

OPERATING SYSTEM

Study Notes for Engineering Students

Chapter 5 : File System and Memory Management

Unit	Topic
5.1	File: concept, attributes, operations, types, structure
5.2	Access methods: sequential, direct
5.3	Swapping
5.4	Allocation methods: contiguous, linked, indexed
5.5	Directory structure: single level, two level, tree structure
5.6	Protection: types of accesses, access control
5.7	Basic memory management: partitioning, fixed and variable
5.8	Free space management techniques: bitmap, linked list
5.9	Virtual memory: concept, paging, page fault, page table
5.10	Page replacement algorithms: FIFO, Optimal, LRU, NRU
	15 Multiple Choice Questions with Answer Key
	10 Broad Questions

www.polynoteshub.co.in

5.1 File

The computer stores programs and data permanently on secondary storage such as magnetic disks, SSDs and tapes. The operating system hides the physical properties of these devices and gives the user a uniform, logical view of storage: the **file**. The part of the operating system that deals with files, directories, allocation of disk space and protection is called the **file system**.

5.1.1 Concept of a File

A **file** is a named collection of related information that is recorded on secondary storage. From the user's point of view a file is the smallest unit of logical secondary storage: data cannot be written to secondary storage unless they are inside a file. Files may be of many kinds: source programs, object programs, executable programs, numeric data, text, payroll records, images, sound and video. The contents of a file are defined by its creator and have a meaning that depends on its type. For example, a text file is a sequence of characters organized into lines (and possibly pages), while an executable file is a series of code sections that the loader can bring into memory and run.

5.1.2 File Attributes

A file is given a name for the convenience of human users, and it is referred to by that name. In addition to the name, a file has other attributes (information stored about the file), which vary from one operating system to another:

Attribute	Meaning
Name	The symbolic file name; the only information kept in human-readable form.
Identifier	A unique tag (usually a number) that identifies the file within the file system; not readable by humans (for example the inode number in UNIX).
Type	Needed by systems that support different kinds of files (text, executable, image ...).
Location	A pointer to the device and to the position of the file on that device.
Size	The current size of the file (in bytes, words or blocks) and possibly the maximum allowed size.
Protection	Access-control information: who can read, write or execute the file.
Time, date and user identification	Creation time, last modification time and last use time, and the owner. Useful for protection, security and usage monitoring.

This information is kept in a **directory structure** that also resides on secondary storage. A directory entry consists of the file's name and its unique identifier; the identifier in turn locates the other attributes. In a large system directory entries may take more than a kilobyte per file.

5.1.3 File Operations

A file is an abstract data type. To define it properly the operating system provides system calls to perform the following basic operations:

Operation	What the operating system does
Create	Finds space in the file system for the file, then makes a new entry in the directory.
Write	A system call specifies the file name and the data to write. The system keeps a write pointer to the location of the next write and updates it after every write.
Read	A system call specifies the file name and where in memory the next block should be put. A read pointer (the current-file-position pointer) is kept and updated.
Reposition (seek)	Changes the current-file-position pointer to a given value. No actual I/O takes place.

Operation	What the operating system does
Delete	Searches the directory for the file, releases all its space so that it can be reused, and erases the directory entry.
Truncate	Erases the contents of the file but keeps its attributes; the file length is reset to zero and its space is released.

Other common operations are **append** (add data at the end), **rename**, and **copy**. To avoid searching the directory for every operation, most systems require an **open()** call before a file is used, and a **close()** call after use. The operating system maintains two tables for this purpose:

- **System-wide open-file table:** contains a copy of the file control block of each open file, the file-open count and other process-independent data such as the file location on disk and the size.
- **Per-process open-file table:** contains a pointer to the corresponding entry in the system-wide table, the current file position and the access rights for that process.

Information associated with an open file includes the **file pointer**, the **file-open count** (the number of processes that have the file open; the entry is removed when the count falls to zero), the **disk location** and the **access rights**.

5.1.4 File Types

If an operating system recognizes the type of a file it can operate on it in reasonable ways. A common technique for implementing file types is to include the type as part of the file name: the name is split into two parts, a **name** and an **extension**, usually separated by a period.

File type	Usual extension	Function
Executable	exe, com, bin	Machine-language program ready to run
Object	obj, o	Compiled machine language, not linked
Source code	c, cpp, java, py, asm	Source code in various languages
Batch / script	bat, sh	Commands for the command interpreter
Text / document	txt, doc, docx	Textual data and documents
Word processor	rtf, wp	Various word-processor formats
Library	lib, a, dll	Libraries of routines for programmers
Print or view	pdf, jpg, gif	ASCII or binary file for printing or viewing
Archive	zip, tar, rar	Related files grouped into one compressed file
Multimedia	mp3, mp4, avi	Binary file with audio or video information

In UNIX the extension is only a hint for the user and applications; the system stores a **magic number** at the beginning of some files to indicate the type. Mac OS stores the **creator** attribute together with the type.

5.1.5 File Structure

File types also decide the **internal structure** of a file. The structure of a file may be viewed at three levels:

- **No structure (stream of bytes or words):** the operating system treats a file as an unstructured sequence of bytes. It gives the maximum flexibility, because the application program imposes its own structure. UNIX and Windows use this model.
- **Simple record structure:** the file is a sequence of records, either of fixed length or of variable length. Lines of text are an example.

- **Complex structures:** a formatted document or a relocatable load file; also trees of records, each with a key, as used in **indexed sequential** files.

Internally, locating an offset within a file can be complicated for the operating system. Disk systems read and write in **blocks** (physical records) of a fixed size, typically 512 bytes to 4 KB, while logical records may have any length. Logical records are therefore **packed into physical blocks**. As a result, some portion of the last block of each file is generally wasted; this wastage is **internal fragmentation**, and on average it is half a block per file. Bigger blocks mean greater internal fragmentation.

Poly Notes Hub

5.2 Access Methods

Files store information. When a file is used, this information must be accessed and read into computer memory. The information in a file can be accessed in several ways; the two basic methods are sequential access and direct access.

5.2.1 Sequential Access

The simplest access method. Information in the file is processed **in order, one record after another**. It is based on a tape model of a file and works equally well on sequential-access devices and on random-access devices. A read operation, **read next**, reads the next portion of the file and automatically advances a file pointer. Similarly **write next** appends to the end of the file and advances to the end of the newly written material. A file can be **reset** (rewound) to the beginning, and on some systems a program may skip forward or backward n records.

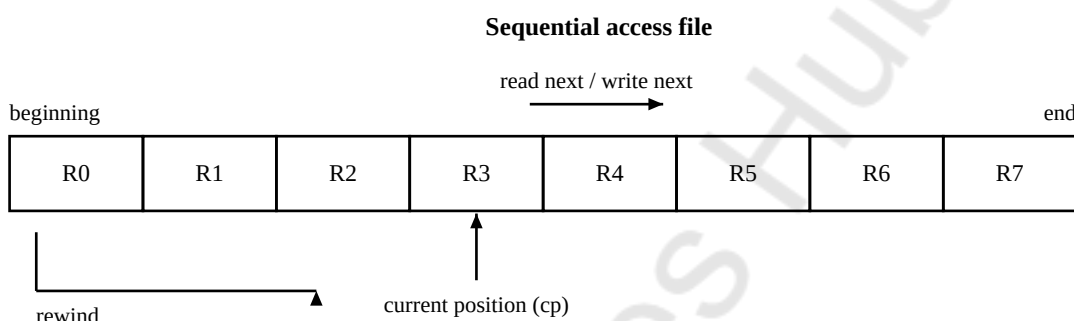


Fig. 5.1 : Sequential access file

Operations: **read next** (read cp ; $cp = cp + 1$), **write next** (write cp ; $cp = cp + 1$), **reset** ($cp = 0$). Sequential access is used by editors and compilers.

5.2.2 Direct Access (Relative Access)

A file is made up of **fixed-length logical records** that allow programs to read and write records rapidly **in no particular order**. This method is based on a **disk model** of a file, since a disk allows random access to any file block. The file is viewed as a numbered sequence of blocks or records; we may read block 14, then block 53, then write block 7. There is no restriction on the order of reading or writing. Direct-access files are of great use for immediate access to large amounts of information, such as databases: when a query arrives, the block that holds the answer is computed and read directly.

The file operations are modified to include the block number n as a parameter: **read n** and **write n** (instead of read next and write next), and **position to n** . The number n is a **relative block number**, an index relative to the beginning of the file (the first relative block is 0). This helps the operating system to decide where the file should be placed and prevents the user from accessing portions of the file system that may not be part of the file. Sequential access can be simulated on a direct-access file by keeping a variable cp (read next: read cp ; $cp = cp + 1$), but the reverse is very inefficient.

Indexed sequential access

Other access methods are built on top of direct access. An **index** to the file, like the index at the back of a book, contains pointers to the various blocks. To find a record, the index is searched first and the pointer is used to access the file directly. For very large files the index itself may be too large to keep in memory, so a **multilevel index** (an index of the index) is used. This is the **indexed sequential access method (ISAM)**.

Feature	Sequential access	Direct access
Order of access	Records processed one after another, in order	Any record can be read or written in any order
Record length	Fixed or variable	Fixed-length records (or blocks)
Operations	read next, write next, reset	read n, write n, position to n
Speed	Slow to reach a distant record (all earlier records must be passed)	Fast; the position of a record is calculated directly
Typical use	Editors, compilers, tape backup, log files	Databases, reservation systems
Device	Tape or disk	Disk (random-access device)

5.3 Swapping

A process must be in main memory to be executed. A process can, however, be **swapped temporarily out of memory to a backing store** and then brought back into memory for continued execution. Swapping makes it possible for the total physical address space of all processes to exceed the real physical memory of the system, and thus increases the degree of multiprogramming.

Standard swapping

Consider a multiprogramming environment with Round Robin scheduling. When a quantum expires, the memory manager starts to **swap out** the process that has just finished and to **swap in** another process into the memory space that has been freed. In the meantime the CPU scheduler allocates a time slice to another process in memory. In a priority-based system, if a higher-priority process arrives and wants service, the memory manager can swap out a lower-priority process and load the higher-priority one; when the higher-priority process finishes, the lower-priority process can be swapped back in and continue. This variant is called **roll out, roll in**.

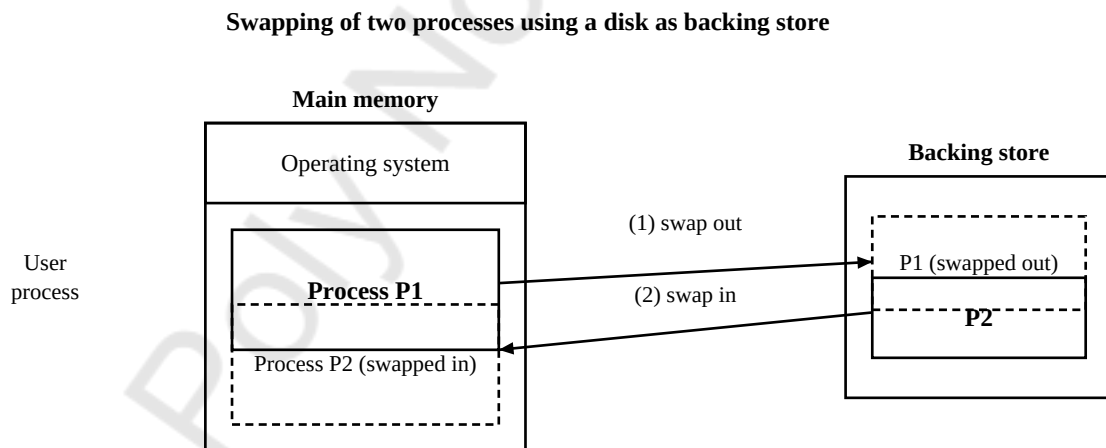


Fig. 5.2 : Swapping of two processes using a disk as a backing store

Important points

- **Backing store:** a fast disk, large enough to hold copies of all memory images for all users, and with direct access to these images.
- **Ready queue:** the system maintains a ready queue of ready-to-run processes which have memory images on disk. The dispatcher checks whether the next process is in memory; if not, and there is no free memory region, it swaps out a process and swaps in the desired one, then reloads the registers and transfers control.

- **Binding:** if address binding is done at assembly or load time the process must be swapped back into the same memory location. With execution-time binding it can be swapped into a different location.
- **Pending I/O:** a process to be swapped out must be completely idle. If it is waiting for I/O into a buffer in user memory, it cannot be swapped (or the I/O must be done only into operating-system buffers).
- **Context-switch time is high.** The major part of the swap time is the **transfer time**, which is directly proportional to the amount of memory swapped.

Example. A user process is 100 MB and the backing store is a hard disk with a transfer rate of 50 MB/s and an average latency of 8 ms.

Transfer time = $100 \text{ MB} / 50 \text{ MB/s} = 2 \text{ s} = 2000 \text{ ms}$. Swap-out time = $2000 + 8 = 2008 \text{ ms}$. Swapping a process out and another of the same size in therefore takes about **4016 ms** (a little over 4 seconds). So the CPU time slice should be much larger than this swap time to be efficient.

Modern systems (UNIX, Linux, Windows) do not use standard swapping. They swap only **portions (pages)** of processes, or enable swapping only when the free memory falls below a threshold. Mobile systems such as Android and iOS do not support swapping; they ask applications to give up memory voluntarily.

5.4 Allocation Methods

The main problem is how to allocate space to files so that disk space is utilized effectively and files can be accessed quickly. Three major methods of allocating disk space are in wide use: **contiguous**, **linked** and **indexed**. Each has advantages and disadvantages, and some systems support all three.

5.4.1 Contiguous Allocation

Each file occupies a set of **contiguous blocks** on the disk. If a file is n blocks long and starts at location b , it occupies blocks b , $b + 1$, $b + 2$, ..., $b + n - 1$. The directory entry for each file indicates the **address of the starting block** and the **length** (number of blocks) of the file.

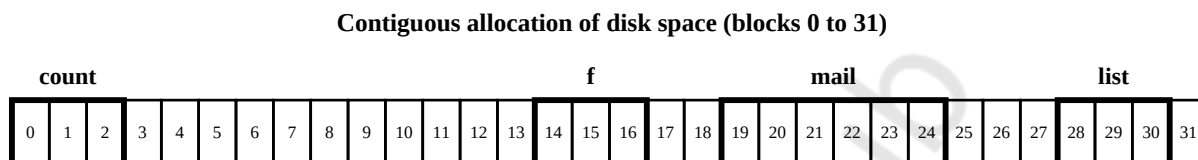


Fig. 5.3 : Contiguous allocation

File	Start	Length	Blocks occupied
count	0	3	0, 1, 2
f	14	3	14, 15, 16
mail	19	6	19 to 24
list	28	3	28, 29, 30

Advantages and disadvantages

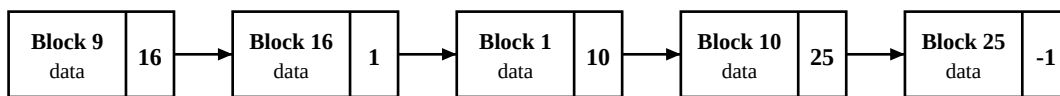
Advantages	Disadvantages
Simple: only the start location and length are needed	External fragmentation: free space is broken into small pieces; there may be enough total space but not in one piece
Supports both sequential and direct access ; block i is at $b + i$	The file size must be known when the file is created, and it is difficult for a file to grow
Fast: the number of disk seeks and head movement is minimal	Compaction (copying the whole file system to another disk to make free space contiguous) is time consuming
	Space is lost if a large amount is allocated in advance but unused (internal fragmentation)

The first-fit and best-fit strategies are used to select a free hole from the set of free holes. To reduce the problem of growing files, some systems use a **modified contiguous scheme**: a contiguous chunk of space called an **extent** is allocated initially, and if the file outgrows it another extent is added. The directory entry then contains the start address, the block count and a link to the first block of the next extent (used in the Veritas file system).

5.4.2 Linked Allocation

Linked allocation solves all the problems of contiguous allocation. Each file is a **linked list of disk blocks**, and the blocks may be **scattered anywhere on the disk**. The directory contains a pointer to the **first and last blocks** of the file. Each block contains a pointer to the next block. These pointers are not available to the user: if each block is 512 bytes and a pointer takes 4 bytes, the user sees blocks of 508 bytes.

Linked allocation : file 'jeep' (start = 9, end = 25)



Each block stores a pointer to the next block; -1 marks end of file

Fig. 5.4 : Linked allocation

To create a new file, simply create a new directory entry with a null (-1) pointer. A write causes a free block to be found and linked at the end of the file. A read follows the pointers from block to block.

Advantages	Disadvantages
No external fragmentation; every free block can be used	Can be used effectively only for sequential access ; to find the i-th block one must follow i pointers
The file can grow freely; the size need not be declared at creation	Space is needed for pointers (e.g. 4 / 512 = 0.78% of the disk)
No need to compact disk space	Reliability: if a pointer is lost or damaged the rest of the file is lost; many disk seeks needed

Clusters: the usual solution to the pointer overhead is to collect blocks into groups called **clusters** (say four blocks) and allocate clusters instead of blocks. The percentage of disk used for pointers decreases, but internal fragmentation increases. **File Allocation Table (FAT):** an important variation is the FAT used by MS-DOS and early Windows. A section of the disk at the beginning of each volume holds a table with **one entry for each disk block**, indexed by block number. The directory holds the first block number of the file; the table entry for that block holds the next block number, and the last block has a special end-of-file value. An unused block has the value 0. The table can be cached, so random access is much faster than plain linked allocation, but the disk head must move to read the FAT.

5.4.3 Indexed Allocation

Linked allocation cannot support efficient direct access because the pointers are scattered with the blocks. **Indexed allocation** solves this by bringing all the pointers together into one location: the **index block**. Each file has its own index block, which is an array of disk-block addresses. The i-th entry of the index block points to the i-th block of the file. The directory contains the address of the index block.

Indexed allocation : index block 19

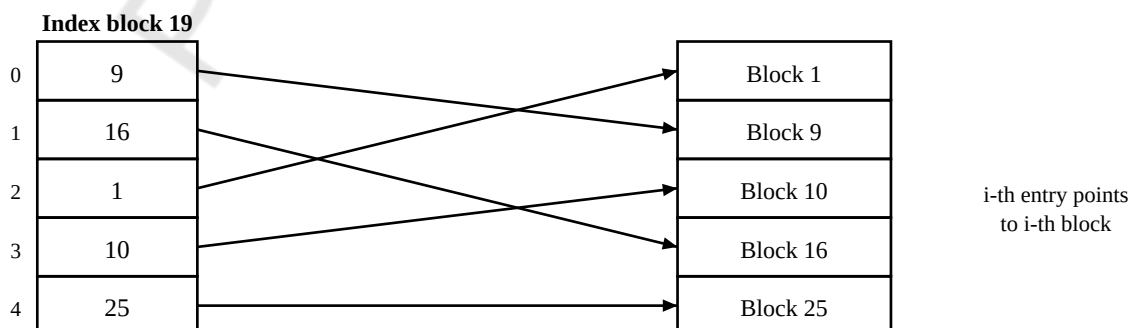


Fig. 5.5 : Indexed allocation of a file

When the file is created all pointers in the index block are set to nil. When the i-th block is first written, a block is taken from the free-space manager and its address is put in the i-th index entry.

Advantages	Disadvantages
Supports direct (random) access efficiently	Index block is overhead: even a one- or two-block file needs a whole index block
No external fragmentation ; any free block can be used	Pointer overhead is generally greater than in linked allocation
File can grow as long as free blocks are available	A large file may need more than one index block

Handling large files: the size of the index block

The index block size is a trade-off between wasted space and the maximum file size. Mechanisms for a file that needs more entries than one index block can hold:

- **Linked scheme:** several index blocks are linked together; the last entry of an index block is either nil or a pointer to the next index block.
- **Multilevel index:** a first-level index block points to a set of second-level index blocks, which point to the file blocks.
- **Combined scheme (UNIX inode):** the inode holds 15 pointers. The first 12 are **direct blocks**; the 13th is a **single indirect** block, the 14th a **double indirect** block and the 15th a **triple indirect** block.

Example. Block size = 4 KB and a pointer = 4 bytes. One index block holds $4096 / 4 = 1024$ pointers. A single index block therefore addresses $1024 \times 4 \text{ KB} = \mathbf{4 \text{ MB}}$ of file. With a two-level index the maximum file size is $1024 \times 1024 \times 4 \text{ KB} = \mathbf{4 \text{ GB}}$.

5.4.4 Comparison of the Allocation Methods

Basis	Contiguous	Linked	Indexed
Access supported	Sequential and direct	Mainly sequential	Sequential and direct
External fragmentation	Yes	No	No
File growth	Difficult	Easy	Easy (until the index is full)
Size known in advance	Yes	No	No
Pointer overhead	None	One pointer in every block	One index block per file
Reliability	Good	Poor (one lost pointer)	Good, but the index block is critical
Speed	Fastest (few seeks)	Slow (many seeks)	Good; extra access to the index block

5.5 Directory Structure

A computer can store millions of files. To organize them, the file system is divided into **partitions (volumes)**, and each partition contains information about the files within it in a **directory**. A directory is a symbol table that translates file names into their directory entries.

Operations on a directory

- **Search** for a file; **create** a file; **delete** a file;
- **List** a directory (list the files and their entries); **rename** a file;
- **Traverse** the file system (access every directory and every file within a directory).

5.5.1 Single-Level Directory

The simplest structure: **all files are contained in the same directory**. It is easy to support and understand. It has significant limitations when the number of files increases or when the system has more than one user.

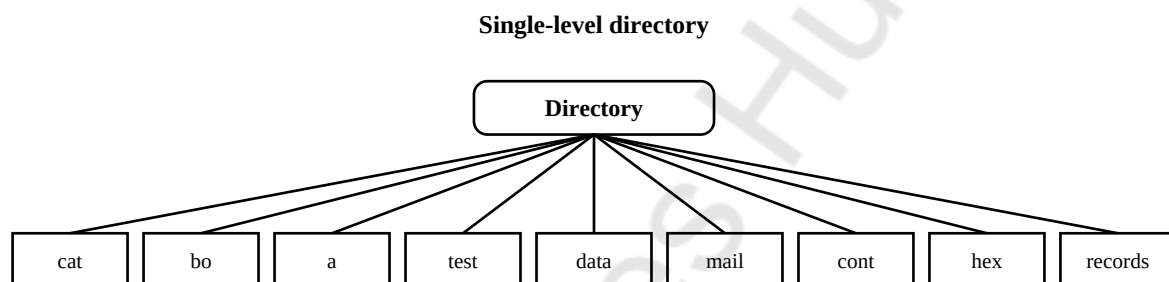


Fig. 5.6 : Single-level directory

- **Naming problem:** all files must have **unique names**. If two users call their data file test, the unique-name rule is violated. Even a single user finds it hard to remember the names of hundreds of files.
- **Grouping problem:** there is no way to group files by type or by purpose.
- Searching is slow for a large directory.

5.5.2 Two-Level Directory

The standard solution to the naming problem is to create a **separate directory for each user**. In the two-level structure each user has his own **User File Directory (UFD)**, which lists only the files of that user. When a user job starts or a user logs in, the system's **Master File Directory (MFD)** is searched; it is indexed by user name or account number, and each entry points to the UFD of that user.

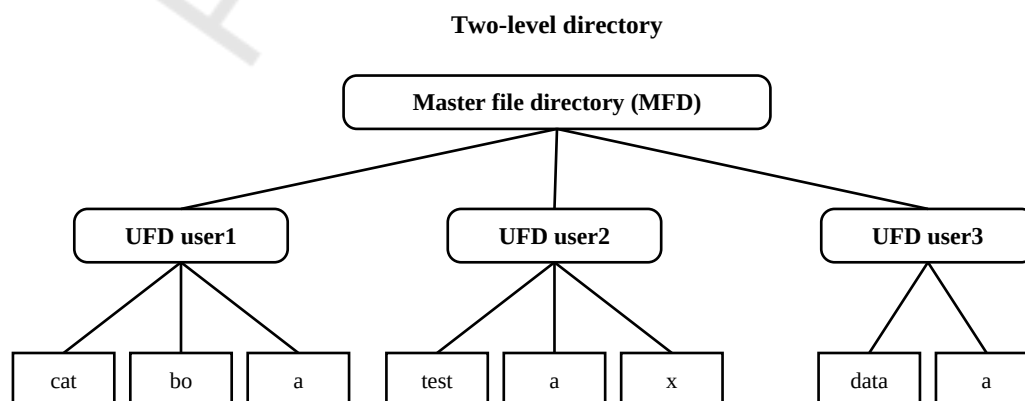
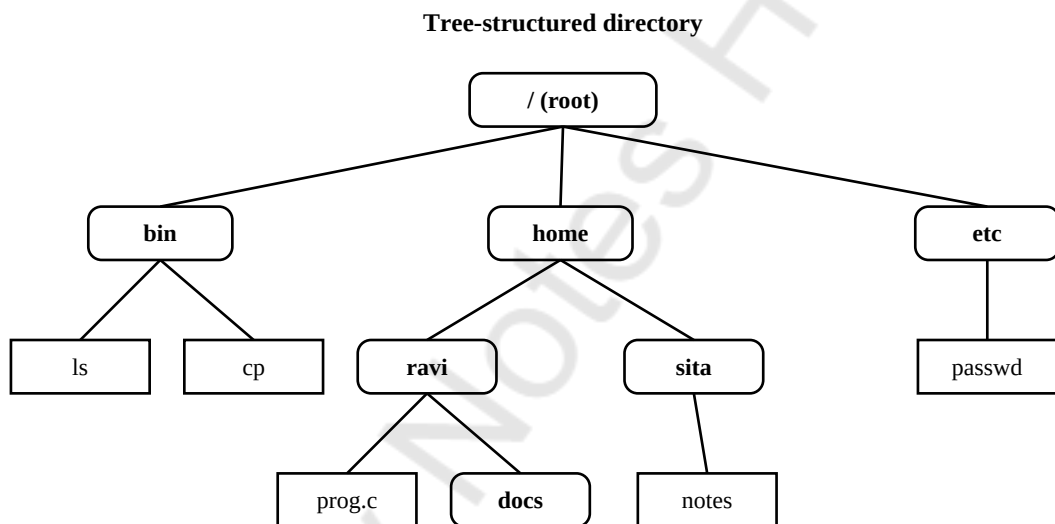


Fig. 5.7 : Two-level directory

- Different users can have files with the **same name**, as long as the names within each UFD are unique. The file is identified by a **path name** such as /user1/cat (user name + file name).
- Efficient searching and isolation of users from each other, which is good for protection but a disadvantage when users want to **cooperate and share files**.
- System files (e.g. compilers and loaders) are placed in a special user directory, which is searched when a command is not found in the user's UFD (the **search path**).
- It still gives **no grouping capability** inside a user's own files.

5.5.3 Tree-Structured Directory

The two-level structure is generalized to a tree of arbitrary height. This allows users to create their own **subdirectories** and organize files accordingly. The tree has a **root directory** and every file in the system has a **unique path name** (the path from the root through the subdirectories to the file). A directory (or subdirectory) contains a set of files or subdirectories; internally all directories have the same format, and one bit in each directory entry defines the entry as a file (0) or a subdirectory (1). Special system calls are used to create and delete directories.



Rounded box = directory, plain box = file. Path: /home/ravi/prog.c

Fig. 5.8 : Tree-structured directory

- Each process has a **current directory** (working directory). If a file is not in the current directory the user must give a path name or change the current directory.
- **Absolute path name:** begins at the root and follows a path to the specified file, e.g. /home/ravi/prog.c.
- **Relative path name:** defines a path from the current directory, e.g. if the current directory is /home/ravi then prog.c or docs/notes refers to a file in it.
- **Deleting a directory:** if the directory is empty it can simply be deleted. If it is not empty, either the user must first delete all its files (as in MS-DOS) or the delete command may recursively remove all its files and subdirectories (as in UNIX rm -r).
- Users can access files of other users by specifying path names, so sharing is possible. The structure is the most common in use (UNIX, Linux, Windows).

Feature	Single level	Two level	Tree structure
Structure	One directory for all files	MFD with one UFD per user	Root with subdirectories of any depth

Feature	Single level	Two level	Tree structure
Naming	All names must be unique	Unique within each user's UFD	Unique path name for each file
Grouping	None	Only by user	Full; subdirectories for any group
Sharing	Possible, but no protection	Difficult (isolation)	Possible with path names
Searching	Slow for many files	Efficient	Efficient, using paths

5.6 Protection

When information is stored in a computer system, we want to keep it safe from physical damage (**reliability**) and from improper access (**protection**). Reliability is generally provided by duplicate copies (backups) of files. Protection can be provided in many ways; in a single-user system it may be physically removing the disks and locking them away, but in a multi-user system other mechanisms are needed.

5.6.1 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:

Type of access	Meaning
Read	Read from the file.
Write	Write or rewrite the file.
Execute	Load the file into memory and execute it.
Append	Write new information at the end of the file.
Delete	Delete the file and free its space for possible reuse.
List	List the name and attributes of the file.

Other operations such as renaming, copying and editing a file may also be controlled. Higher-level functions can be built on the lower-level ones: for example copying a file may be implemented as a sequence of read requests, so a user with read access can also copy, print and so on.

5.6.2 Access Control

The most common approach to the protection problem is to make access **dependent on the identity of the user**. Different users may need different types of access to a file or directory. The most general scheme is to associate with each file and directory an **Access-Control List (ACL)** specifying the user names and the types of access allowed for each user. When a user requests access to a file, the operating system checks the access list; if the user is listed for the requested access, the access is allowed, otherwise a protection violation occurs.

The problem with ACLs is that the list may be **long** (listing every user is tedious, and the directory entry becomes variable-sized). To condense the list, many systems classify users into **three classes**:

- **Owner:** the user who created the file.
- **Group:** a set of users who are sharing the file and need similar access.
- **Universe (others):** all other users in the system.

UNIX uses three fields of three bits each: **r** (read), **w** (write) and **x** (execute), one field for each of owner, group and universe. A file therefore needs only **9 bits** for protection, shown as **rw-rw-rw-**. Each bit can be written as a number ($r = 4$, $w = 2$, $x = 1$), so permission 761 means:

Class	Octal digit	Binary	Permission	Meaning
Owner	7	111	rwX	read, write and execute
Group	6	110	rw-	read and write
Universe	1	001	--X	execute only

The command `chmod 761 game` gives these permissions to the file `game`. Other protection approaches: a **password** associated with each file (the user must remember many passwords, and one password gives all-or-none access), and the use of **disk-block passwords**. The ACL approach is more general and widely used.

Poly Notes Hub

5.7 Basic Memory Management

Memory is a large array of words or bytes, each with its own address. The CPU fetches instructions from memory according to the value of the program counter. Since main memory is limited, the operating system's **memory manager** must keep track of which parts of memory are in use and which are free, allocate memory to processes when they need it, and deallocate it when they are done.

Background: logical and physical addresses

- A **logical address** (virtual address) is generated by the CPU. A **physical address** is the address seen by the memory unit. The set of all logical addresses is the **logical address space** and the set of physical addresses is the **physical address space**.
- The run-time mapping from virtual to physical addresses is done by a hardware device, the **Memory Management Unit (MMU)**. In the simplest scheme, a **relocation (base) register** value is added to every address generated by the user process.
- **Protection:** a **limit register** holds the range of logical addresses. Every address generated by the CPU is checked against the limit; if it is outside the range, a trap to the operating system occurs. This protects the operating system and other processes from the running process.

5.7.1 Partitioning

One of the simplest methods of memory allocation is to divide memory into several **partitions**. Each partition may contain exactly one process, so the degree of multiprogramming is bound by the number of partitions. Memory is usually divided into two parts: one for the resident operating system (usually in low memory with the interrupt vector) and one for the user processes. There are two schemes of partitioning: fixed and variable.

5.7.2 Fixed Partitioning (Static)

Main memory is divided into a number of **fixed-size partitions** at system start-up. The partitions may be of **equal size or of unequal sizes**. Any process whose size is less than or equal to the partition size can be loaded into a partition. When a partition is free, a process is selected from the input queue and loaded into it; when the process terminates, the partition becomes available for another process. This was the technique used by IBM OS/360 (MFT, Multiprogramming with a Fixed number of Tasks).

- **Internal fragmentation:** a process rarely fills its partition exactly, so memory inside a partition is wasted. For example a 6 MB process in an 8 MB partition wastes 2 MB.
- The **degree of multiprogramming is limited** by the number of partitions.
- A process larger than the largest partition cannot be run unless overlays are used.
- Simple to implement and little operating-system overhead.

5.7.3 Variable Partitioning (Dynamic)

The partitions are of **variable length and number**. When a process arrives it is allocated exactly as much memory as it needs and no more. The operating system keeps a table indicating which parts of memory are available and which are occupied. Initially all memory is available for user processes and is one large **block (hole)** of available memory. As processes enter and leave, memory contains a set of holes of various sizes scattered through memory. This was used by IBM OS/360 MVT (Multiprogramming with a Variable number of Tasks).

Fixed and variable partitioning of memory

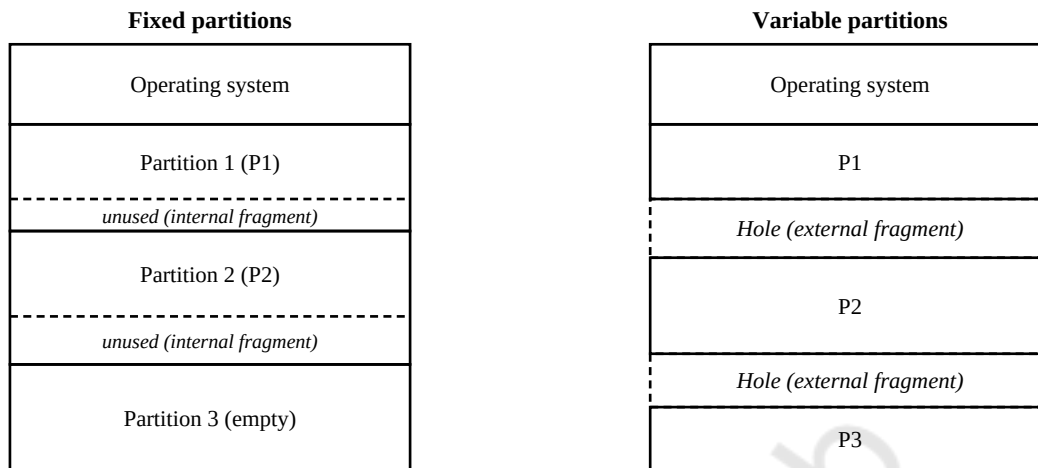


Fig. 5.9 : Fixed and variable partitioning

Dynamic storage-allocation strategies

When a process arrives and needs memory, the system searches the set of holes for one that is large enough. If the hole is too large it is split: one part is allocated to the process and the other is returned to the set of holes. Adjacent holes are merged. The following strategies are used to choose a hole:

Strategy	Description	Remark
First fit	Allocate the first hole that is big enough . The search may start at the beginning or where the previous search ended.	Fast
Best fit	Allocate the smallest hole that is big enough . The entire list must be searched unless it is ordered by size.	Leaves the smallest leftover hole, but produces tiny unusable holes
Worst fit	Allocate the largest hole . The entire list must be searched. The leftover hole is the largest.	Generally the worst performer

Example. Memory partitions (holes) in order: 100 KB, 500 KB, 200 KB, 300 KB and 600 KB. Processes of 212 KB, 417 KB, 112 KB and 426 KB arrive in this order.

First fit: 212 goes to 500 (leaves 288); 417 goes to 600 (leaves 183); 112 goes to 288 (leaves 176); **426 must wait** (no hole is large enough).

Best fit: 212 goes to 300; 417 goes to 500; 112 goes to 200; 426 goes to 600. **All four processes are placed.**

Worst fit: 212 goes to 600 (leaves 388); 417 goes to 500; 112 goes to 388 (leaves 276); **426 must wait.**

In this example best fit makes the most efficient use of memory.

Fragmentation

Basis	Internal fragmentation	External fragmentation
Meaning	Unused memory inside an allocated partition or block	Free memory exists, but in many small non-contiguous holes
Occurs in	Fixed partitioning, paging (last page), block-based file allocation	Variable partitioning, contiguous file allocation, segmentation
Remedy	Use smaller or variable-size units	Compaction (shuffle memory contents to place all free memory together; possible only with execution-time binding) or non-contiguous allocation (paging)

Statistical analysis of first fit shows that, given N allocated blocks, another $0.5 N$ blocks are lost to fragmentation: one-third of memory may be unusable. This is the **50-percent rule**.

5.8 Free Space Management Techniques

Since disk space is limited, the space of deleted files must be reused. To keep track of free disk space, the system maintains a **free-space list** that records all free disk blocks (those not allocated to any file or directory). To create a file the free-space list is searched for the required amount of space, and that space is allocated to the new file and removed from the list. When a file is deleted its space is added to the free-space list. The same techniques are applied to free memory.

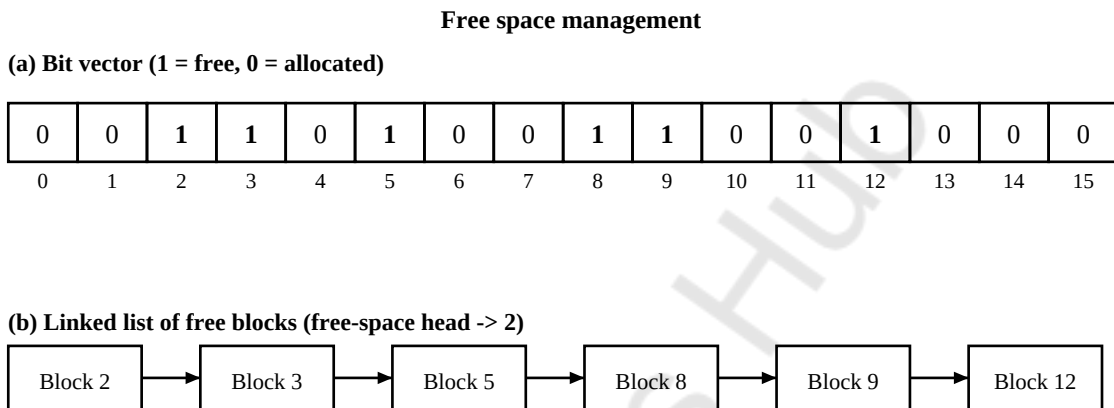


Fig. 5.10 : Bit vector and linked list of free blocks

5.8.1 Bitmap (Bit Vector)

The free-space list is implemented as a **bit map or bit vector**. 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** (some systems use the opposite convention). For example, consider a disk of 16 blocks where blocks 2, 3, 5, 8, 9 and 12 are free and the rest are allocated. The free-space bit map is **0011010011001000** (Fig. 5.10 a).

Advantages and disadvantages

Advantages	Disadvantages
Relatively simple and efficient to find the first free block or n consecutive free blocks on the disk	The bit map must be kept in main memory for efficiency; it is not efficient unless the entire vector is in memory
Many computers have instructions that return the offset of the first 1 bit in a word, so searching is fast	Needs extra space: a 1 TB disk with 4 KB blocks needs 2^{28} bits = 32 MB of bit map
Easy to get contiguous space for a file	For large disks the bit map may not fit in memory and must be written to disk periodically

Finding the first free block. Scan the words of the bit map for the first word that is not 0 (a word with all 0 bits has no free block). The first free block number = (number of bits per word) x (number of 0-value words) + (offset of the first 1 bit). For example, with 8-bit words and the bit map 00000000 00110100 ..., the first word is all zeros, and the first 1 bit in the next word is at offset 2, so the first free block is $8 \times 1 + 2 = 10$.

5.8.2 Linked List

Another approach 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). The first free block contains a pointer to the next free block, and so on. In the example above, the system would keep a pointer to block 2 as the first free block; block 2 would contain a pointer to block 3, which points to block 5, which points to block 8, and so on (Fig. 5.10 b).

Advantages	Disadvantages
No waste of space except the pointer in each free block	Not efficient to traverse the list: reading each block needs substantial I/O time
Only a pointer to the first free block needs to be kept in memory	Getting a contiguous area of free space is difficult
Allocating a single block is fast: take the first block of the list	A damaged pointer breaks the chain

Variations of the linked list

- **Grouping:** the first free block stores the addresses of n free blocks. The first $n - 1$ of these are actually free, and the last one contains the addresses of another n free blocks. A large number of free blocks can be found quickly.
- **Counting:** generally several contiguous blocks are allocated or freed together. Each entry in the free-space list stores the **address of the first free block and the count n** of free contiguous blocks that follow.
- **Space maps (ZFS):** a log-structured list of free-space changes, kept per metaslab, to reduce the I/O for very large file systems.

Basis	Bit map	Linked list
Information kept	One bit per block	A pointer inside each free block
Finding n contiguous blocks	Easy	Difficult
Memory needed	Whole vector for speed (can be large)	Only the head pointer
Space overhead	Fixed; proportional to the disk size	None for the list, except pointers inside free blocks
Speed of search	Fast with hardware instructions	Slow; the list must be traversed

5.9 Virtual Memory

5.9.1 Concept

The instructions being executed must be in physical memory, but this limits the size of a program to the size of physical memory. In many cases the entire program is not needed: error-handling code is rarely run, arrays and tables are often allocated more memory than they use, and some options and features of a program are rarely used. **Virtual memory** is a technique that allows the **execution of processes that are not completely in memory**. It separates logical memory as perceived by the user from physical memory.

Benefits

- A program can be **larger than the physical memory**; the programmer sees a very large virtual address space.
- **More programs** can run at the same time, as each program takes less physical memory; this increases CPU utilization and throughput.
- **Less I/O** is needed to load or swap each user program into memory, so programs run faster.
- Processes can **share** libraries and memory easily; pages can be shared between processes (for example during `fork()`).

Virtual memory is commonly implemented by **demand paging**: a page is brought into memory only when it is needed. A **lazy swapper** (pager) never swaps a page into memory unless the page will be needed.

5.9.2 Paging

Paging is a memory-management scheme that permits the physical address space of a process to be **non-contiguous**, so it avoids external fragmentation and the need for compaction. Physical memory is divided into fixed-size blocks called **frames**, and logical memory is divided into blocks of the same size called **pages**. When a process is to be executed, its pages are loaded into any available frames from the backing store, which is itself divided into blocks of the same size. The page size is defined by the hardware and is a power of 2, typically between 512 bytes and 16 MB (commonly 4 KB).

Address translation

Every address generated by the CPU is divided into two parts: a **page number (p)** and a **page offset (d)**. The page number is used as an index into a **page table**, which contains the base address (frame number *f*) of each page in physical memory. This base address is combined with the page offset to define the physical address that is sent to the memory unit.

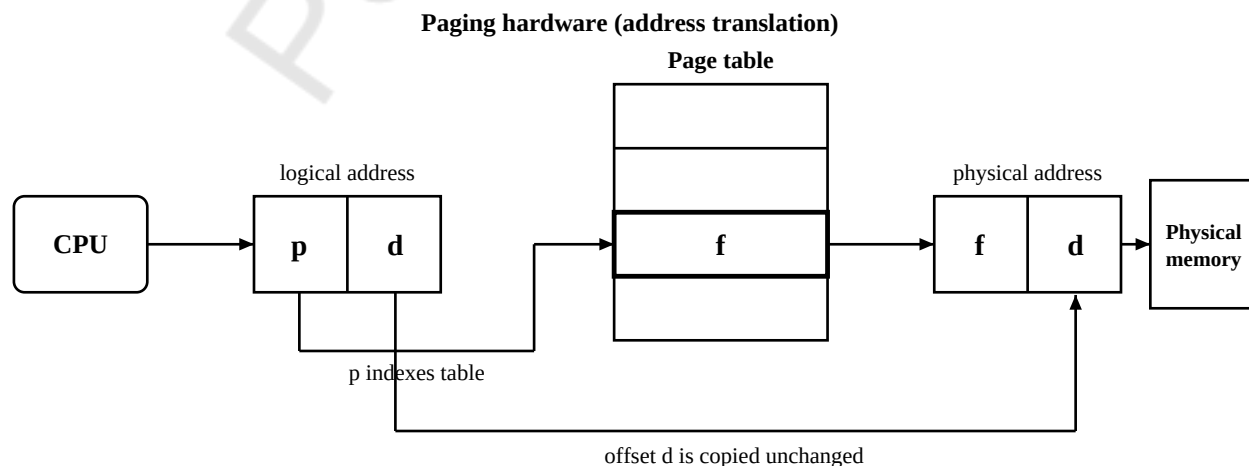


Fig. 5.11 : Paging hardware

- Page number $p = \text{logical address} \div \text{page size}$; offset $d = \text{logical address} \bmod \text{page size}$.
- If the logical address space is 2^m and the page size is 2^n , the high-order $(m - n)$ bits give the page number and the low-order n bits give the offset.
- Physical address = (frame number $\times$ page size) + offset.

Mapping of logical pages to physical frames through the page table

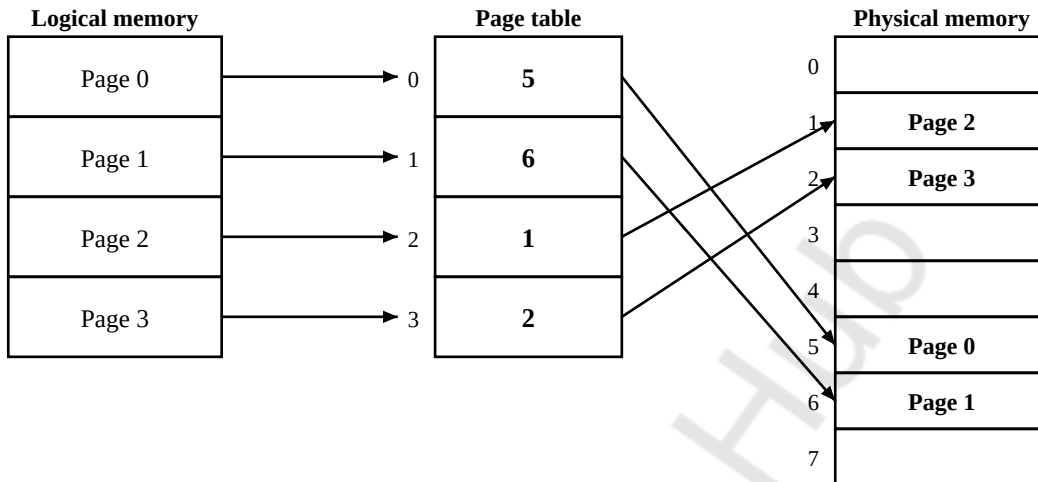


Fig. 5.12 : Paging model of logical and physical memory

Example 1. Page size = 4 bytes; the page table of Fig. 5.12 maps page 0 to frame 5, page 1 to frame 6, page 2 to frame 1 and page 3 to frame 2. For logical address 13: page number = $13 \div 4 = 3$, offset = $13 \bmod 4 = 1$. Page 3 is in frame 2, so the physical address = $2 \times 4 + 1 = 9$.

Example 2. Page size = 1 KB (1024 bytes), logical address 3000: page number = $3000 \div 1024 = 2$, offset = $3000 - 2048 = 952$. If page 2 is in frame 5, the physical address = $5 \times 1024 + 952 = 6072$.

Internal fragmentation in paging: paging has no external fragmentation, but there may be internal fragmentation in the **last page** of a process. If the process size is not a multiple of the page size, the last frame is not completely full. On average, half a page per process is wasted, so small pages reduce this waste but increase the size of the page table.

5.9.3 Page Table

The page table is a per-process data structure kept in main memory. The operating system keeps a copy of it for each process (in the PCB). A **page-table base register (PTBR)** points to the page table. Because every data or instruction access then needs two memory accesses (one for the page-table entry and one for the data), a small, very fast hardware cache called the **Translation Look-aside Buffer (TLB)** is used. The TLB stores a few page-number and frame-number pairs; if the page number is found in the TLB (a **TLB hit**) the frame number is available immediately, otherwise (a **TLB miss**) the page table in memory is consulted.

Each page-table entry (PTE) contains:

Field	Purpose
Frame number	The physical frame in which the page is stored.
Valid / present bit	1 means the page is legal and in memory ; 0 means the page is not in memory (or not in the process's address space). Access to a page with bit 0 causes a page fault .
Protection bits	Read, write, execute permissions for the page.

Field	Purpose
Referenced bit (R)	Set by hardware whenever the page is read or written. Used by page-replacement algorithms.
Modified / dirty bit (M)	Set by hardware when the page is written. A modified page must be written back to disk when replaced; an unmodified page need not be.

For large address spaces the page table itself becomes very large (for a 32-bit address space and 4 KB pages there are 2^{20} entries), so it is organized as a **hierarchical (multilevel) page table**, a **hashed page table**, or an **inverted page table**.

5.9.4 Page Fault

When a process tries to access a page that is **not brought into memory** (the valid-invalid bit is set to invalid), a **page fault** occurs and the hardware traps to the operating system. The procedure for handling a page fault (Fig. 5.13) is as follows:

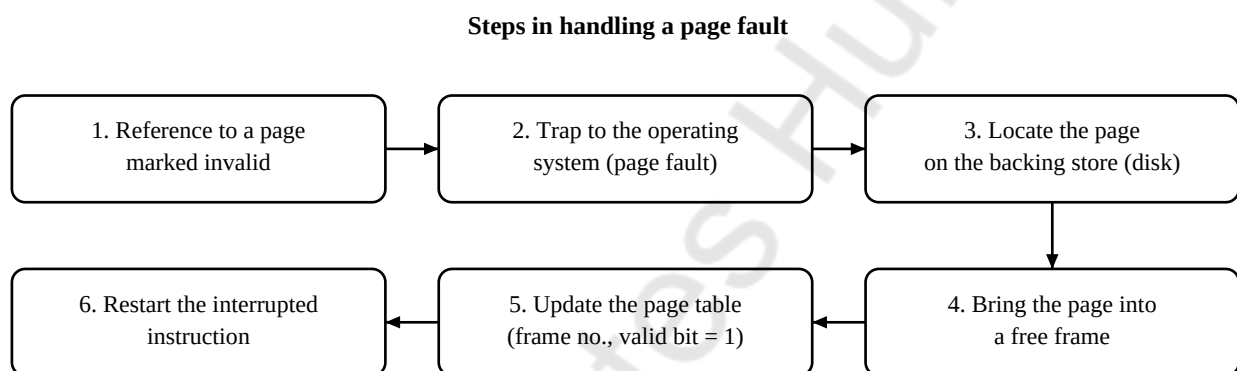


Fig. 5.13 : Steps in handling a page fault

- 1 The operating system checks an internal table (kept with the PCB) to determine whether the reference was a valid or an invalid memory access. If it was **invalid** the process is terminated.
- 2 If the reference was valid but the page is not in memory, the operating system **finds a free frame** (taken from the free-frame list). If there is no free frame, a **page-replacement algorithm** selects a victim page.
- 3 A disk operation is scheduled to **read the desired page** into the newly allocated frame. The process waits, and the CPU may be given to another process.
- 4 When the read is complete, the **page table is updated** (frame number set, valid bit = 1) and the internal table is modified.
- 5 The **instruction that was interrupted is restarted**. The process can now access the page as though it had always been in memory.

Pure demand paging starts a process with no pages in memory: the first instruction causes a page fault, and the process faults until all the pages it needs are in memory. **Locality of reference** (programs tend to use the same pages repeatedly in a limited region) keeps the page-fault rate low.

Performance of demand paging

Let p be the page-fault rate ($0 \leq p \leq 1$) and m_a the memory-access time. Then **Effective Access Time (EAT)** = $(1 - p) \times m_a + p \times (\text{page-fault service time})$. The page-fault service time includes the time to service the interrupt, read the page from disk (the largest part) and restart the process.

Example. $m_a = 200$ ns and the average page-fault service time = 8 ms = 8,000,000 ns. Then $EAT = (1 - p) \times 200 + p \times 8,000,000 = 200 + 7,999,800 \times p$ ns. If one access in 1000 causes a page fault ($p = 0.001$), $EAT = 200 + 7999.8 = \mathbf{8199.8 \text{ ns, about } 8.2 \text{ microseconds}}$, which is a slowdown by a factor of 40. To keep the degradation below 10% ($EAT < 220$ ns), p must be less than 0.0000025, i.e. fewer than one fault in 399,990 accesses.

Poly Notes Hub

5.10 Page Replacement Algorithms

When a page fault occurs and **no free frame** is available, the operating system must select a page in memory that is not really in use and swap it out to make room for the page that is needed. This is **page replacement**. The basic steps are:

- 1 Find the location of the desired page on the disk.
- 2 Find a free frame. If there is one, use it; otherwise use a **page-replacement algorithm to select a victim frame**. Write the victim page to the disk **only if it is modified** (the dirty bit is set) and change the page and frame tables accordingly.
- 3 Read the desired page into the newly freed frame and change the page and frame tables.
- 4 Restart the user process.

An algorithm is evaluated by running it on a particular string of memory references, called a **reference string**, and counting the number of **page faults**. In general, the more frames available the fewer the page faults. A good algorithm has the **lowest page-fault rate**. In all examples below the reference string is 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1 and the number of frames is 3. (F = page fault, - = hit.)

5.10.1 FIFO (First In First Out)

The simplest page-replacement algorithm. FIFO associates with each page the time when it was brought into memory. When a page must be replaced, the **oldest page** (the one that has been in memory the longest) is chosen. A FIFO queue of all pages in memory can be kept: the page at the head is replaced and a new page is inserted at the tail.

Ref	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
F1	7	7	7	2	2	2	2	4	4	4	0	0	0	0	0	0	0	7	7	7
F2		0	0	0	0	3	3	3	2	2	2	2	2	1	1	1	1	1	0	0
F3			1	1	1	1	0	0	0	3	3	3	3	3	2	2	2	2	2	1
Fault	F	F	F	F	-	F	F	F	F	F	F	-	-	F	F	-	-	F	F	F

Number of page faults with FIFO = **15**. Hit ratio = $5/20 = 0.25$.

Advantages	Disadvantages
Easy to understand and implement	May replace a heavily used page just because it is old (e.g. a variable initialized early and used continually)
Needs only a queue of pages	Performance is not always good; suffers from Belady's anomaly

Belady's anomaly

For some page-replacement algorithms the page-fault rate may **increase as the number of allocated frames increases**. This is Belady's anomaly. Consider the reference string 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5 under FIFO:

Number of frames	Page faults (FIFO)
3	9
4	10 (more frames, more faults)

Stack algorithms such as Optimal and LRU never suffer from Belady's anomaly.

5.10.2 Optimal Page Replacement

The Optimal algorithm (OPT or MIN) has the **lowest page-fault rate of all algorithms** and never suffers from Belady's anomaly. It states: **replace the page that will not be used for the longest period of time**. It needs knowledge of the future reference string, which is not available, so it cannot be implemented in a real system; it is used as a **benchmark** to compare other algorithms.

Ref	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
F1	7	7	7	2	2	2	2	2	2	2	2	2	2	2	2	2	2	7	7	7
F2		0	0	0	0	0	0	4	4	4	0	0	0	0	0	0	0	0	0	0
F3			1	1	1	3	3	3	3	3	3	3	3	1	1	1	1	1	1	1
Fault	F	F	F	F	-	F	-	F	-	-	F	-	-	F	-	-	-	F	-	-

Number of page faults with Optimal = **9**. Hit ratio = $11/20 = 0.55$. (The first 3 faults are unavoidable, because the frames are empty.)

5.10.3 LRU (Least Recently Used)

LRU is an approximation of the optimal algorithm that looks **backward** in time instead of forward. It replaces the page that **has not been used for the longest period of time**, on the assumption that the recent past is a good predictor of the near future (locality of reference).

Ref	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
F1	7	7	7	2	2	2	2	4	4	4	0	0	0	1	1	1	1	1	1	1
F2		0	0	0	0	0	0	0	0	3	3	3	3	3	3	0	0	0	0	0
F3			1	1	1	3	3	3	2	2	2	2	2	2	2	2	2	7	7	7
Fault	F	F	F	F	-	F	-	F	F	F	F	-	-	F	-	F	-	F	-	-

Number of page faults with LRU = **12**. Hit ratio = $8/20 = 0.40$. This is better than FIFO (15) but worse than Optimal (9).

Implementation of LRU

- **Counters:** each page-table entry has a time-of-use field and the CPU has a logical clock; at every reference the clock value is copied into the entry. The page with the smallest time value is replaced. Requires a search of the page table and a memory write for every reference.
- **Stack:** a stack of page numbers is kept as a doubly linked list. Whenever a page is referenced it is removed from the stack and put on the top. The most recently used page is always at the top and the **LRU page is at the bottom**. Each update needs up to six pointer changes, but no search is required for replacement.
- LRU is a stack algorithm and does not suffer from Belady's anomaly. Few computers provide enough hardware support for true LRU, so approximations (reference bits, second chance) are used.

5.10.4 NRU (Not Recently Used)

NRU is a simple algorithm that approximates LRU using the two status bits associated with each page: the **Referenced bit (R)** and the **Modified bit (M)**. When a process starts, both bits are 0 for all its pages. The **R bit is set** whenever the page is referenced (read or written), and the **M bit is set** whenever the page is written to. Periodically (for example, on every clock interrupt, every 20 ms) the **R bit is cleared** by the operating system so that pages not referenced recently can be told apart from those that have been. The M bit is not cleared, because the page must be written back to the disk if it is replaced.

At a page fault, the operating system inspects all pages and divides them into four classes using the R and M bits:

Class	R	M	Meaning
Class 0	0	0	Not referenced, not modified (best victim)
Class 1	0	1	Not referenced, modified (R was cleared by the clock tick)
Class 2	1	0	Referenced, not modified
Class 3	1	1	Referenced and modified (worst victim)

The NRU algorithm removes a page **at random from the lowest-numbered non-empty class**. It is better to remove a modified page that has not been referenced in at least one clock tick (class 1) than a clean page that is in heavy use (class 2). NRU is easy to understand, moderately efficient to implement and gives adequate performance, although not optimal.

Example. Four pages are in memory with the following bits: A (R=1, M=0), B (R=0, M=1), C (R=1, M=1), D (R=0, M=0). A page fault occurs. Classes: A = class 2, B = class 1, C = class 3, D = class 0. The lowest non-empty class is class 0, so **page D is replaced**. If D were not present, B (class 1) would be chosen next.

5.10.5 Comparison of the Algorithms

Algorithm	Page to be replaced	Page faults (example)	Remark
FIFO	The oldest page in memory	15	Simple; Belady's anomaly possible
Optimal	Page not used for the longest time in the future	9	Lowest faults; not implementable; used as a benchmark
LRU	Page not used for the longest time in the past	12	Good performance; needs hardware support (counter or stack)
NRU	Random page from the lowest non-empty (R, M) class	Depends on bits	Simple and cheap; uses R and M bits; approximates LRU

Quick Revision Points

- File = named collection of related information. Attributes: name, identifier, type, location, size, protection, time/date/user. Operations: create, write, read, reposition, delete, truncate.
- Access methods: sequential (read next, write next, reset) and direct (read n, write n).
- Swapping moves a whole process between main memory and the backing store; the transfer time is the major cost.
- Allocation: contiguous (external fragmentation, fast), linked (no direct access, pointer overhead), indexed (index block, direct access).
- Directories: single level (naming problem), two level (MFD and UFD), tree (path names, subdirectories).
- Protection: access types read, write, execute, append, delete, list; access control through ACL and owner-group-universe (rwx).
- Fixed partitioning gives internal fragmentation; variable partitioning gives external fragmentation. First fit, best fit, worst fit.
- Free space: bit map (1 bit per block) and linked list (also grouping and counting).
- Virtual memory executes processes not fully in memory. Paging uses fixed-size pages and frames; the page table maps page number to frame number; a page fault is a trap when the valid bit is 0.
- Page replacement: FIFO (oldest), Optimal (farthest future use), LRU (least recently used), NRU (lowest R, M class).

Multiple Choice Questions

Choose the correct option for each question. The answer key is given at the end of this section.

Q1. Which of the following is NOT a file attribute?

- (a) Name
- (b) Size
- (c) Protection
- (d) Burst time

Q2. Which access method allows a record to be read in any order, without reading the earlier records?

- (a) Sequential access
- (b) Direct access
- (c) Reset access
- (d) Tape access

Q3. Swapping moves a process between main memory and:

- (a) CPU registers
- (b) Cache memory
- (c) Backing store
- (d) Input queue

Q4. Which file-allocation method suffers from external fragmentation and requires the file size to be known in advance?

- (a) Contiguous allocation
- (b) Linked allocation
- (c) Indexed allocation
- (d) FAT allocation

Q5. Which allocation method supports efficient direct access and has no external fragmentation?

- (a) Contiguous
- (b) Linked
- (c) Indexed
- (d) None of these

Q6. A directory structure with a Master File Directory (MFD) and User File Directories (UFD) is called:

- (a) Single-level directory
- (b) Two-level directory
- (c) Tree-structured directory
- (d) Acyclic graph directory

Q7. In UNIX, the permission 754 gives the group the access:

- (a) rwx
- (b) r-x
- (c) r--
- (d) rw-

Q8. Fixed partitioning of memory suffers mainly from:

- (a) External fragmentation
- (b) Internal fragmentation
- (c) Thrashing
- (d) Belady's anomaly

Q9. The memory-allocation strategy that selects the smallest hole that is big enough is:

- (a) First fit
- (b) Best fit
- (c) Worst fit
- (d) Next fit

Q10. In a bit vector (1 = free) 0011010011001000, the number of free blocks is:

- (a) 4
- (b) 5
- (c) 6
- (d) 8

Q11. The page table maps:

- (a) Frame numbers to disk blocks
- (b) Page numbers to frame numbers
- (c) Files to directories
- (d) Virtual pages to TLB entries only

Q12. A page fault occurs when:

- (a) The referenced page is not in main memory
- (b) The page table is full
- (c) The CPU is idle
- (d) The referenced page is in the TLB

Q13. Belady's anomaly is associated with which page-replacement algorithm?

- (a) Optimal
- (b) LRU
- (c) FIFO
- (d) NRU

Q14. The Optimal page-replacement algorithm replaces the page that:

- (a) Is the oldest in memory
- (b) Was least recently used
- (c) Will not be used for the longest time in the future
- (d) Has the R bit set

Q15. For the reference string 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1 with 3 frames, the number of page faults under LRU is:

- (a) 9
- (b) 12
- (c) 15
- (d) 18

Answer Key

Q	Ans	Q	Ans	Q	Ans
1	d	6	b	11	b
2	b	7	b	12	a
3	c	8	b	13	c
4	a	9	b	14	c
5	c	10	c	15	b

Short explanations. Q1: burst time is a process (scheduling) property, not a file attribute. Q4: contiguous allocation needs the size in advance and fragments the disk externally. Q5: indexed allocation uses an index block, so any block can be reached directly. Q7: 754 = rwx (owner), r-x (group), r-- (universe). Q8: a process rarely fills its fixed partition. Q10: 0011 0100 1100 1000 contains six 1 bits (blocks 2, 3, 5, 8, 9, 12). Q13: FIFO can show more faults with more frames. Q15: from the simulation in Section 5.10.3 LRU gives 12 faults.

Broad Questions

- 1 Define a file. Explain the attributes, operations, types and internal structure of a file. How does the operating system use the open-file tables?
- 2 Explain the sequential and direct access methods of a file with the operations used in each. Compare them and describe indexed sequential access.
- 3 What is swapping? Explain standard swapping with a neat diagram. Discuss the issues of pending I/O and context-switch time. A process of 50 MB is swapped to a disk with a transfer rate of 25 MB/s and a latency of 10 ms; find the swap-out time.
- 4 Explain the contiguous, linked and indexed methods of allocation of disk space with neat diagrams. Compare them on the basis of fragmentation, access and reliability.
- 5 Describe the single-level, two-level and tree-structured directories with diagrams. Discuss the advantages and disadvantages of each, and explain absolute and relative path names.
- 6 Explain the types of file access. Describe the access-control list and the owner-group-universe scheme used by UNIX. Interpret the permission 754.
- 7 Explain fixed and variable partitioning of memory. Describe first fit, best fit and worst fit. For holes of 100, 500, 200, 300 and 600 KB and processes of 212, 417, 112 and 426 KB, show how each strategy places the processes.
- 8 Explain how free space is managed on a disk using a bit map and a linked list. Mention the advantages and disadvantages of each, and the variations grouping and counting.
- 9 What is virtual memory? Explain paging, the page table and address translation with a neat diagram. Explain the steps in handling a page fault and derive the effective access time.
- 10 Explain the FIFO, Optimal, LRU and NRU page-replacement algorithms. For the reference string 1, 2, 3, 4, 2, 1, 5, 6, 2, 1, 2, 3, 7, 6, 3, 2, 1, 2, 3, 6 and 3 frames, find the number of page faults for FIFO, Optimal and LRU. What is Belady's anomaly?

- End of Chapter 5 -