Memory Management

The main purpose of a computer system is to execute programs. These programs and the data they access must be (at least partially) loaded in main memory during execution.

Modern computer systems maintain several processes in memory during system execution. There are various approaches for memory management. Selection of them depends on many factors, including the hardware design. In the following sections we discuss various ways for memory management by OS.

Contents

Chapter Objectives

- *Explain how memory is allocated to processes.*
- *Discuss different memory management techniques*
- *Illustrate the difference between logical and physical addresses.*
- *Explain how the logical to physical address binding happens*
- *Discuss virtual memory*
- *Illustrate what is demand paging and how a page fault is handled*

9.1 Main Memory

Memory is central to the operation of a modern computer system. Memory consists of a large array of **bytes**, each with its own **address**. The CPU fetches instructions from memory according to the value of the **program counter**. A program must be loaded (from disk) into memory and put in a process to run.

9.1.1 Basic Hardware

Main memory and the **registers** built into each processing core are the only general-purpose storage that the CPU can access directly. These registers take one CPU clock to access, whereas the main memory can take many cycles. To mitigate this problem, fast memory (cache) is used between main memory and the CPU registers.

Protection of memory is required to ensure correct operation of the processes. For this, each process has to run within a separate memory space. It protects the processes from each other, and it is must for concurrent execution of multiple processes in memory. In order to separate memory addresses, we need to be able to determine the range of legal addresses that a process can access. To ensure that a process can access only its legal addresses, **a pair of base and limit registers** can be used to define the logical address space of a process:

- **The base register** holds the smallest legal physical memory address.
- **The limit register specifies the size of the range**.

For example, if the base register = 300040 and limit register is 120900, then the program can legally access all address from 300040 to 420939.

Any attempt by a program executing in user mode to access OS memory or other user's memory causes error. The base and limit registers can only be loaded by the OS.

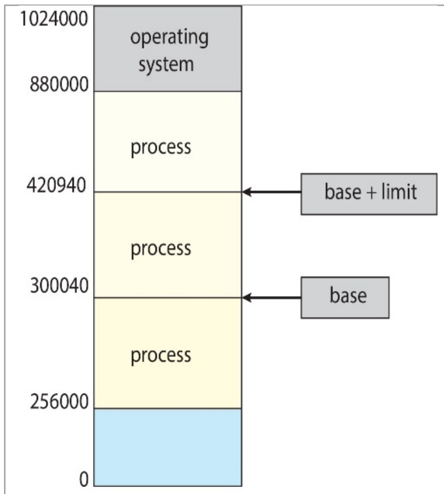

Figure 1 *A Base and a Limit register define a logical address space*

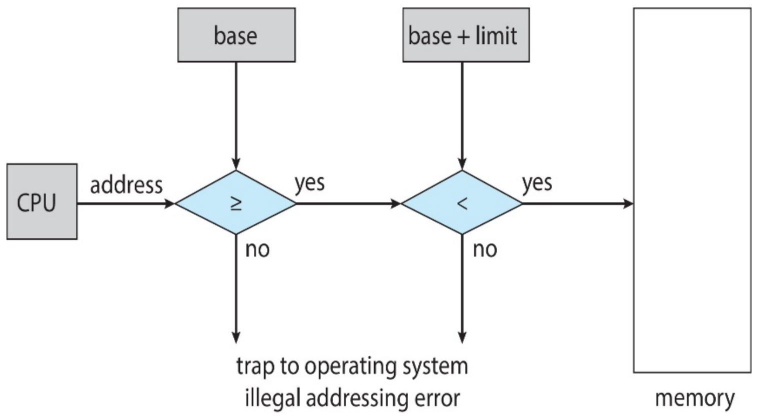

Figure 2 *Hardware address protection with base and limit registers*

9.1.2 Address Binding

Usually a program resides on a disk as a binary executable file. To run, the program must be brought into memory and placed within the context of a process. As the process executes, it accesses instructions and data from memory. After the process terminates, this memory is freed.

Most systems allow a user process to reside in any part of the physical memory. Addresses in the source program are generally symbolic (such as a variable). A compiler typically binds these symbolic addresses to relocatable addresses (such as 14 bytes from the starting address of this module). The linker or loader in turn binds these relocatable addresses to absolute addresses.

Address binding of **instructions and data** to **memory addresses** can happen at **three** different stages:

- **Compile time –** If memory location of a process is known before compilation, then absolute code can be generated. If the start location changes, the program must be recompiled.
- **Load time** – If the memory location is not known at the compile time, then the compiler must generate relocatable code. In this case, the final binding can be delayed until load time.
- **Execution time –** If the process can be moved during its execution from one memory segment to another, then the address binding must be delayed until run time. Most OS use this method.

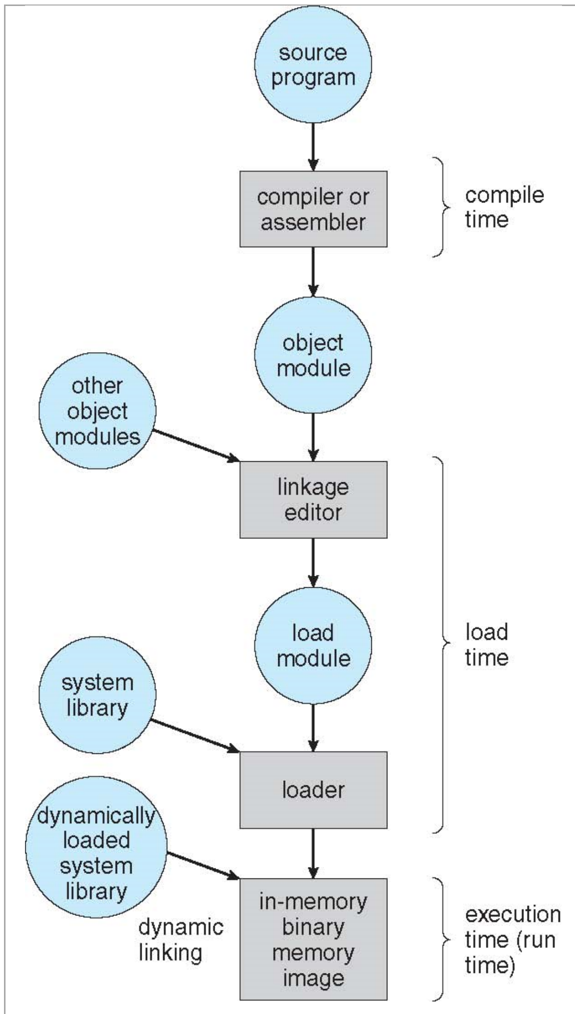

***Figure 3** Multistep processing of a user program*

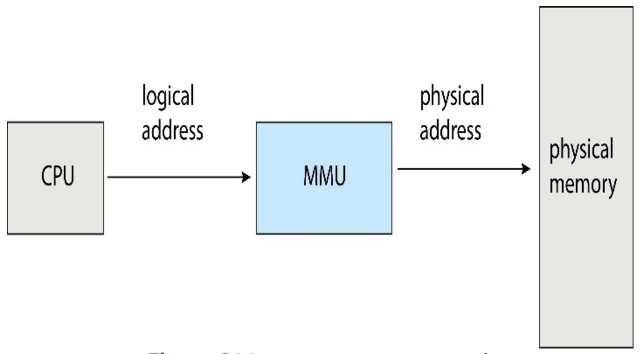

***Figure 4** Memory management unit*

9.1.3 Logical vs. Physical Address Space

The concept of a logical address space that is bound to a separate physical address space is central to proper memory management.

- **Logical address** – It is an address generated by the CPU (also referred to as virtual address). These are translated by the **memory management unit (MMU)** to actual physical addresses in memory. Logical address space is the set of all logical addresses generated by a program.

- **Physical address** – These are the addresses seen by the memory unit, loaded into the memory-address register of the memory. Physical address space is the set of all physical addresses generated by a program.

Logical and physical addresses are the same in compile-time and load-time address-binding schemes. But they differ in execution-time address-binding scheme. The user program deals with logical addresses; it never sees the real physical addresses.

9.2 Contigious Memory Allocation

The main memory must allocate both the OS and the various user processes. One approach to allocating memory for processes is to allocate partitions of contiguous memory of varying sizes. Each process is contained in a single section of memory that is contagious to the section containing the next process.

9.3 Paging

One problem of contiguous memory allocation is that it creates many fragmentation in memory. To minimize this, modern operating systems use paging to manage memory allocation, where the physical address space of a process can be noncontiguous.

In this approach, the physical memory is divided into **fixed-sized blocks** called **frames**. The frame size is power of 2 and between 512 bytes to 16 Mbytes. The logical memory is divided into **blocks** of the same size called **pages**. A logical address now has **two** parts:

- **A page number** – It is an **index** of a per process page table that contains the **base address** of each **frame** in physical memory.
- **A page offset** – The offset is the specific location in the frame being referenced.

OS keeps track of the free frames, and each page in the logical address space is mapped to a free frame in the physical memory during program execution.

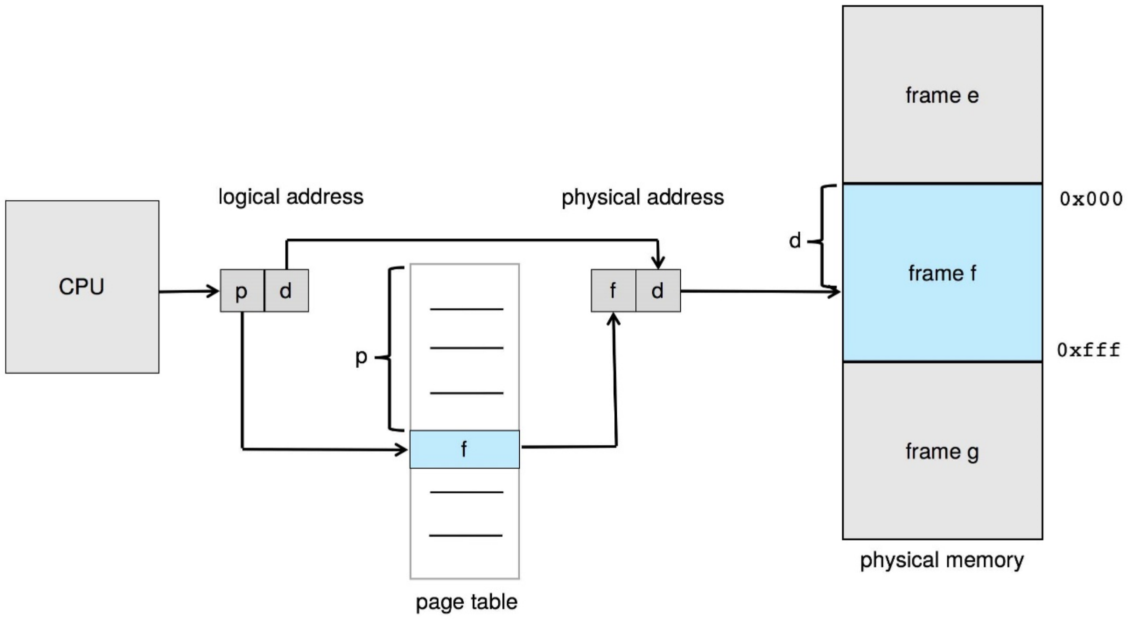

Figure 5 *Paging hardware*

10.1 Virtual Memory

Virtual memory is a technique that allows the execution of processes that are not **completely loaded** into memory. It abstracts the main memory into extremely large, uniform array of storage, separating logical memory from the physical memory. Thus programs larger than the physical memory can run. The virtual address space of a process refers to the logical view of how a process is stored in memory.

10.2 Demand Paging

The previous approach of memory management we discussed is to load the entire program in physical memory during execution. But a program can be very large, and we may not initially need the entire program in memory. Therefore, an alternative technique is to **load pages only when they are needed**. This is called demand paging, commonly used in virtual memory systems. Pages that are never needed are never loaded into physical memory.
As the pages are partially loaded, with each page table entry, a **valid–invalid bit** is included to mark the pages which are in memory or not-in-memory. If a page is needed that is not in memory (invalid), a **page fault** occurs. The procedure for handling a page fault is as follows:

- The OS checks an internal table to see if the reference was valid or invalid memory access.
  - If the reference is invalid, the process is terminated.
  - If it is valid, then the absent page is brought into memory from secondary storage:
    - A free frame is located
    - Read operation is done to bring the page into the frame
    - The page table in updated with marking the page valid.
    - The instruction that caused the page fault is restarted.

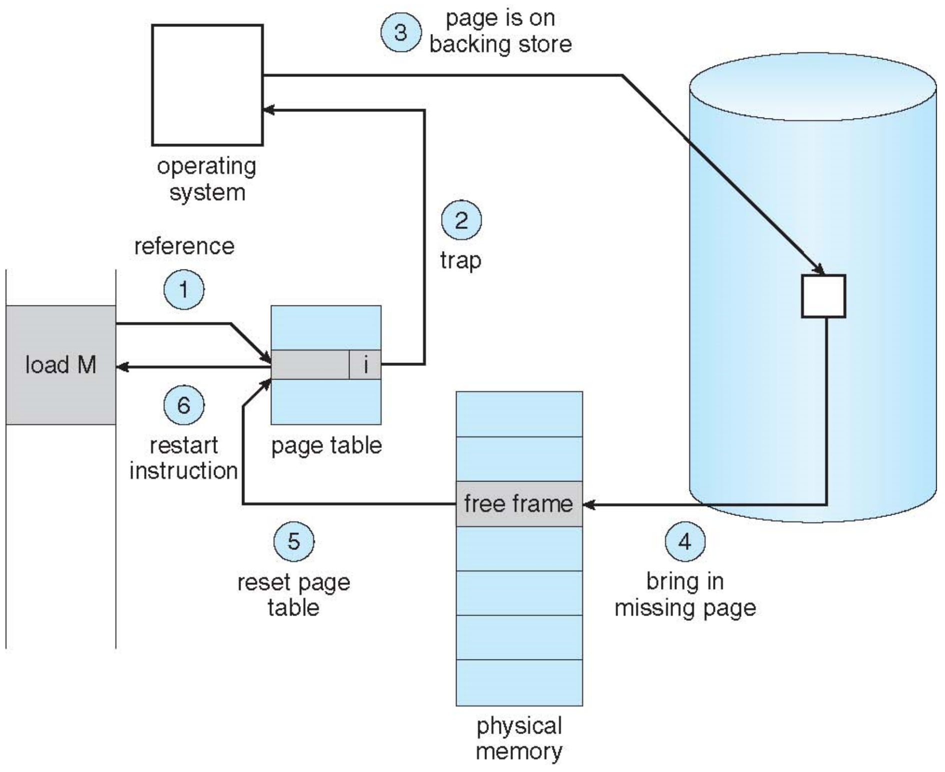

Figure 6 Steps in handling a page fault

Storage Management

Computer systems must provide mass storage for permanently storing files and data. Modern computers implement mass storage as secondary storage, using both hard disks and nonvolatile memory devices. There are various storage devices, so OS need to provide a wide range of functionality so that applications can control these properly. In this section we discuss how mass storage is structured and accessed.

Contents

Chapter Objectives

- *Describe the physical structure of secondary storage devices*
- *Discuss how these can be accessed efficiently.*

11.1 Overview of Mass Storage Structure

Hard disk drives and nonvolatile memory devices are the major secondary storage I/O units on most computers. Modern secondary storage is structured as large one-dimensional array of **logical blocks**.

Requests for secondary storage I/O are generated by the file system and by the virtual memory system. Each request specifies the address on the device to be referenced in the form of a **logical block number**.

11.2 HDD Scheduling

One of the responsibilities of the OS is to use the hardware efficiently. For HDDs, this means:

- **Minimizing** access time and
- **Maximizing** data transfer bandwidth.

For HDDs and other mechanical storage devices that use platters, **access time** has **two** major components:

- **The seek time** – is the time for device arm to move the heads to the cylinder containing the desired sector
- **The rotational latency** – is the additional time for the platter to rotate the desired sector to the head.

The device bandwidth is the total number of bytes transferred divided by the total time between the first request for service and the completion of the last transfer.

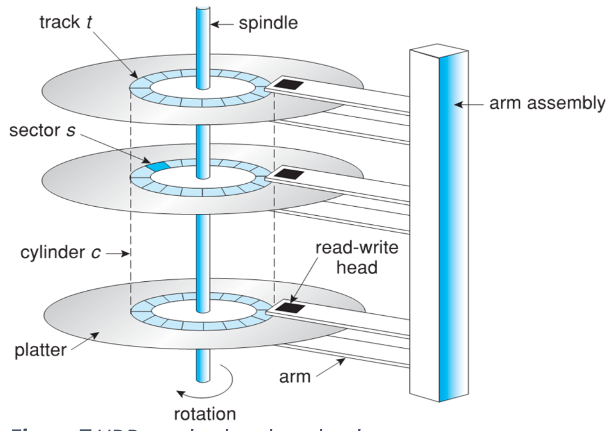

Figure 7 *HDD moving head mechanism*

Both the access time and bandwidth can be improved by managing the **order** in which **storage I/O requests** are **serviced**. A queue is used to hold the read/write requests to a device. The following algorithms are designed to make those improvements through strategies for disk queue ordering.

11.2.1 FCFS Scheduling

As we mentioned earlier that drive controllers have small buffers and can manage a queue of I/O requests. We illustrate the scheduling algorithm with a request queue with following requests for I/O to blocks on cylinders (0-199): **98, 183, 37, 122, 14, 124, 65, 67**

According to FCFS strategy. If the disk head is initially at cylinder 53, it will first move from 53 to 98, then to 183, 37, 122, 14, 124, 65 and finally to 67 with a total head movement of 640 cylinders.

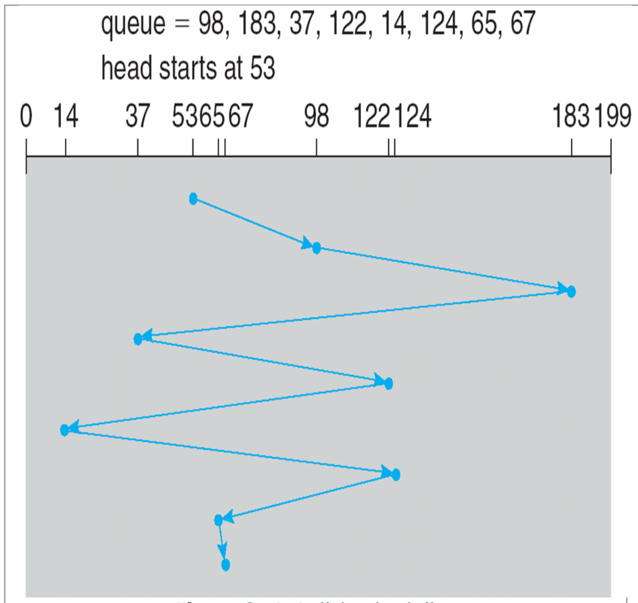

Figure 8 *FCFS disk scheduling*

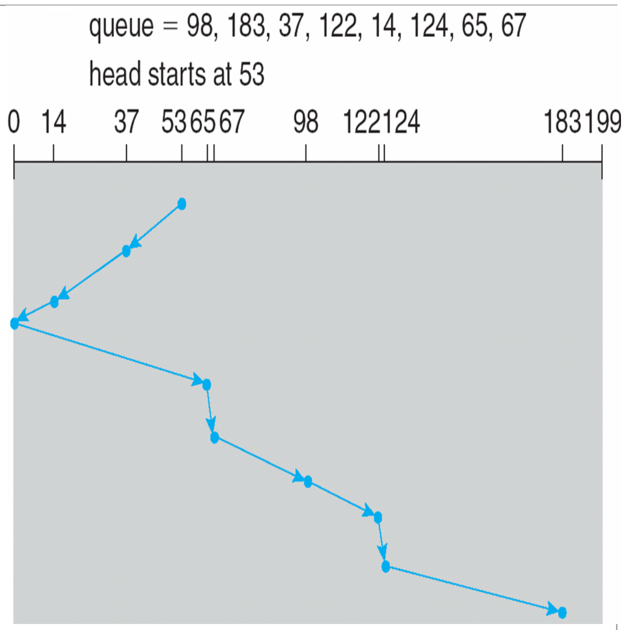

Figure 9 *SCAN disk scheduling*

11.2.2 SCAN Scheduling

In the scan algorithm, the disk arm starts at one end of the disk, and moves toward the other end, servicing requests as it reaches each cylinder, until it gets to the other end of the disk, where the head movement is reversed and servicing continues. The head continuously scans back and forth across the disk. It is also called the elevator algorithm.

For the read/write requests shown earlier, when the disk arm is moving towards 0 and the initial position is 53, the head will next serve 37, and then 14. At cylinder 0, the arm will reverse and will move toward the other end of the disk, servicing the requests at 65, 67, 98, 122, 124 and 183. The total head movement is 208 cylinders, which is better than FCFS.

Performance of disk scheduling algorithms can vary greatly on hard disks. In contrasts, because sold-state disks have no moving parts, performance varies little among scheduling algorithms, and quite often a simple FCFS can be used.

File System

A file is a collection of related information defined by its creator. Files are mapped to the OS onto physical mass-storage devices. A file system describes how files are mapped onto physical devices as well as how they are accessed and manipulated by both users and programs. In this section we discuss the basics of file system management by OS.

Contents

Chapter Objectives

- *Explain the function of file systems*
- *Describe the file operations and file structures*

13.1 File Concept

File system is the most visible aspect of a general-purpose OS. It provides the mechanism for on-line storage and access of both data and programs of the OS and users. By file system, the OS provides a uniform logical view of the stored information. The OS abstracts from the physical properties of its storage devices to define a logical storage unit – the file. Files are mapped by the OS onto physical, usually nonvolatile storage devices. From users' perspective, a file is the smallest allotment of logical secondary storage. Commonly files represent programs (source and object) and data. Data files may be numeric, alphabetic, alpha-numeric or binary.

13.1.1 File Attributes

A file's attributes vary from one OS to another, but generally consists of following these:

- **Name** – The symbolic file name is the only information kept in human-readable form.
- **Identifier** – This unique tag (usually a number) identifies file within file system. It is the non-human-readable name for the file.
- **Type** – This is needed for systems that support different types of files.
- **Location** – This is a pointer to the location of the file on the device.
- **Size** – The current size of the file (in bytes, words or blocks).
- **Protection** – These are access control information on who can do reading, writing, executing and so on.

- **Timestamps and user identification** – These information may consists of time and performer (OS, user) of file creation, last modification, and last use. These data are useful for protection, security, and usage monitoring.

13.1.2 File Operations

A file is an abstract data type. To define a file properly, we need to consider the operations that can be performed on files. The OS provides system calls to do the following operations on files:

- **Creating a file** – This is allocating space for the file and making a new entry in the directory.
- **Opening a file** – This is searching the directory structure on disk for the file and moving its contents to memory. All other operations except create and delete, requires an open file.
- **Writing a file** – This is performing a write operation to a file specified by an open file handle and a location in the file, given by a pointer for doing the next write.
- **Reading a file** – This is performing a read operation from a file specified by an open file handle and a read pointer to the location for doing the next read.
- **Reposition within a file** – This is repositioning the current file position pointer of the open file to a given value. This does not require any actual I/O.
- **Deleting a file** – This requires locating the directory entry for the file, releasing the file space and marking the directory entry free.
- **Truncating a file** – This allows all attributes of the file to remain unchanged except for file length. The file length can be reset to length 0 and its file space can be released.
- **Closing a file** – This is moving the content of the file from the main memory to the directory structure on the disk.

13.2 Access Methods

While using, a file can be accessed in several ways:

- **Sequential Access** – This is the simplest access method. Information in the file is processed in order, one record after another.
- **Direct Access** – In this method a file is made up of **fixed length logical records**, which allow programs to read and write records rapidly and in no particular order.

14 File System Implementation

Several on-storage and in-memory **structures** are used to implement a file system. In this section we discuss the following structures:

- **A boot control block** (per volume) – contains information needed by the system to boot an OS from that volume. If the disk does not contain an OS, this block can be empty. It is typically the first block of a volume.
- **A volume control block** (per volume) – contains volume details, such as the number of blocks per volume, the size of the blocks, a free-block count and free-block pointers etc.
- **A directory structure** (per file system) – is used to organize the files. This can include the file names and associated identifiers.

- **A file control block** (per file) – contains many details about the file. It has a unique identifier number to allow association with a directory entry. To create a new file, a new file control block (FCB) is allocated.

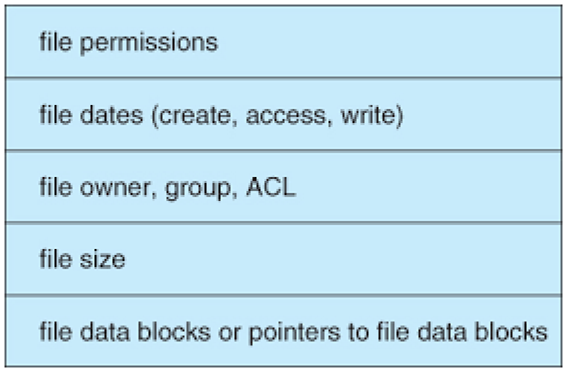
file permissions
file dates (create, access, write)
file owner, group, ACL
file size
file data blocks or pointers to file data blocks

Figure 10 *A typical file control block*

14.4 Allocation methods

The direct access nature of the secondary storage gives us flexibility in the implementation of files. The main problem is how to allocate space to these files so that storage space is utilized effectively and files can be accessed quickly.

There are **three** major methods of allocating secondary storage space for files:

- **Contiguous allocation** – requires that each file occupy a set of contiguous blocks on the device. It is defined by the address of the first block and the length of the file. It causes external fragmentation of the storage volume.
- **Linked allocation** – each file is a linked list of storage blocks, the blocks may be scattered anywhere in the device. Each directory entry has a pointer to the first block of the file. It minimizes external fragmentation but can only be used effectively for sequential access.
- **Indexed allocation** – each file has its own index block, with an array of storage block addresses. The i^{th} entry in the index block points to the i^{th} block of the file.

Security

Security ensures the authentication of system users to protect the integrity of the information stored in the system (both data and code), as well as physical resources of the computer system. The security system prevents unauthorized access, malicious destruction or alteration of data and accidental introduction of inconsistency. In this section we define the security problem in a computer system.

16.1 The Security Problem

We say that a system is secure if its resources are used and accessed as intended under all circumstances. Unfortunately, total security cannot be achieved. We need mechanisms to make security breaches a rare occurrence rather than the norm. Security violations (or misuses) of the system can be categorized as intentional (malicious) or accidental. It is easier to protect against accidental misuse than against malicious misuse. For the most part protection mechanisms are the core of accident avoidance.

The following are the forms of accidental or malicious security attack - security violation categories:

- **Breach of confidentiality** – Unauthorized reading of data
- **Breach of integrity** – Unauthorized modification of data
- **Breach of availability** – Unauthorized destruction of data
- **Theft of service** – Unauthorized use of resources
- **Denial of service (DOS)** – Prevention of legitimate use

The intruders/hackers/attackers are those who attempt to breach security. To protect a system from the attackers we must take security measures at four levels as security breach can happen at any level:

- **Physical level**
- **Network level**
- **Operating system level**
- **Application level**

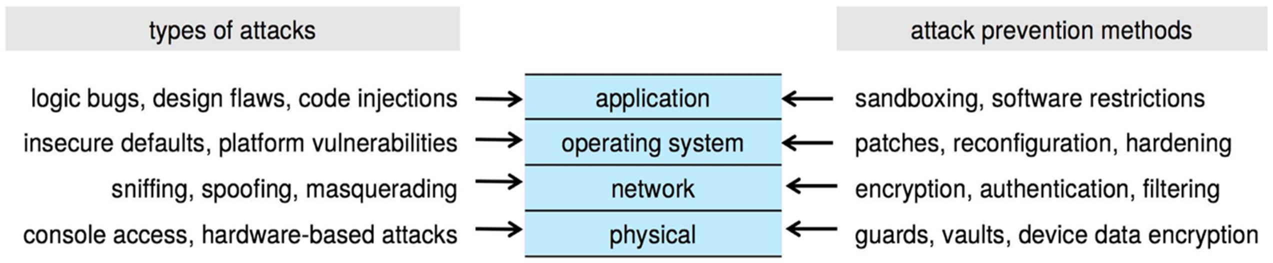
types of attacks
attack prevention methods
logic bugs, design flaws, code injections
application
sandboxing, software restrictions
insecure defaults, platform vulnerabilities
operating system
patches, reconfiguration, hardening
sniffing, spoofing, masquerading
network
encryption, authentication, filtering
console access, hardware-based attacks
physical
guards, vaults, device data encryption

Figure 11 *The four layered model of security*