions**:
  - o 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 which are mounted locally.
  - o The VFS 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.
- ✓ The VFS activates file-system-specific operations to handle local requests according to their file-system types and even calls the NFS protocol procedures for remote requests. File handles are constructed from the relevant vnodes and are passed as arguments to these procedures. The layer implementing the file system type or the remote-file-system protocol is the **third layer** of the architecture.



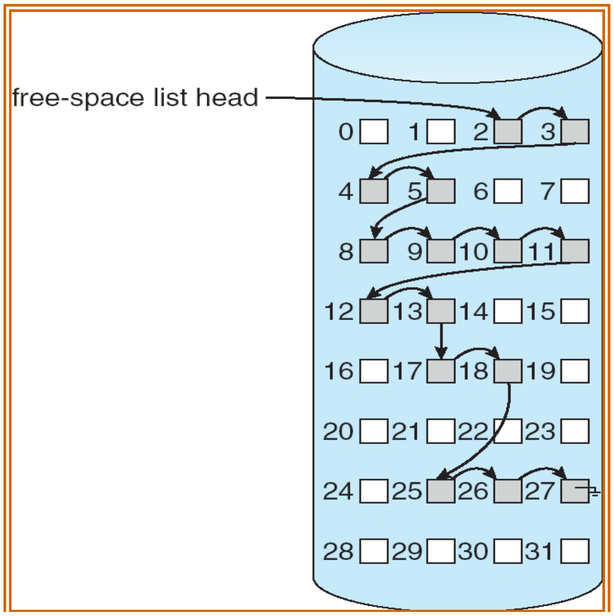
- o The **four main object** types defined by the Linux VFS are:
  - o The **inode object**, which represents an individual file.
  - o The **file object,** which represents an open file.
  - o The **superblock object**, which represents an entire file system.
  - o The **dentry object**, which represents an individual directory entry.

4.10 Directory Implementation

Directory is implemented in **two ways,**

4.10.1 Linear list

- ✓ Linear list is a simplest method.
- ✓ It uses a linear list of file names with pointers to the data blocks. It uses a linear search to find a particular entry as shown in below **figure below.**
- ✓ It is **simple** for programming but **time consuming** to execute.
  - ✓ To create a new file, we must first search the directory to be sure that no existing file has the same name. Then, we 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. We can **mark** the entry as unused or we can **attach** it to a list of free directory entries. A **third** alternative is to copy the last entry in the directory into the freed location and to decrease the length of the directory.
- ✓ A linked list can also be used to decrease the time required to delete a file. Linear search is the main **disadvantage.**
- ✓ An **advantage** of the sorted list is that a sorted directory listing can be produced without a separate sort step.



4.10.2 Hash table

- ✓ Hash table decreases the directory search time.
- ✓ Insertion and deletion are fairly straight forward.
- ✓ Hash table takes the value computed from that file name and it returns a pointer to the file name in the linear list.
- ✓ Insertion and deletion are straightforward, but some provision must be made for **collision** i.e; situations in which two file names hash to the same location.
- ✓ The **major difficulties** with a hash table are it is generally **fixed size** and the dependence of the hash function on that size.

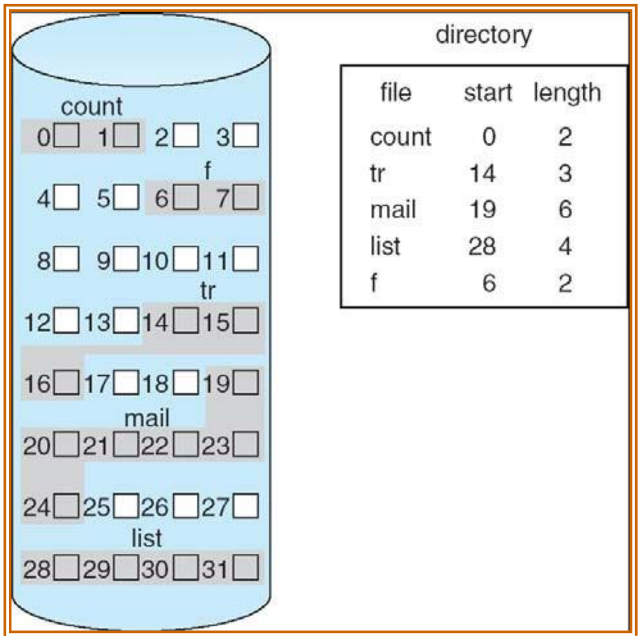
- ✓ A **chained-overflow hash table** can be used where each hash entry can be a linked list instead of an individual value, and we can resolve collisions by adding the new entry to the linked list.

4.11 Allocation Methods (storage mechanisms available to store files)

Three major methods of allocating disk space are,

4.11.1 Contiguous Allocation

- ✓ A **single set of blocks** is allocated to a file at the time of file creation. This is a **pre-allocation strategy** that uses portion of variable size.
- ✓ The file allocation table needs single entry for each file, showing the **starting block and the length of the file.**
- ✓ The below **figure** shows the contiguous allocation method.

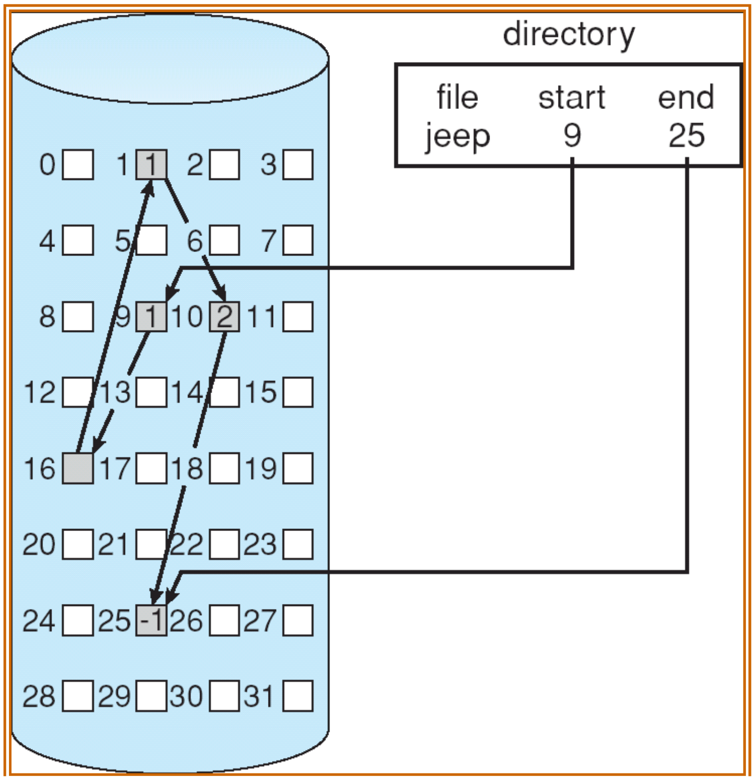


file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

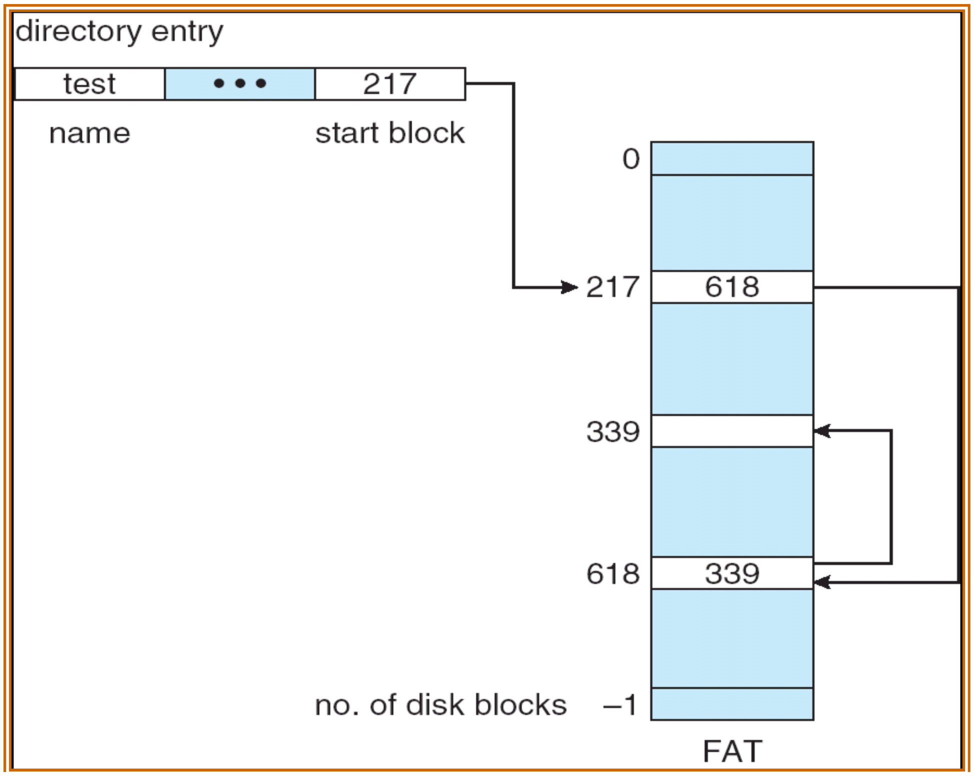
- ✓ Contiguous allocation algorithm suffers from **external fragmentation.**
- ✓ **Sequential and direct access** can be supported by contiguous allocation.
- ✓ Compaction is used to solve the problem of external fragmentation.
- ✓ Another **problem** with contiguous allocation algorithm is **pre-allocation**, that is it is necessary to declare the size of the file at the time of creation.
- ✓ Even if the total amount of space needed for a file is known in advance, preallocation may be inefficient. A file that will grow slowly over a long period must be allocated enough space for its final size, even though much of that space will be unused for a long time. The file therefore has a large amount of **internal fragmentation.**
- ✓ To **minimize these drawbacks**, some operating systems use a modified ontiguous-allocation scheme. Here, a contiguous chunk of space is allocated initially; then, if that amount proves not to be large enough, another chunk of contiguous space, known as an **extent** is added. The location of a file's blocks is then recorded as a location and a block count, and a link to the first block of the next extent.

4.11.2 Linked Allocation

- It **solves the problem of contiguous allocation**. This allocation is on the basis of an individual block. Each block contains a **pointer** to the next block in the chain.
- The disk block can be scattered anywhere on the disk.
- The **directory** contains a pointer to the first and the last blocks of the file.
- The below **figure** shows the linked allocation. To create a new file, simply create a new entry in the directory.

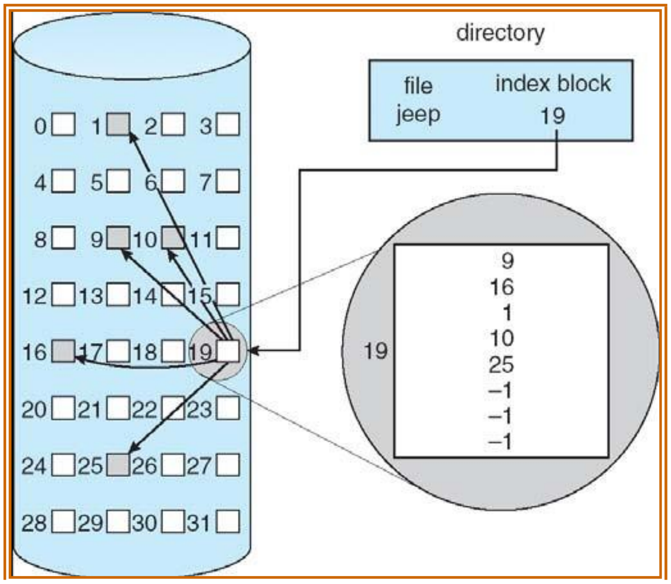


- There is **no external fragmentation** since only one block is needed at a time.
- The size of a file need not be declared when it is created. A file can continue to grow as long as free blocks are available.
- The major **problem** is that it can be used effectively only for **sequential-access** files. Another disadvantage is the **space required for the pointers**. The usual **solution** to this problem is to collect blocks into multiples, called **clusters** and to allocate clusters rather than blocks.
- Another problem of linked allocation is **reliability.** One partial solution is to use doubly linked lists, and another is to store the file name and relative block number in each block.
- An important **variation on linked allocation** is the use of a **File Allocation Table (FAT).** 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 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. This 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**. An **illustrative example** is the FAT structure shown in below **figure 11.8** for a file consisting of disk blocks 217, 618, and 339.



4.11.3 Indexed Allocation

- ✓ The **file allocation table** contains a separate one level **index** for each file. The index has one entry for each portion allocated to the file.
- ✓ The **ith** entry in the index block points to the **ith** block of the file. The below **figure 11.8** shows indexed allocation.



- ✓ The indexes are not stored as a part of file allocation table but they are kept as a **separate block** and the entry in the file allocation table points to that block.

- ✓ Allocation can be made on either fixed size blocks or variable size blocks. When the file is created all pointers in the index block are set to **nil.** When an entry is made a block is obtained from **free space manager**.
- ✓ Indexed allocation supports both **direct access and sequential access** to the file. It supports direct access, without suffering from external fragmentation, because any free block on the disk can satisfy a request for more space. Indexed allocation also suffer from wasted space.
- ✓ The **pointer overhead** of the index block is generally **greater than** the pointer overhead of linked allocation.
- ✓ Every file must have an index block, so we want the index block to be as small as possible. If the index block is **too small** 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,
  - o **Linked scheme**
    - ✓ 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 is **nil** or is a pointer to another index block.
  - o **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.
  - o **Combined scheme**
    - ✓ Another alternative, used in the UFS, is to keep the First 15 pointers of the index block in the file's inode. 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 do not need a separate index block.
    - ✓ 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**.
    - ✓ In this method, the number of blocks that can be allocated to a file exceeds the amount of space addressable by the four-byte file pointers used by many operating systems.

<u>4.11.4</u> ***<u>Performance</u>***

- ✓ The allocation methods vary in their storage efficiency and data-block access times. Both are important criteria in selecting the proper methods for an operating system to implement.
- ✓ For any type of access, **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; but for direct access an access to the ith block might require i disk reads.
- ✓ The operating system must have appropriate data structures and algorithms to support both allocation methods.
- ✓ Files can be **converted** from one type to another by the creation of a new file of

- the desired type, into which the contents of the old file are copied. The old file may then be deleted and the new file renamed.
- **Indexed allocation** is more complex. If the index block is already in memory, then the access can be made directly. But 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. For a two-level index, two index-block reads might be necessary. For an extremely large file, accessing a block near the end of the file would require reading in all the index blocks before the needed data block finally could be read. Thus, **the performance of indexed allocation** depends on the **index structure, on the size of the file, and on the position of the block desired.**
- Some systems combine contiguous allocation with indexed allocation by using contiguous allocation for small files and automatically switching to an indexed allocation if the file grows large. Since most files are small, and contiguous allocation is efficient for small files, average performance can be quite good.

4.12 Free Space Management

- Since disk space is limited, we need to reuse the space from deleted files for new Files. **To keep track of free disk space,** the system maintains a **free-space list**.
- The free-space list records all free disk blocks-those not allocated to some file or directory. To **create a file**, we 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.
- There are **different methods** to manage free space.

4.12.1 Bit Vector

- The free-space list is implemented as **bit vector** or **bit map**.
- Each block is represented by 1 bit. If the **block is free, the bit is 1; if the block is allocated, the bit is 0.**
- **For example**, consider a disk where blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17,
- 18, 25, 26, and 27 are free and the rest of the blocks are allocated. The free-space bit map would be 001111001111110001100000011100000 ...
- The **main advantage** of this approach is its relative **simplicity** and its **efficiency** in finding the first free block or **n consecutive free blocks** on the disk.

4.12.2 Linked List

- Another approach to free-space management is to link together all the free disk blocks, keeping a pointer to the first free block in a special location on the disk and caching it in memory.
- This first block contains a pointer to the next free disk block, and so on.
- This scheme is not efficient because to traverse the list, we must read each block, which requires substantial I/0 time.
- The operating system simply needs a free block so that it can allocate that block to a file, so the first block in the free list is used.

4.12.3 ***Grouping***

- ✓ A modification of the free-list approach stores the addresses of n free blocks in the first free block. The **first n-1** of these blocks are actually free. The last block contains the addresses of other n free blocks, and so on.
- ✓ The addresses of a large number of free blocks can be found quickly, unlike the situation when the linked-list approach is used.

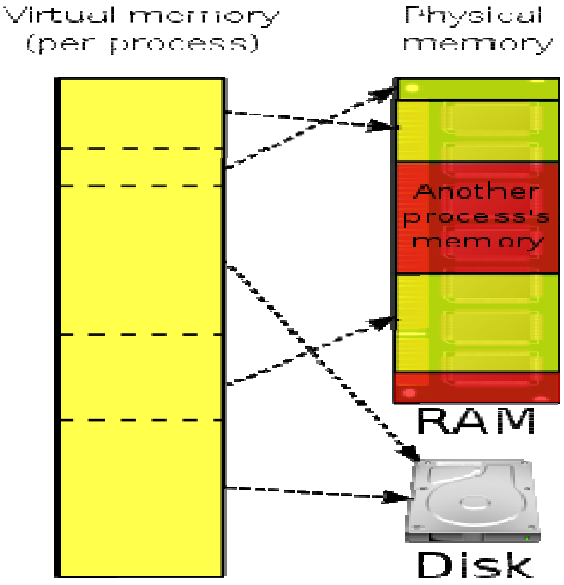
4.12.4 ***Counting***

- ✓ Another approach takes advantage of several contiguous blocks may be allocated or freed simultaneously.
- ✓ Rather than keeping a list of n free disk addresses, we can keep the address of the first free block and the number (n) of free contiguous blocks that follow the first block.
- ✓ Each entry in the free-space list then consists of a **disk address and a count.**

VIRTUAL MEMORY MANAGEMENT

Virtual memory is a memory management capability of an OS that uses hardware and software to allow a computer to compensate for physical memory shortages by temporarily transferring data from random access memory (RAM) to disk storage

Virtual memory is a technique that allows the execution of processes that are not completely in memory. The main advantage of this scheme is that programs can be larger than physical memory. Virtual memory also allows processes to share files easily and to implement shared memory. It also provides an efficient mechanism for process creation. But virtual memory is not easy to implement.



4.1 Background

An examination of real programs shows us that, in many cases, the entire program is not needed to be in physical memory to get executed. Even in those cases where the entire program is needed, it may not need all to be at the same time.

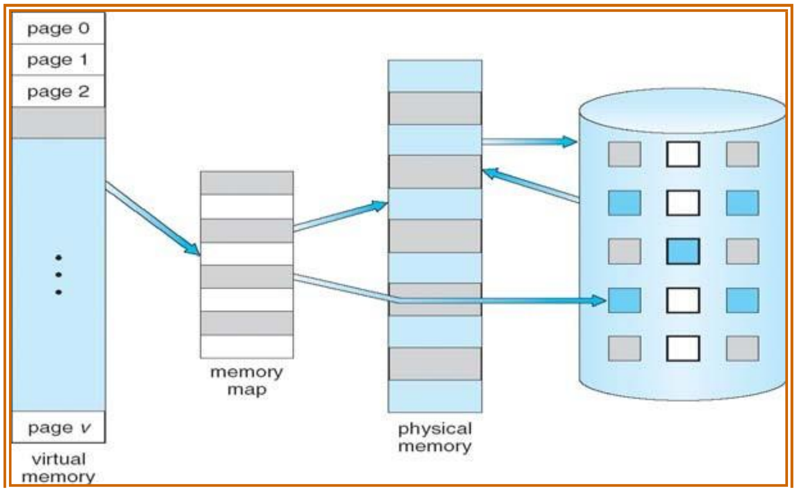
The ability to execute a program that is only partially in memory would confer many

A program will not be limited by the amount of physical memory that is available.

benefits:

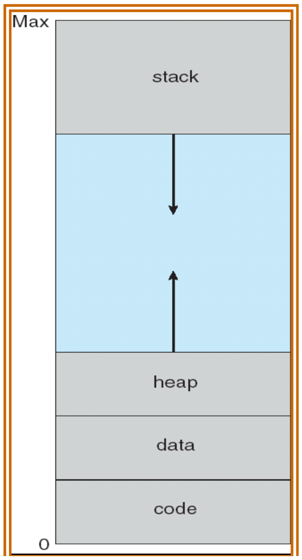
- More than one program can run at the same time which can increase the throughput and CPU utilization.
- Less **I/O** operation is needed to swap or load user program in to memory. So each user program could run faster.

Virtual memory involves the **separation of user's logical memory from physical memory.** This separation allows an extremely large virtual memory to be provided for programmers when there is small physical memory as shown in below **figure.**



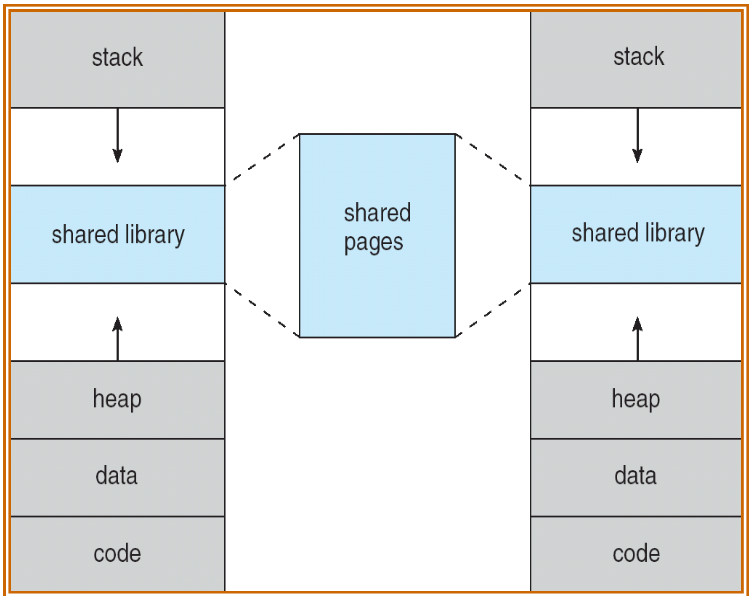
The **virtual address space** of a process refers to the logical (or virtual) view of how a process is stored in memory. In below **figure, heap is allowed** to grow **upward** in memory as it is used for dynamic memory allocation and **stack** to grow **downward** in memory through successive function calls.

The **large blank space (or hole)** between the heap and the stack is part of the virtual address space but will require actual physical pages only if the heap or stack grows. Virtual address spaces that include holes are known as **sparse address spaces**.



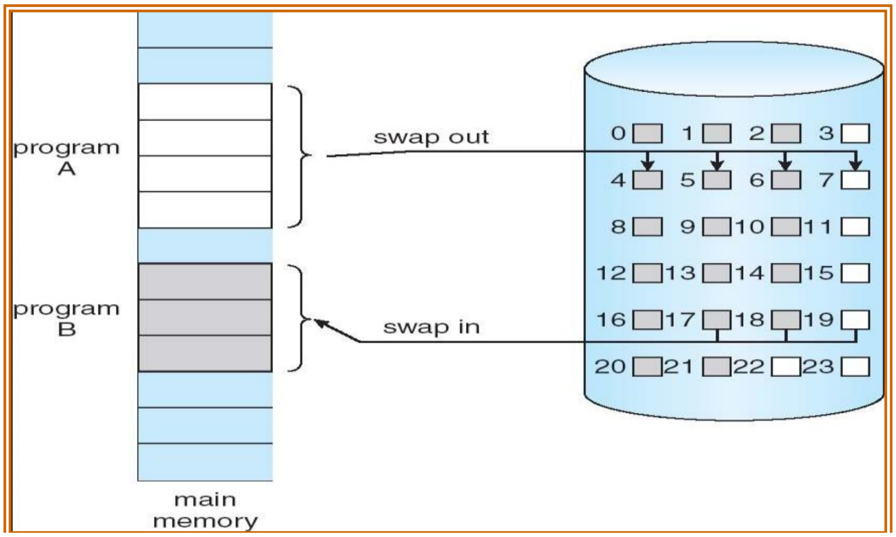
Virtual memory allows files and memory to be shared by two or more processes through **page sharing**. This leads to the following **benefits,**

- **System libraries** can be shared by several processes through mapping of the shared object into a virtual address space as shown in below **figure.**
- Virtual memory allows one process to create a region of memory that it can share with another process as shown in below **figure**.
- Virtual memory can allow pages to be shared during **process creation** with the fork() system call thus speeding up process creation.



4.2 Demand Paging

The process of loading the page into memory on demand (whenever page fault occurs) is known as demand paging.



A demand paging system is quite similar to a paging system with swapping where processes reside in secondary memory and pages are loaded only on demand i,e **lazy swapper (never swaps** a page into memory unless that page will be needed), not in advance. When a context switch occurs, the operating system does not copy any of the old program's pages out to the disk or any of the new program's pages into the main memory Instead, it just begins executing the new program after loading the first page and fetches that program's pages as they are referenced. While executing a program, if the program references a page which is not available in the main memory because it was swapped out a little ago, the processor treats this invalid memory reference as a **page fault** and transfers control from the program to the operating system to demand the page back into the memory.

Advantages

Following are the advantages of Demand Paging −

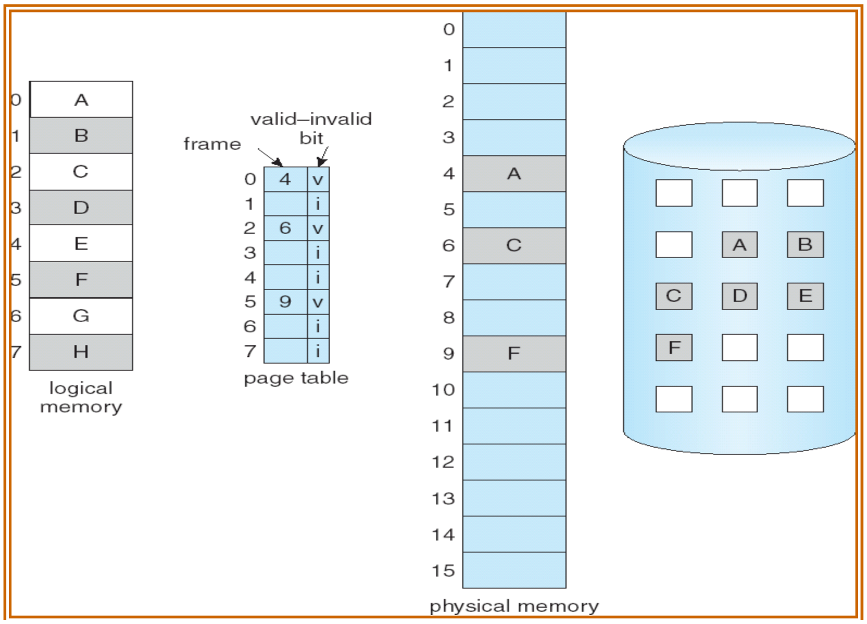
- Large virtual memory.
- More efficient use of memory.
- There is no limit on degree of multiprogramming.

Disadvantages

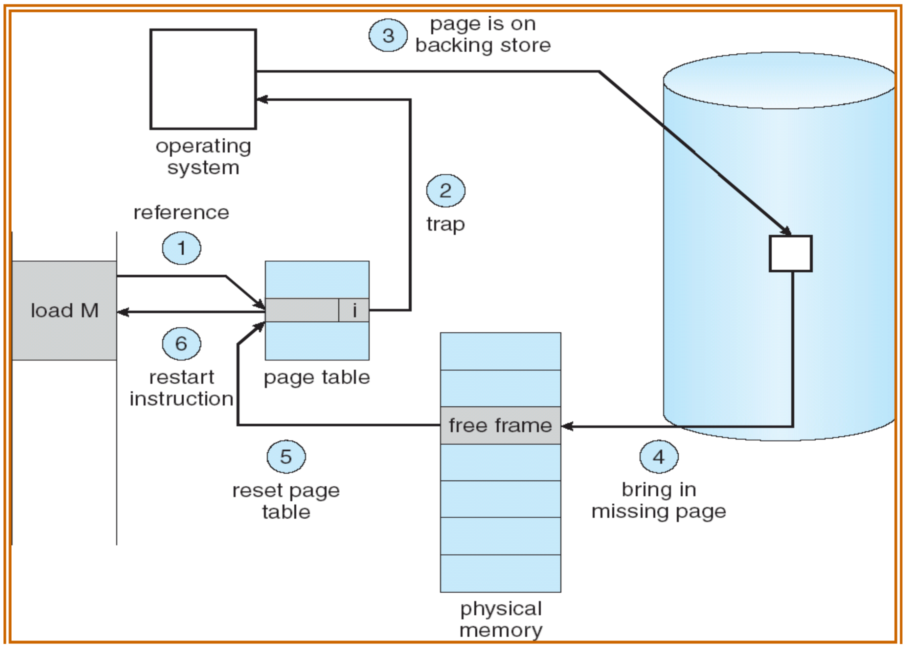
- Number of tables and the amount of processor overhead for handling page interrupts are greater than in the case of the simple paged management techniques.

Implementation:

- The **valid-invalid bit** scheme distinguishes between the pages that are in memory and the pages that are on the disk. If the bit is valid then the page is both legal and is in memory. If the bit is invalid then either the page is not valid or is valid but is currently on the disk**.**
- The **page-table entry** for a page that is brought into memory is set as valid but the page-table entry for a page that is not currently in memory is either simply marked invalid or contains the address of the page on disk as shown in below **figure**.
- Access to the page which is marked as invalid causes a **page fault trap**.



- ***The steps for handling*** page fault is straight forward and is shown in below **figure**,



1. We check the internal table of the process to determine whether the reference made is valid or invalid.
2. If invalid, terminate the process. If valid, then the page is not yet loaded and we now page it in.
3. We find a free frame.
4. We schedule disk operation to read the desired page in to newly allocated frame''.
5. When disk read is complete, we modify the internal table kept with the process to indicate that the page is now in memory.
6. We restart the instruction which was interrupted by the trap. The process can now access the page.

- In extreme cases, executing the process is started **without pages** in memory. When the OS sets the instruction pointer of process which is not in memory, it generates a page fault. After this page is brought in to memory the process continues to execute, faulting every time until every page that it needs is in memory. This scheme is known as **pure demand paging**. That is, it never brings the page in to memory until it is required.

- The **hardware support f**or demand paging is same as paging and swapping.
  - **Page table:** It has the ability to mark an entry invalid through valid-invalid bit.
  - **Secondary memory:** This holds the pages that are not present in main memory. It is a high speed disk. It is known as the **swap device,** and the section of disk used for this purpose is known as **swap space.**

- ***Performance of demand paging***

  - Demand paging can have significant effect on the performance of the computer system.
  - **Effective access time is computed** for a demand-paged memory.

- The memory-access time, denoted **ma**, ranges from 10 to 200 nanoseconds. As long as we have no page faults, the effective access time is **equal to** the memory access time.
- If a page fault occurs, we must first read the relevant page from disk and then access the desired word.
- Let **p** be the **probability** of a page fault (0<=p<=1). The **effective access time** is then,

effective access time = (1 - p) * ma + p * page fault time.

- To compute the effective access time, we must know how much **time** is needed to **service a page fault**. A page fault causes the following **sequence** to occur,

1. Trap to the OS.
2. Save the user registers and process state.
3. Determine that the interrupt was a page fault.
4. Check that the page reference was legal and determine the location of the page on disk.
5. Issue a read from disk to a free frame.
   a. Wait in a queue for this device until the read request is serviced.
   b. Wait for the device seek and/or latency time.
   c. Begin the transfer of the page to a free frame.
6. While waiting, allocate the CPU to some other user.
7. Receive an interrupt from the disk I/O subsystem.
8. Save the registers and process state for the other user.
9. Determine that the interrupt was from the disk.
10. Correct the page table and other table to show that the desired page is now in memory.
11. Wait for the CPU to be allocated to this process again.
12. Restore the user registers, process state and new page table, then resume the interrupted instruction.

- The **three major components** of the **page-fault service time**,
  1. Service the page-fault interrupts.
  2. Read in the page.
  3. Restart the process.

- With an **average page-fault service time** of **8 milliseconds** and a memory access time of **200 nanoseconds**, the effective access time in nanoseconds is

effective access time = (1 - p) * (200) + p (8 milliseconds)
= (1 - p) * 200 + p * 8,000,000
= 200 + 7,999,800 * p.

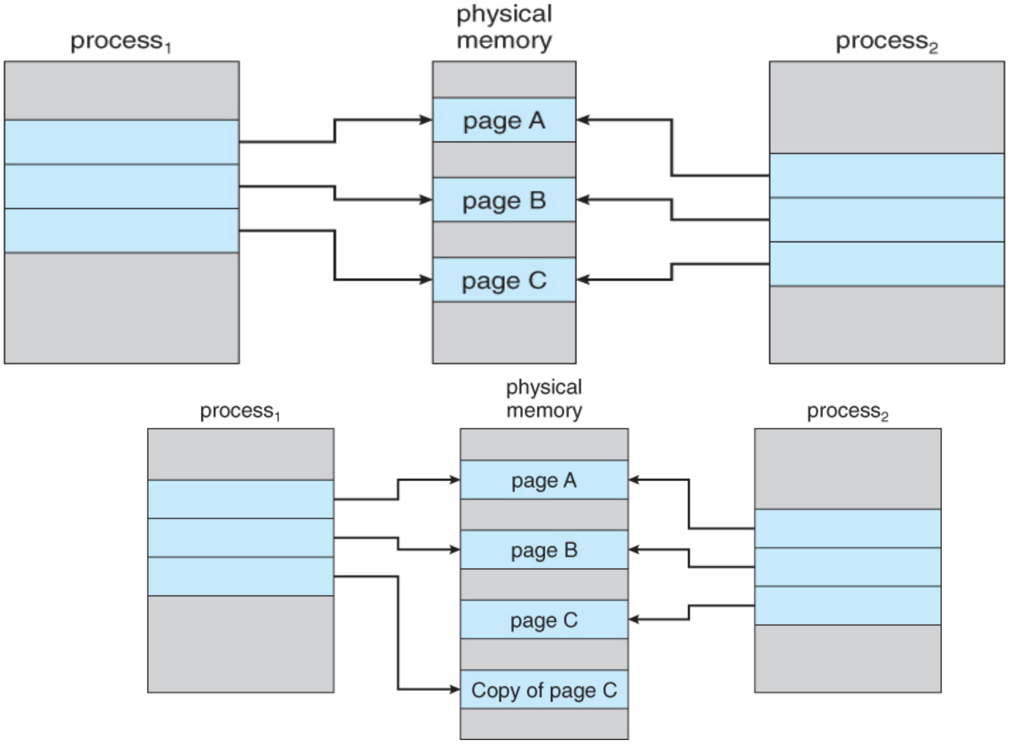
- The effective access time is **directly proportional** to the **page-fault rate.**
- If one access out of 1,000 causes a page fault, the effective access time is **8.2 microseconds.** The computer will be slowed down by a factor of 40 because of demand paging. If we want performance degradation to be less than 10 percent, we need,

$220>200 + 7,999,800 * p,$
$20>7,999,800 * p,$
$p < 0.0000025.$

4.3 Copy-on-write

Copy-on-write technique allows both the parent and the child processes 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.

Copy-on-write is illustrated in below **figure**, which shows the contents of the physical memory **before and after** process 1 modifies page C where process 1 and process 2 are parent and child process.



If a child process tries 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.

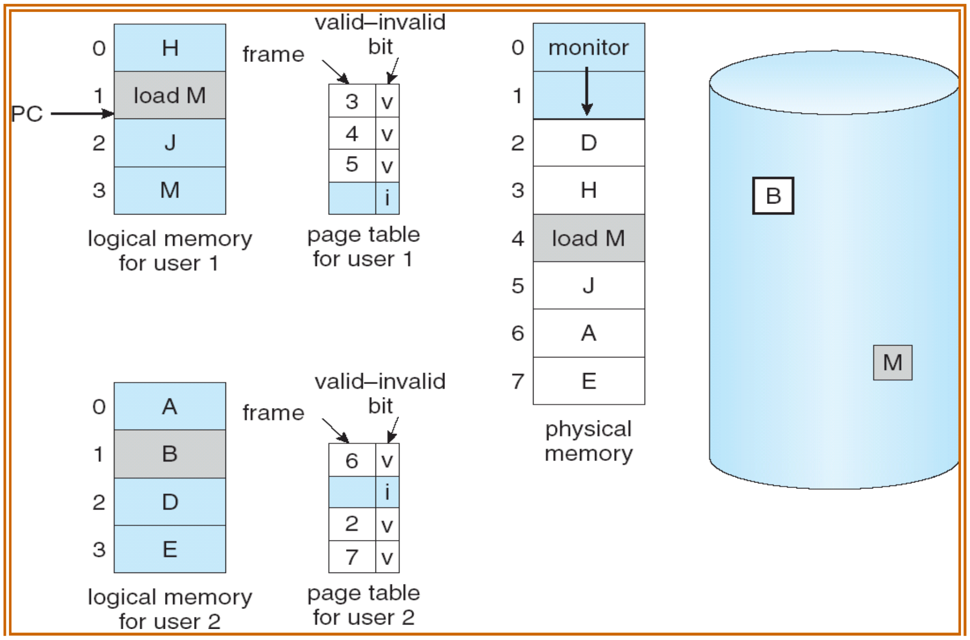
The new pages are obtained from the **pool** of free pages. Operating systems allocate these pages using a **technique** known as **zero-fill-on-demand.** Zero-fill-on-demand pages have been **zeroed-out** before being allocated, thus erasing the previous contents.

4.4 Page Replacement:

Page replacement policy deals with **replacing (removing) pages** from memory to free frames, for bringing in the new pages when desired page is not in main memory.

Page replacement algorithms are the techniques using which an Operating System decide which memory pages to swap out, write to disk when a page of memory needs to be allocated. Paging happens whenever a page fault occurs and a free page cannot be used for allocation purpose accounting to reason that pages are not available or the number of free pages is lower than required pages.

If **page fault** occurs while executing the process, the operating system determines where the desired page is residing on the disk, and finds that there are no free frames on the free frame list as shown in below **figure**.

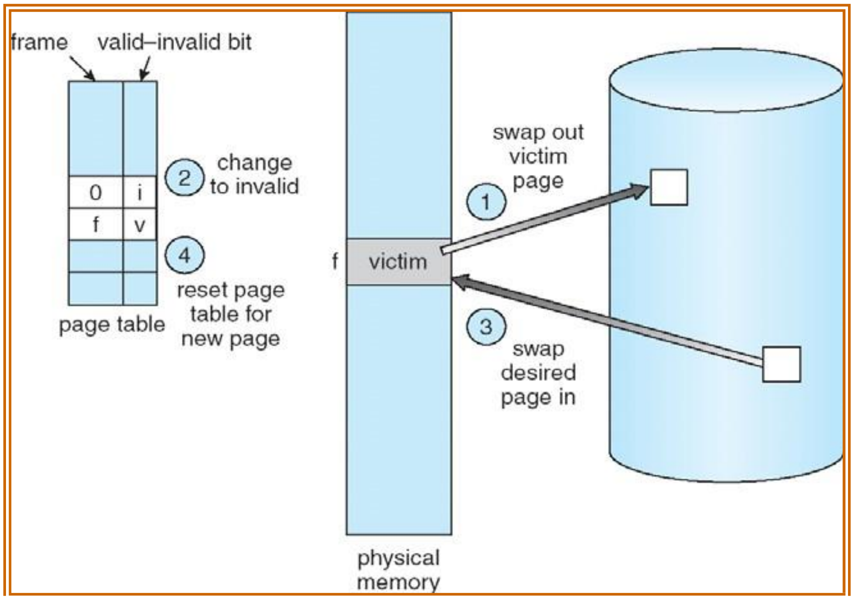


The OS has **several options** like; it could **terminate** the user process or instead **swap out** a process, freeing all its frames and thus reduce the level of multiprogramming.

Basic Page Replacement:

If frame is not free, the frame not currently being used is searched and used for page replacement. The frame is freed by writing its contents to swap space and changing the page table to indicate that the page is no longer in memory as shown in below **figure**. The freed frame is used to hold the page for which the process faulted. The **page-fault service routine** is **modified** as follows to include page replacement,

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



If no frames are free, the **two page transfers** (one out and one in) are required. This will doubles the page-fault service time and increases the effective access time. This overhead can be reduced by using **modify (dirty) bit**. Each page or frame may have modify (dirty) bit associated with it. 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 page is selected for replacement, modify bit must be checked. If the bit is set, then the page is modified and page must be written back to the disk. If the bit is not set then the page has not been modified. Therefore, writing the memory page to the disk can be avoided as it is already there.

Two major problems to implement demand paging

If multiple processes are in memory, it must be decided that how many frames to allocate to each process and which page replacement is needed. The frames must be selected that are to be replaced.

- ✓ **frame allocation algorithm**
- ✓ **Page replacement algorithm**.

.

1. FIFO page replacement algorithm

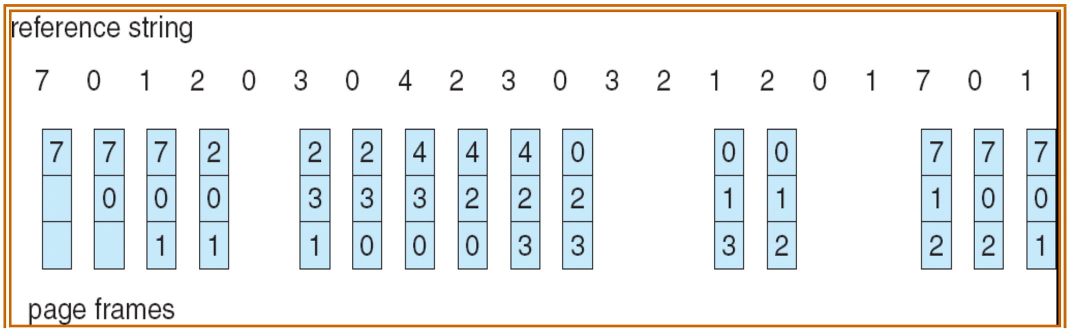
This is the **simplest** page replacement algorithm. A FIFO replacement algorithm associates the time of each page when that page was brought into memory. When a page is to be replaced the oldest one is selected. We replace the queue at the head of the queue. When a page is brought into memory, we insert it at the tail of the queue.

For example, consider the following **reference string** with **frames size 3 and** initially empty.

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

The first three references (7,0,1) causes page faults and are brought into the empty frames.

The next reference 2 replaces page 7 because the page 7 was brought in first. Since 0 is the next reference and 0 is already in memory we have no page fault for this reference. The next reference 3 replaces page 0 so but the next reference to 0 causes page fault. Page 1 is then replaced by page 0. This will continue till the end of string as shown in below **figure** and there are 15 faults all together.

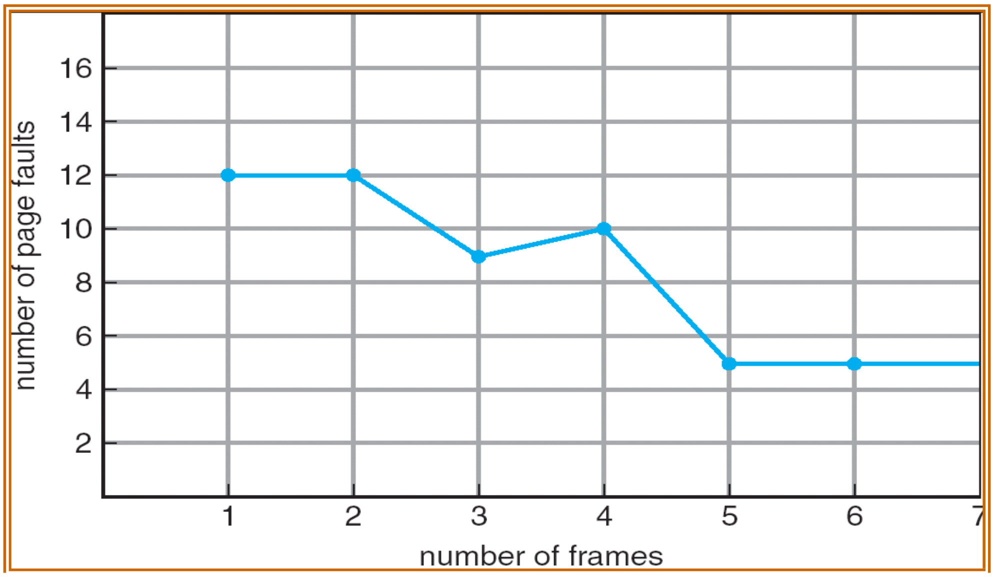


For some page replacement algorithm, the **page fault may increase** as the number of allocated **frames increases**. This is called as **Belady's Anamoly**. FIFO replacement algorithm may face this problem.

To illustrate **Belady's Anamoly** with a FIFO page-replacement algorithm, consider the following reference string.

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

The below **figure 9.12** shows the **curve of page faults** for this reference string versus the number of available frames. The number of faults for four frames (ten) is greater than the number of faults for three frames (nine).



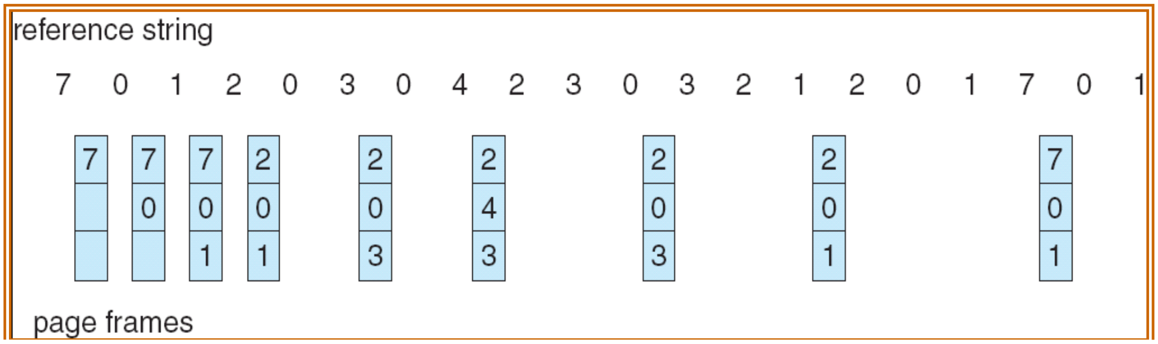
2. *Optimal page replacement algorithm*

Optimal page replacement algorithm is mainly used to **solve** the problem of Belady's Anamoly. Optimal page replacement algorithm has the lowest page fault rate of all algorithms.
An optimal page replacement algorithm is also called OPT or MIN. The working is simple "**Replace the page that will not be used for the longest period of time".**

For example, consider the following **reference string** with **3 frames** initially empty.

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

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. The page 0 will be used at 5 and page 1 at 14. This will continue till the end of the string as shown in **figure.**



- ✓ With only 9 page faults, optimal replacement is much better than a FIFO, which had 15 faults.
- ✓ This algorithm is **difficult** to implement because it requires **future knowledge** of reference strings.

3. *Least Recently Used (LRU) page replacement algorithm*

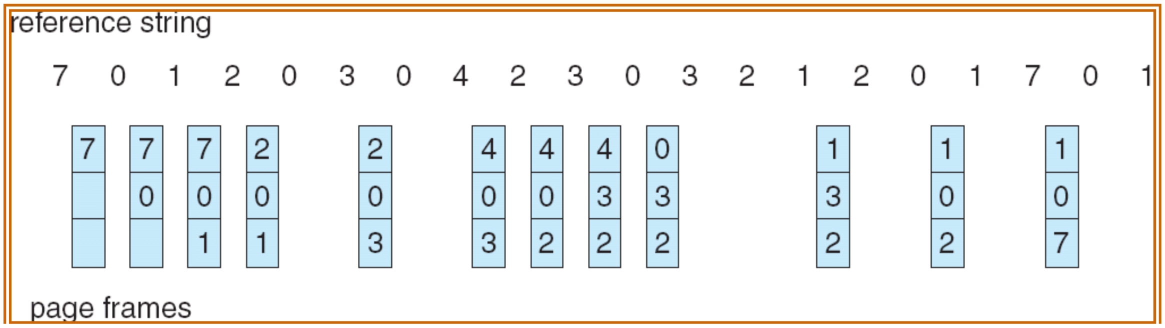
If the optimal algorithm is not feasible, an approximation to the optimal algorithm is possible. The main difference between OPT and FIFO is that, FIFO algorithm uses the time when the pages was brought in and OPT uses the time when a page is to be used. The LRU algorithm "**Replaces the pages that have not been used for longest period of time".**

The LRU associated its pages with the time of that pages last use. This strategy is the optimal page replacement algorithm looking **backward in time** rather than forward.

For example, consider the following **reference string** with **3 frames** initially empty.

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

LRU replacement associates with each page the time of that page's last use. When a page must be replaced LRU chooses the page that has not been used for the longest period of time. The result of applying LRU replacement to our example reference string is shown in below **figure.**

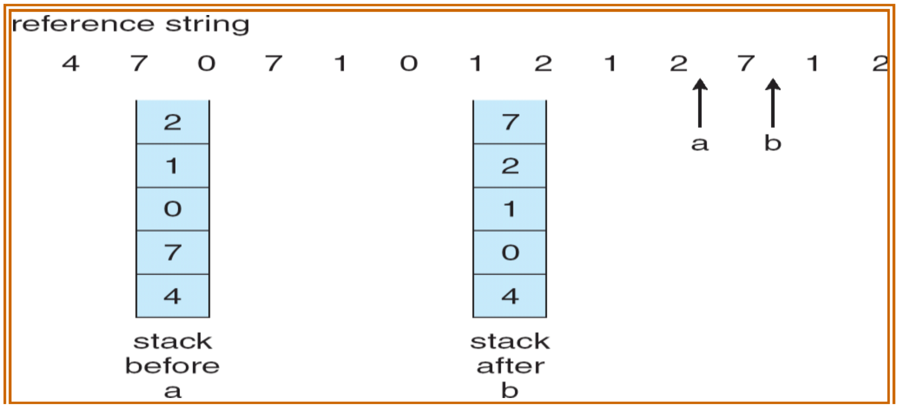


The first 5 faults are similar to optimal replacement. When reference to page 4 occurs, LRU sees that **page 2** is used least recently. The most recently used page is page 0 and just before page 3 was used. The LRU policy is often used as a page replacement algorithm and considered to be **good**.

<u>Two implementations are possible</u>:

- **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 and we replace the page **with smallest time value**. The time must also be maintained when page tables are changed.

- **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 as shown in below **figure 9.15**. 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. Neither optimal replacement nor LRU replacement suffers from Belady's Anamoly. These are called **stack algorithms.**

- 

4. LRU Approximation page replacement algorithm

- ✓ Many systems provide hardware support in the form of a **reference bit.**
- ✓ The reference bit for a page is **set by the hardware** whenever that page is referenced. Reference bits are associated with each entry in the page table.
- ✓ Initially, all bits are **cleared to 0** by the operating system. As a user process executes, the bit associated with each **page is set to 1**.
- ✓ The three **LRU Approximation page replacement algorithms** are as follows,

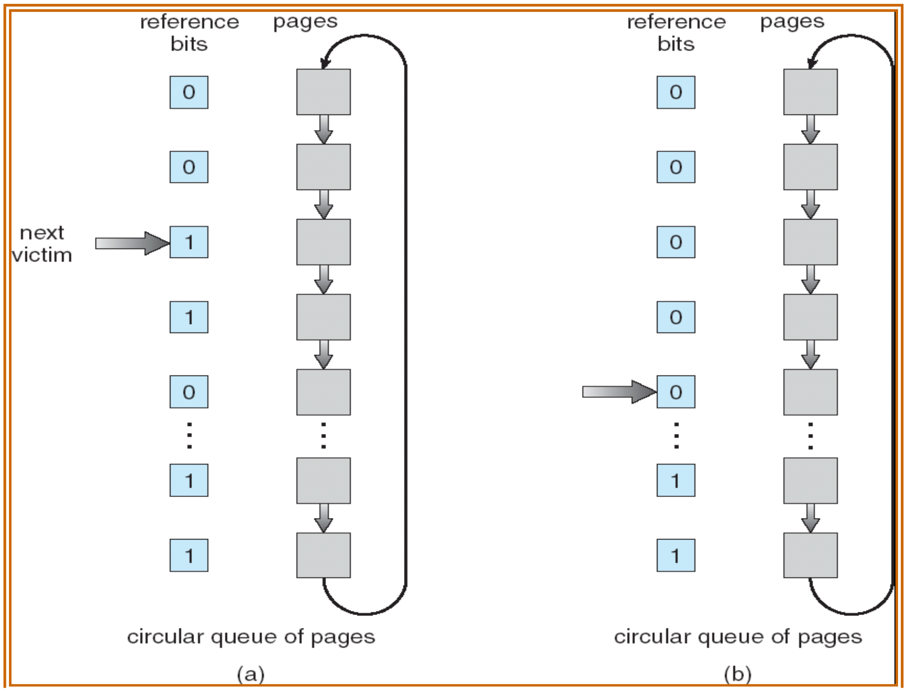
▪ Additional-Reference-Bits Algorithm

- ✓ We can keep an **8-bit byte** 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**.
- ✓ **For example**, if the shift register contains **00000000**, then the page has not been used for eight time periods; a page that is used at least once in each period has a shift register value of **11111111**. A page with a value of **11000100 has been used more recently than one with a value of 01110111**.

  i.e., the page with the lowest number is the LRU page and it can be replaced. In the extreme case, the number can be reduced to zero, leaving only the reference bit itself. This algorithm is called the **Second-Chance Algorithm**

▪ Second-Chance Algorithm

- ✓ The **basic algorithm** of second-chance replacement is a **FIFO** replacement algorithm.
- ✓ When a page has been selected we inspect its reference bit. If the value is 0, we proceed to replace this page; but if the reference bit is **set to 1,** we give the page a **second chance** and move on to select the next FIFO page. When a page gets a second chance, its reference bit is cleared, and its arrival time is reset to the current time. Thus, a page that is given a second chance will not be replaced until all other pages have been replaced.
- ✓ **One way to implement** the second-chance algorithm (**clock algorithm**) is as a **circular queue**. A pointer indicates which page is to be replaced next. When a frame is needed, the pointer advances until it finds a page with a 0 reference bit. As it advances, it clears the reference bits as shown in below **figure 9.16**.
- ✓ Once a victim page is found, the page is replaced, and the new page is inserted in the circular queue in that position.
- ✓ When all bits are set, the pointer cycles through the whole queue, giving each page a second chance. It clears all the reference bits before selecting the next page for replacement. Second-chance replacement degenerates to FIFO replacement if all bits are set.



- **Enhanced Second-Chance Algorithm**

  - ✓ We can **enhance** the second-chance algorithm by considering the **reference bit and the modify bit as an ordered pair.** With these two bits, we have four possible classes,
    - ○ **(0, 0)** neither recently used nor modified - best page to replace.
    - ○ **(0, 1)** not recently used but modified - not quite as good, because the page must be written out before replacement.
    - ○ **(1, 0)** recently used but clean - probably will be used again soon.
    - ○ **(1, 1)** recently used and modified - probably will be used again soon, and the page must be written out to disk before it can be replaced.
  - ✓ Each page is in one of these four classes. We replace the first page encountered in the lowest nonempty class.
  - ✓ The major **difference** between this algorithm and the simpler clock algorithm is that here we give preference to those pages that have been modified to reduce the number of I/Os required.

<u>5. Count Based Page Replacement</u>

- ✓ 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.

- ○ **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.

- **Most Frequently Used(MFU)**
  - ✓ This is based on the principle that the page with the smallest count was probably just brought in and has yet to be used.

6. Page-Buffering Algorithms

- ✓ Systems keep a pool of free frames and when a page fault occurs, a victim frame is chosen as before. The desired page is read into a free frame from the pool before the victim is written out. This procedure allows the process to restart as soon as possible, without waiting for the victim page to be written out. When the victim is later written out, its frame is added to the free-frame pool.
- ✓ An **expansion** of this idea is to maintain a list of **modified pages**. Whenever the paging device is idle, a modified page is selected and is written to the disk. Its modify bit is then reset. This scheme increases the probability that a page will be clean when it is selected for replacement and will not need to be written out.
- ✓ Another modification is to keep a pool of free frames but to remember which page was in each frame. Since the frame contents are not modified when a frame is written to the disk, the old page can be reused directly from the free-frame pool if it is needed before that frame is reused. No I/O is needed in this case. When a page fault occurs, we first check whether the desired page is in the free-frame pool. If it is not, we must select a free frame and read into it.

7. Applications and Page Replacement

- ✓ Applications accessing data through the operating system's virtual memory perform worse than if the operating system provided no buffering at all. An **example** is a **database**, which provides its own memory management and I/0 buffering. Applications like this understand their memory use and disk use better than an operating system that is implementing algorithms for general-purpose use.
- ✓ In another **example, data warehouses** frequently perform massive sequential disk reads, followed by computations and writes. The LRU algorithm would be removing old pages and preserving new ones, while the application would more likely be reading older pages than newer ones. Here, MFU would be more efficient than LRU.
- ✓ Because of such problems, some operating systems give special programs the ability to use a disk partition as a large sequential array of logical blocks, without any file-system data structures. This array is sometimes called the **raw disk**, and I/O to this array is termed **raw I/0**.

4.5 Allocation of Frames:

It is concerned with how we allocate the fixed amount of free memory among the various processes.The simplest case, Consider a single-user system with 128 KB of memory composed of pages 1 KB in size. This system has 128 frames. The operating system may take 35 KB, leaving 93 frames for the user process. When the free-frame list exhaust, a page-replacement algorithm would be used to select one of the 93 in-memory pages to be replaced with the 94th, and so on. When the process terminates the 93 frames would once again be placed on the free-frame list.

Minimum Number of Frames

- ✓ The strategies for the allocation of frames are constrained in various ways. We cannot allocate more than the total number of available frames. We must also allocate at least a minimum number of frames.
- ✓ One reason for allocating at least a minimum number of frames involves performance. As the number of frames allocated to each process decreases, the page-fault rate increases, slowing process execution.
- ✓ Whereas the minimum number of frames per process is defined by the architecture, the maximum number is defined by the amount of available physical memory.

Allocation Algorithms

- ✓ The easiest way to split **m frames** among **n processes** is to give everyone an **equal share** i.e., **m/n** frames. **For example**, if there are **93 frames** and **five processes,** each process will get **18 frames**. The **three leftover frames** can be used as a **free-frame buffer pool**. This scheme is called **equal allocation**.
- ✓ An alternative is to recognize that various processes will need differing amounts of memory. Consider a system with a **1-KB 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**, it does not make **much sense** to give each process **31 frames**. The student process does not need more than 10 frames, so the other 21 are wasted.
- ✓ To **solve this problem**, we can use **proportional allocation** in which we allocate available memory to each process according to **its size**. Let the size of the virtual memory for process $\mathbf{p_i}$ be $\mathbf{s_i}$, and define

$$\mathbf{S=\sum s_i}$$

- ✓ Then, if the **total number of available frames is m**, we allocate $\mathbf{a_i}$ frames to
- ✓ process $\mathbf{p_i}$, where $\mathbf{a_i}$ is approximately

$$\mathbf{a_i = s_i /S * m.}$$

- ✓ We must adjust each $\mathbf{a_i}$ to be an integer that is greater than the minimum number of frames required, with a **sum not exceeding m.**
- ✓ With **proportional allocation**, we would **split 62 frames** between **two processes**, one of 10 pages and one of 127 pages, by allocating **4 frames and 57 frames**, respectively, since

$$\mathbf{10/137 \times 62 \sim 4, and}$$
$$\mathbf{127/137 \ X\ 62 \sim 57.}$$

- ✓ In this way, both processes share the available frames according to their **"needs"** rather than equally.
- ✓ We may want to give the **high-priority process** more memory low-priority processes to speed its execution. One solution is to use a proportional allocation scheme wherein the ratio of frames depends not on the relative sizes of processes but rather on the **priorities of processes** or on a **combination of size and priority.**

- **Global versus Local Allocation**

  ✓ Another important factor in the way frames are allocated to the various processes is page replacement. With multiple processes competing for frames, we can classify **page-replacement algorithms** into **two broad categories**, **Global and local replacement.** The difference between them are,

Global replacement	local replacement
Allows process to replace the frames from the set of all frames, even if that frame is allocated to other process.	Each process selects a replacement only from its own set of allocate frames.
Number of frames will change.	Number of frame does not change.
Increases system throughput	Less system throughput
Process cannot control its own page fault rate	Process can control its own page fault rate
Commonly used	Rarely used

4.6 Thrashing:

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. The process continues to fault; it quickly faults again and again, replacing pages that it must **bring back in** immediately. This high paging activity is called **thrashing.**

Cause of Thrashing

1. **High degree of multiprogramming**: If the number of processes keeps on increasing in the memory than number of frames allocated to each process will be decreased. So, less number of frames will be available to each process. Due to this, page fault will occur more frequently and more CPU time will be wasted in just swapping in and out of pages and the utilization will keep on decreasing.

For example:
Let free frames = 400

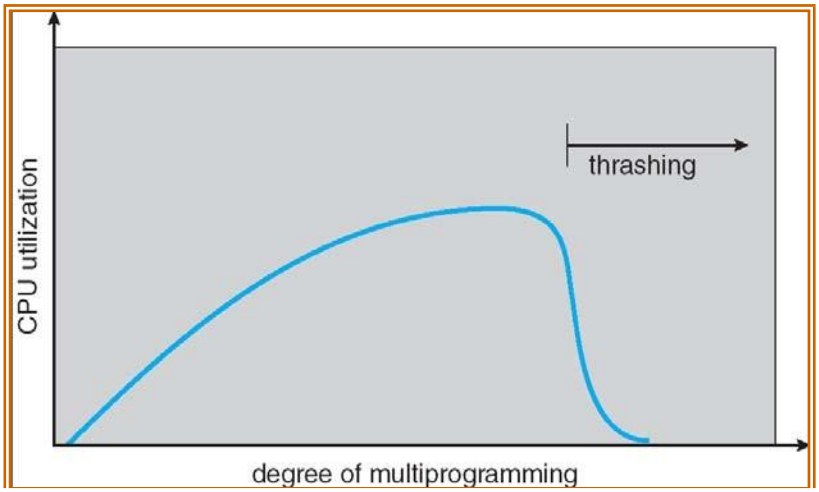
Case 1: Number of process = 100
Then, each process will get 4 frames.

Case 2: Number of process = 400
Each process will get 1 frame.

Case 2 is a condition of thrashing, as the number of processes is increased, frames per process are decreased. Hence CPU time will be consumed in just swapping pages.

2. **Lacks of Frames**: If a process has less number of frames then less pages of that process will be able to reside in memory and hence more frequent swapping in and out will be required. This may lead to thrashing. Hence sufficient amount of frames must be allocated to each process in order to prevent thrashing.

The CPU scheduler sees the decreasing CPU utilization and increases the degree of multiprogramming. The new process tries to get started by taking frames from other processes, causing **more page faults and a longer queue** for the paging device. As a result, CPU utilization drops even further, and the CPU scheduler tries to increase the degree of multiprogramming even more. But this causes **thrashing** and the CPU utilization drops sharply as shown in below **figure**.



Recovery of Thrashing :

- Do not allow the system to go into thrashing by instructing the long term scheduler not to bring the processes into memory after the threshold.
- If the system is already in thrashing then instruct the mid term scheduler to suspend some of the processes so that we can recover the system from thrashing.

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. But how do we know how many frames it "needs"? There are **several techniques.**

- The **working-set strategy** starts by looking at how many frames a process is actually using. This approach defines the locality of process execution.
- The **locality model** states that, as a process executes, it moves from locality to locality. A **locality** is a **set of pages that are actively used**.

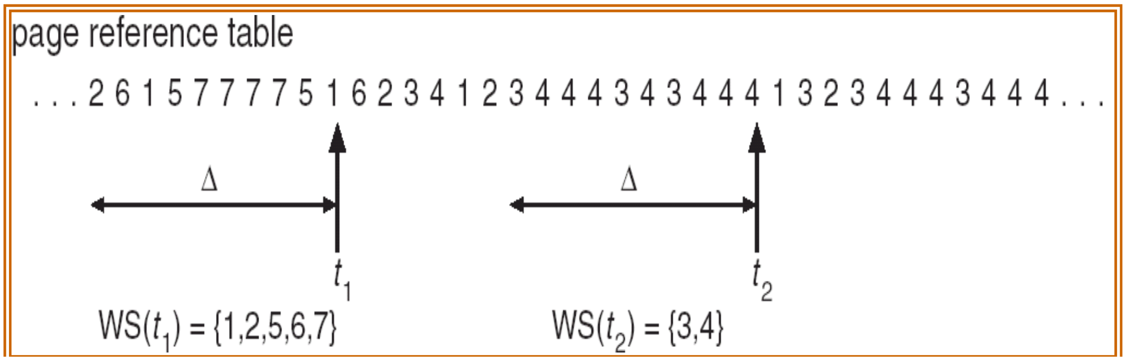
1. Working set model

- ✓ The **working set model** is based on the assumption of locality.
- ✓ This model uses a **parameter** Δ to define the **working set window**.
- ✓ The idea is to examine the most recent Δ **page** references. The set of pages in the most recent Δ page references is the working set as shown in below **figure 9.18**.
- ✓ If a page is in active use, it will be in the working set. If it is no longer being used, it will drop from the working set Δ time units after its last reference. Thus, the working set is an approximation of the program's locality.

- ✓ **For example,** the sequence of memory references is as shown in below **figure**. If Δ = 10 memory references, then the working set at time **t_1** is {1, 2, 5, 6, 7}. By time **t_2**, the working set has changed to {3, 4}.
- ✓ The accuracy of the working set depends on the selection of Δ. The most **important property** of the working set is its **size**. If we compute the working-set size, **WSS_i**, for each process in the system, we can then consider that

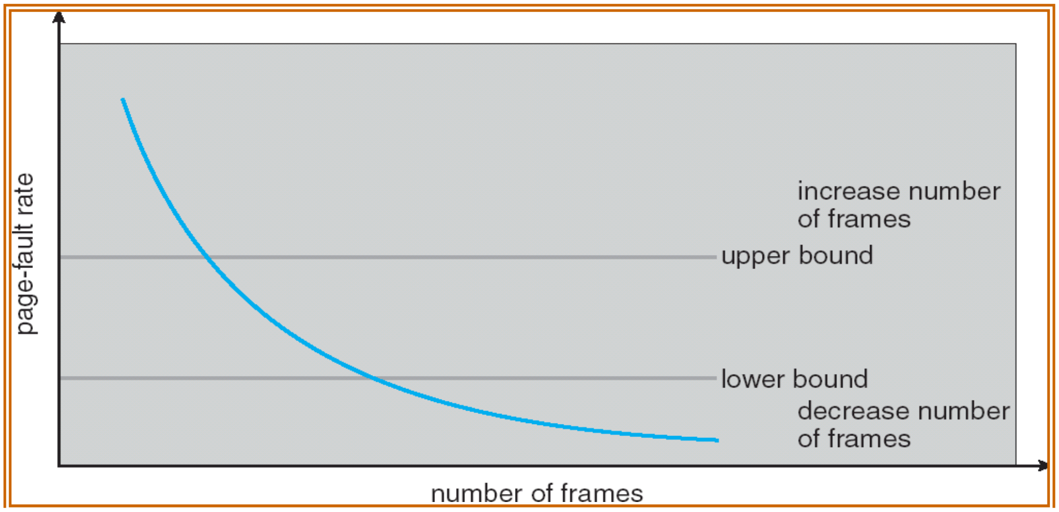
$$\mathbf{D=\sum WSS_i}$$

- ✓ Where **D** is the total demand for frames. Each process is actively using the pages in its working set. Thus, process i needs **WSS_i** frames. If the total demand is greater than the total number of available frames **(D > m)**, thrashing will occur, because some processes will not have enough frames.
- ✓ Working set **prevents** thrashing by keeping the degree of multiprogramming as high as possible. Thus it optimizes the CPU utilization.
- ✓ The main **disadvantage** of this model is keeping track of the working set. The working-set window is a **moving window**. At each memory reference, a new reference appears at one end and the oldest reference drops off the other end. A page is in the working set if it is referenced anywhere in the working-set window.



2. ***Page-Fault Frequency:***

- ✓ The working-set model is a **clumsy** way to control thrashing. A strategy that uses **Page Fault Frequency (PFF)** takes a more **direct approach**.
- ✓ When page fault rate is too high, we know that the process needs more frames. Conversely, if the page-fault rate is too low, then the process may have too many frames.
- ✓ We can establish **upper and lower bounds** on the desired page-fault rate as shown in below **figure 9.19**. If the actual page-fault rate exceeds the upper limit, we allocate the process another frame; if the page-fault rate falls below the lower limit, we remove a frame from the process. Thus, we can directly measure and control the page-fault rate to prevent thrashing.
- ✓ As with the working-set strategy, we may have to **suspend a process.** If the page-fault rate increases and no free frames are available, we must select some process and suspend it. The freed frames are then distributed to processes with high page-fault rates.



QUESTION BANK

1. For page reference string : 1,2,3,4,2,1,5,6,2,1,2,3,7,6,3,2,1,2,3,6, how many page faults would occur for LRU and optimal alg. Assuming 2 and 6 frames.

2. What is the cause of thrashing? How does system detect thrashing?

3. With diagram, discuss steps involved in handling a page fault.

4. Prove that Belady's anomaly exists for the following reference string 1,2,3,4,1,2,5,1,2,3,4,5 using FIFO algorithm when numbers of frames used are 3 and 4.

5. For the following page reference calculate the page faults that occur using FIFO and LRU for 3 and 4 page frames respectively, 5,4,2,1,4,3,5,4,3,2,1,5.

6. How to satisfy a request of size n from a list of free holes? Explain.

7. Write a note on virtual memory.

8. Explain demand paging

9. Write a note on thrashing. Explain recovery techniques from thrashing(working set model & page fault frequency)

10. What is Be lady's anomaly? Describe the working set model.

11. Explain the need of page replacement algorithms.

12. What is frame allocation policy?