MODULE 4

Memory Management

Virtual Memory Management

File System Interface And Options

Background

- Memory consists of 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. These instructions may cause additional loading from and storing to specific memory addresses.
- A typical instruction-execution cycle, for example, first fetches an instruction from memory. The instruction is then decoded and may cause operands to be fetched from memory. After the instruction has been executed on the operands, results may be stored back in memory.
- We are interested only in the sequence of memory addresses generated by the running program.
- We begin our discussion by covering several issues that are pertinent to the various techniques for managing memory

Why Memory Management is Required?

- **Allocate and de-allocate memory before and after process execution.**
- **To keep track of used memory space by processes.**
- **To minimize fragmentation issues.**
- **To proper utilization of main memory.**
- **To maintain data integrity while executing of process.**

Basic Hardware

- Program must be brought (from disk) into memory and placed within a process for it to be run
- Main memory and registers are only storage CPU can access directly
- Memory unit only sees a stream of **addresses + read requests, or address + data and write requests**
- Register access in one CPU clock (or less)
- Main memory can take many cycles, causing a **stall**
- **Cache** sits between main memory and CPU registers
- Protection of memory required to ensure correct operation

Basic Hardware Continued

- A pair of **base** and **limit registers** define the logical address space.
- The base register holds the **smallest legal physical memory address**; **the limit register specifies the size of the range**(range of logical addresses).

- CPU must check every memory access generated in user mode to be sure it is between base and limit for that user.

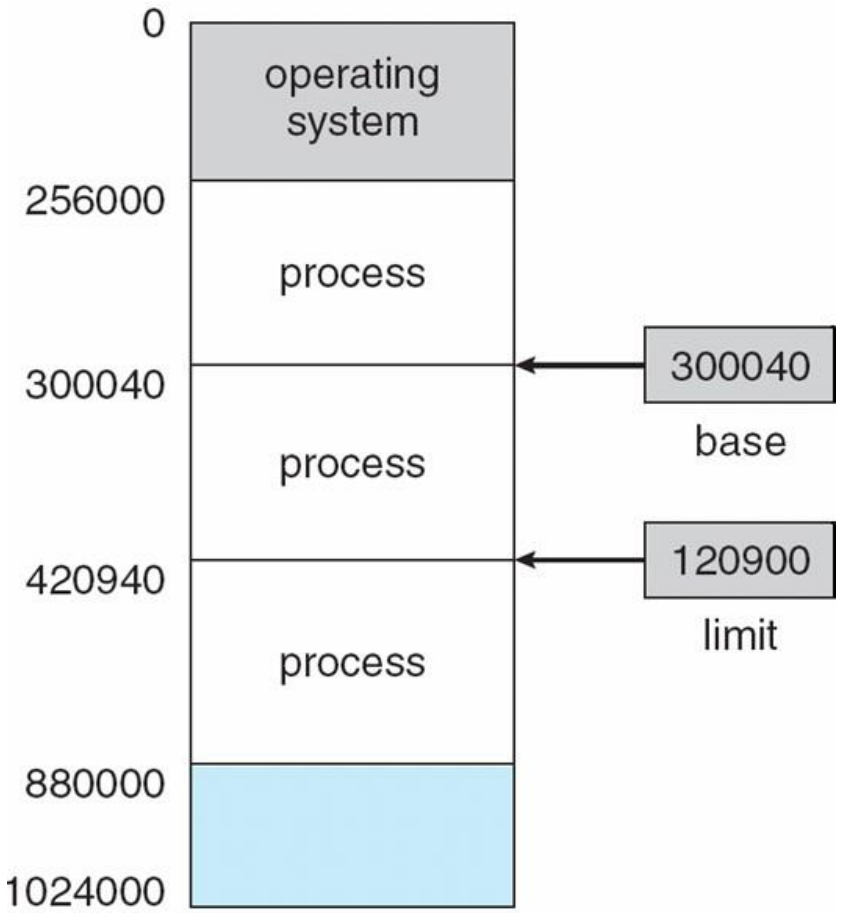


Figure 8.1 A base and a limit register define a logical address space

Hardware Address Protection

- Protection of memory space is accomplished by having the CPU hardware compare every address generated in user mode with the registers. Any attempt by a program executing in user mode to access operating-system memory or other users' memory results in a trap to the operating system

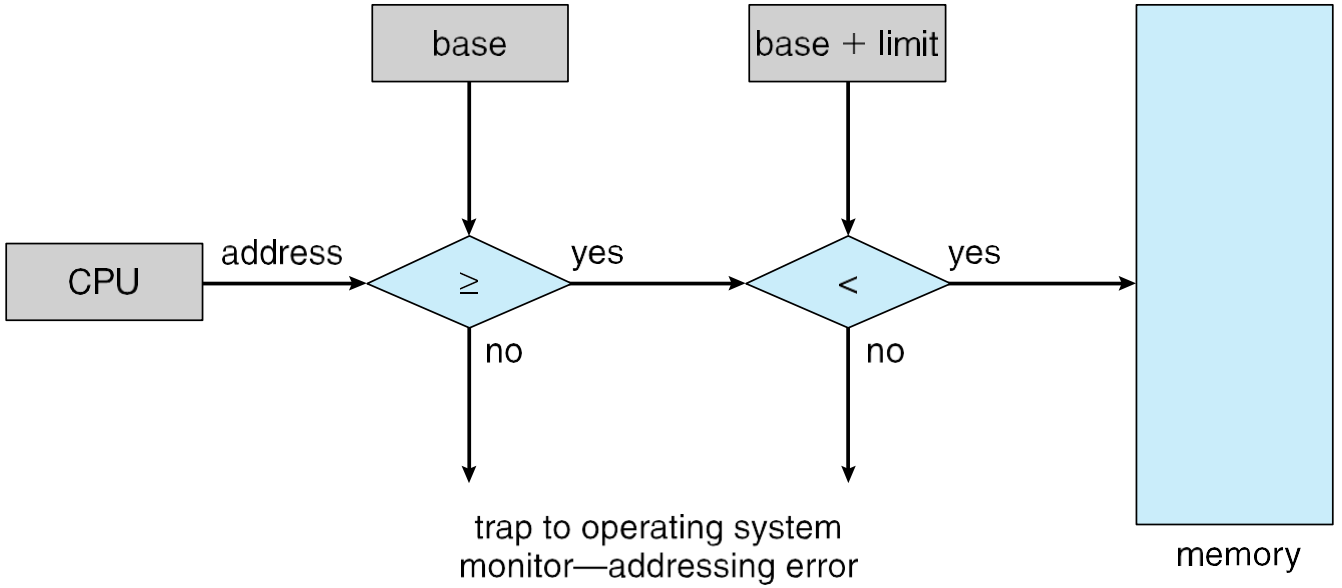


Figure 8.2 Hardware address protection with base and limit registers

Address Binding

- Programs on disk, ready to be brought into memory to execute form an **input queue**
  - Without support, must be loaded into address 0000
- In most cases, a user program will go through several steps-some of which may be optional-before being executed (Figure 8.3).
- Addresses may be represented in different ways during these steps.
  - Addresses in the source program are generally **symbolic (such as count).**
  - A compiler will typically bind these **symbolic addresses to relocatable addresses** (such as "14 bytes from the beginning of this module").
  - The linkage editor or loader will in turn bind the **relocatable addresses to absolute addresses** (such as 74014). Each binding is a mapping from one address space to another

Address Binding Continued

- Address binding of instructions and data to memory addresses can happen at three different stages
  - **Compile time**: If memory location known a priori, **absolute code** can be generated; must recompile code if starting location changes
  - **Load time**: Must generate **relocatable code** if memory location is not known at compile time
  - **Execution time**: Binding delayed until run time if the process can be moved during its execution from one memory segment to another
    - Need hardware support for address maps (e.g., base and limit registers)

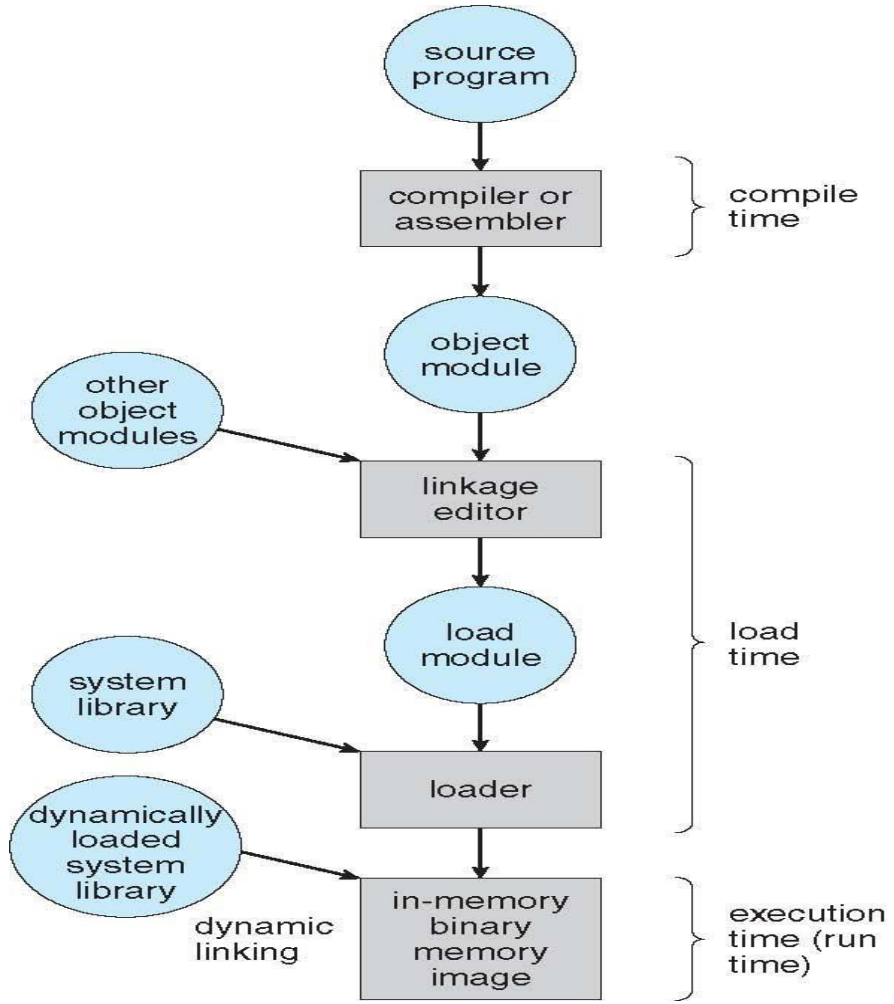


Figure 8.3 Multistep processing of a user program.

Logical Versus Physical Address Space

- **Logical address** – generated by the CPU; also referred to as **virtual address**
- **Physical address** – address seen by the memory unit
- Logical and physical addresses are the same in compile-time and load-time address-binding schemes; logical (virtual) and physical addresses differ in execution-time address-binding scheme.
- The set of all logical addresses generated by a program is a **logical address space**; the set of all physical addresses corresponding to these logical addresses is a **physical address space.**
- The run-time mapping from virtual to physical addresses is done by a hardware device called the **memory-management unit (MMU)**.

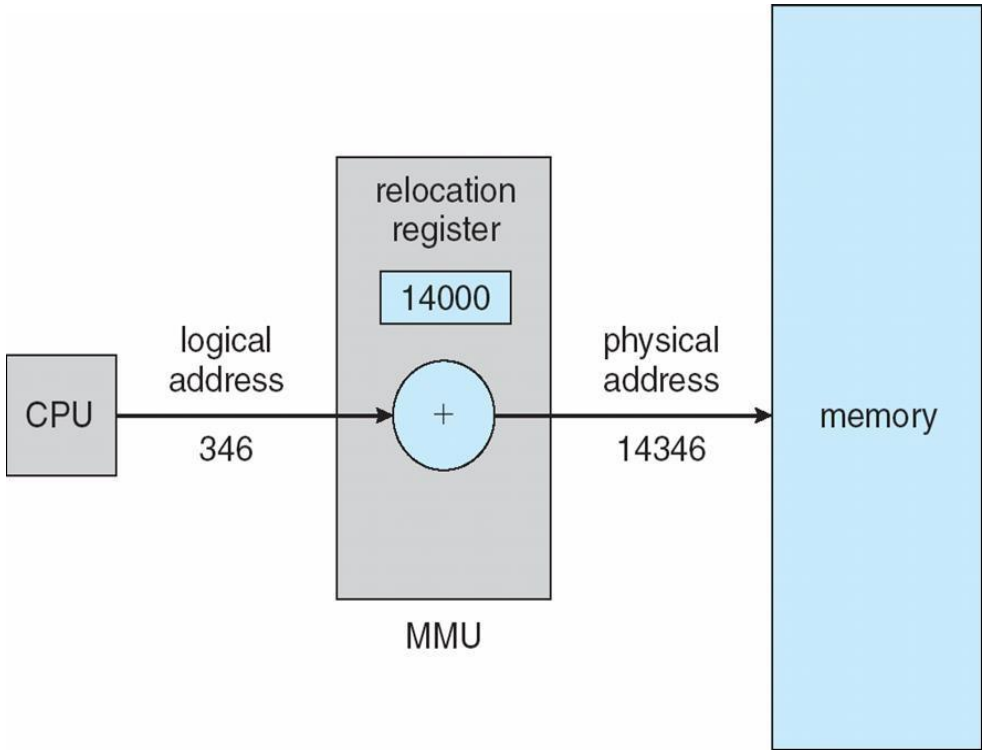


Figure 8.4 Dynamic relocation using a relocation register

Dynamic Loading, Dynamic Linking and Shared Libraries

- To obtain better memory-space utilization, we can use dynamic loading. With dynamic loading, a routine is not loaded until it is called. The advantage of dynamic loading is that an unused routine is never loaded.
- **Static linking** – system libraries and program code combined by the loader into the binary program image. Dynamic linking –linking postponed until execution time.
- Small piece of code, **stub**, used to locate the appropriate memory-resident library routine. Stub replaces itself with the address of the routine, and executes the routine
- Operating system checks if routine is in processes' memory address.
- A shared library is a file containing object code that several *a.out* files may use simultaneously while executing.

Swapping

- A process can be **swapped** temporarily out of memory to a backing store, and then brought back into memory for continued execution
  - Total physical memory space of processes can exceed physical memory
- **Backing store** – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images
- **Roll out, roll in** – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed
- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped
- System maintains a **ready queue** of ready-to-run processes which have memory images on disk

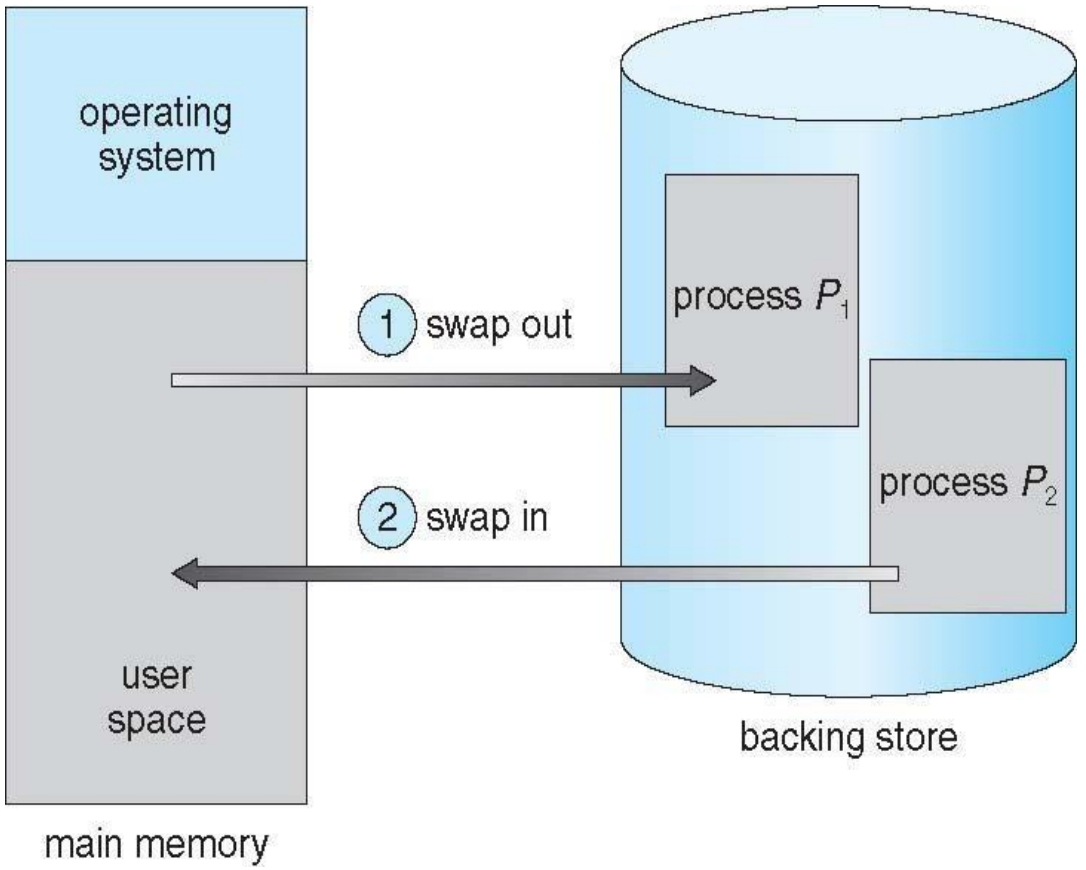


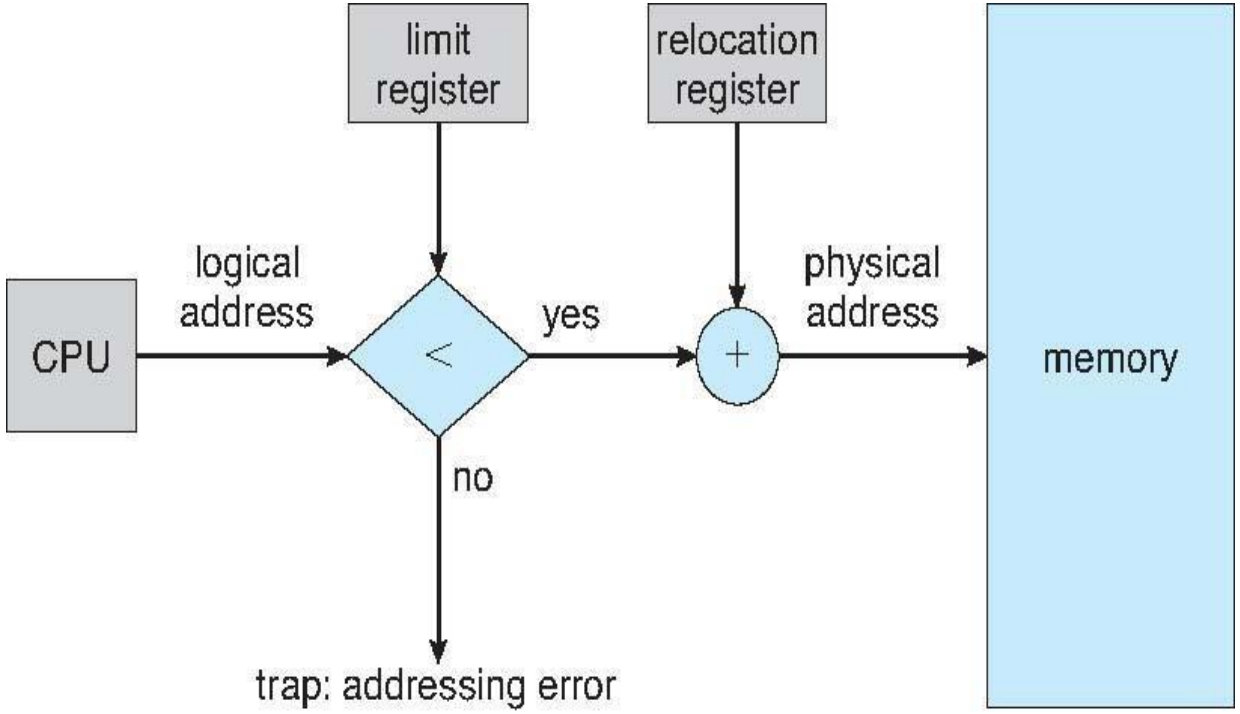
Figure 8.5 Swapping of two processes using a disk as a backing store.

Contiguous Memory Allocation

- The main memory must accommodate both the operating system and the various user processes. We therefore need to allocate the parts of the main memory in the most efficient way possible.
- The memory is usually divided **into two partitions**: one for the resident operating system and one for the user processes. We can place the operating system in either low memory or high memory.
- We usually want several user processes to reside in memory at the same time. We therefore need to consider how to allocate available memory to the processes that are in the input queue waiting to be brought into memory. In this contiguous memory allocation, each process is contained in a single contiguous section of memory.

Memory Mapping and Protection

- Relocation registers used to protect user processes from each other, and from changing operating-system code and data
  - Base register contains value of smallest physical address
  - Limit register contains range of logical addresses – each logical address must be less than the limit register
  - MMU maps logical address *dynamically*
  - Can then allow actions such as kernel code being **transient** and kernel changing size



Memory Allocation

One of the simplest methods for allocating memory is to divide memory into several fixed-sized partitions.

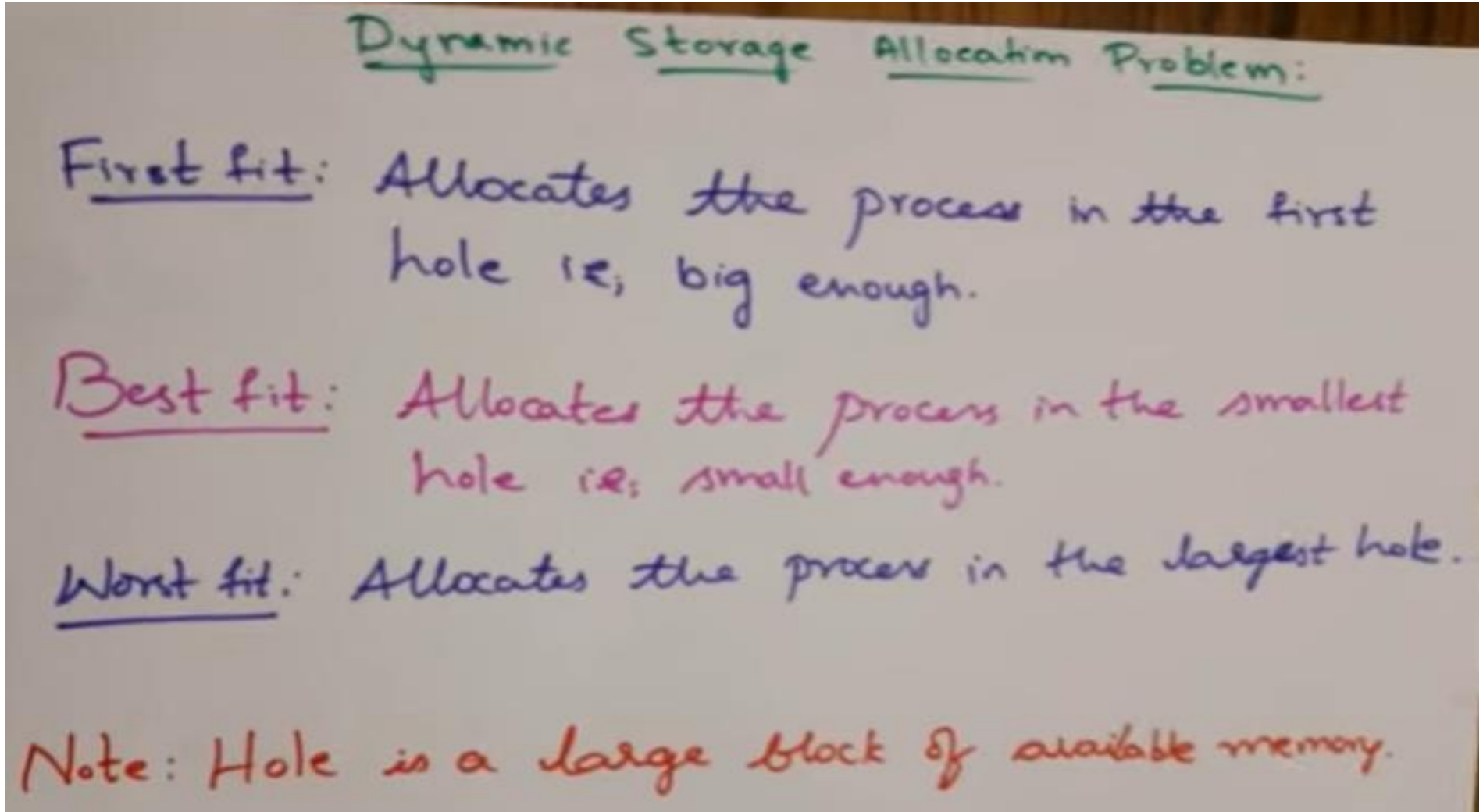
Fixed-partition Scheme: Multi-programming with fixed partitioning is a contiguous memory management technique in which the main memory is divided into fixed sized partitions which can be of equal or unequal size. Whenever we have to allocate a process memory then a free partition that is big enough to hold the process is found. Then the memory is allocated to the process.If there is no free space available then the process waits in the queue to be allocated memory. It is one of the most oldest memory management technique which is easy to implement. s is needed, keeping the rest available to satisfy future requests.

Variable Partitioning :

Multi-programming with variable partitioning is a contiguous memory management technique in which the main memory is not divided into partitions and the process is allocated a chunk of free memory that is big enough for it to fit. The space which is left is considered as the free space which can be further used by other processes. It also provides the concept of compaction. In compaction the spaces that are free and the spaces which not allocated to the process are combined and single large memory space is made.

Memory Allocation

General dynamic storageallocation problem, ,Which concerns how to satisfy a request of size n from a list of free holes. There are many solutions to this problem. **The first-fit, best-fit, and worst-fit** strategies are the ones most commonly used to select a free hole from the set of available holes

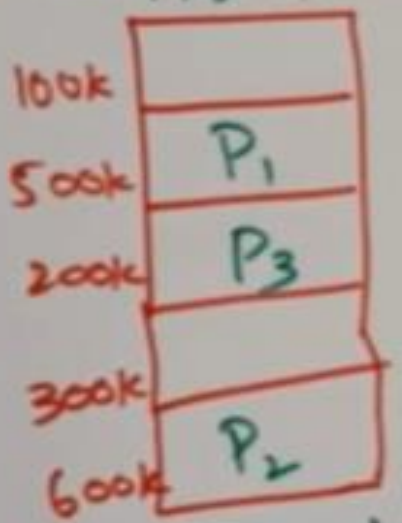


Numericals on First fit, Best fit & Worst fit

1) Given 5 memory partitions of 100k, 500k, 200k, 300k and 600k (in order). How would each of first fit, best fit and worst fit algorithm place processes of 212KB, 417K, 112K, 426K (in order)?

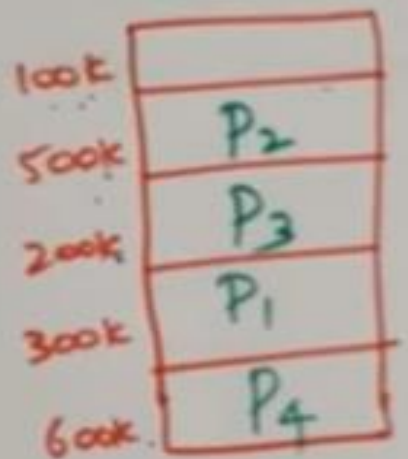
Solution:

First fit



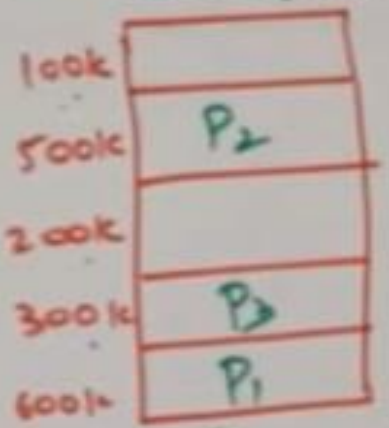
P4 → starvation

Best fit - smallest



Best fit

Worst fit → largest hole



P4 → starvation

2) Given 5 memory partitions of 10k, 5k, 30k, 25k, 40k (in order)
How would each of first fit, best fit and worst fit algorithm
place processes of J_1-20k, J_2-15k, J_3-30k, J_4-5k (in order)

Fragmentation

Both the first-fit and best-fit strategies for memory allocation suffer from external fragmentation

Fragmentation is an unwanted problem in the operating system in which the processes are loaded and unloaded from memory, and free memory space is fragmented. Processes can't be assigned to memory blocks due to their small size, and the memory blocks stay unused.

Contiguous memory allocation allocates space to processes whenever the processes enter RAM. These RAM spaces are divided either by fixed partitioning or by dynamic partitioning. As the process is loaded and unloaded from memory, these areas are fragmented into small pieces of memory that cannot be allocated to coming processes.

One solution to the problem of external fragmentation compaction, Another possible solution to the external-fragmentation problem is to permit the logical address space of the processes to be noncontiguous, thus allowing a process to be allocated physical memory wherever the latter is available. Two complementary techniques achieve this solution: paging and segmentation.

Paging

Paging is a memory-management scheme that permits the physical address space of a process to be noncontiguous. Paging avoids the considerable problem of fitting memory chunks of varying sizes onto the backing store; most memory-management schemes used before the introduction of paging suffered from this problem

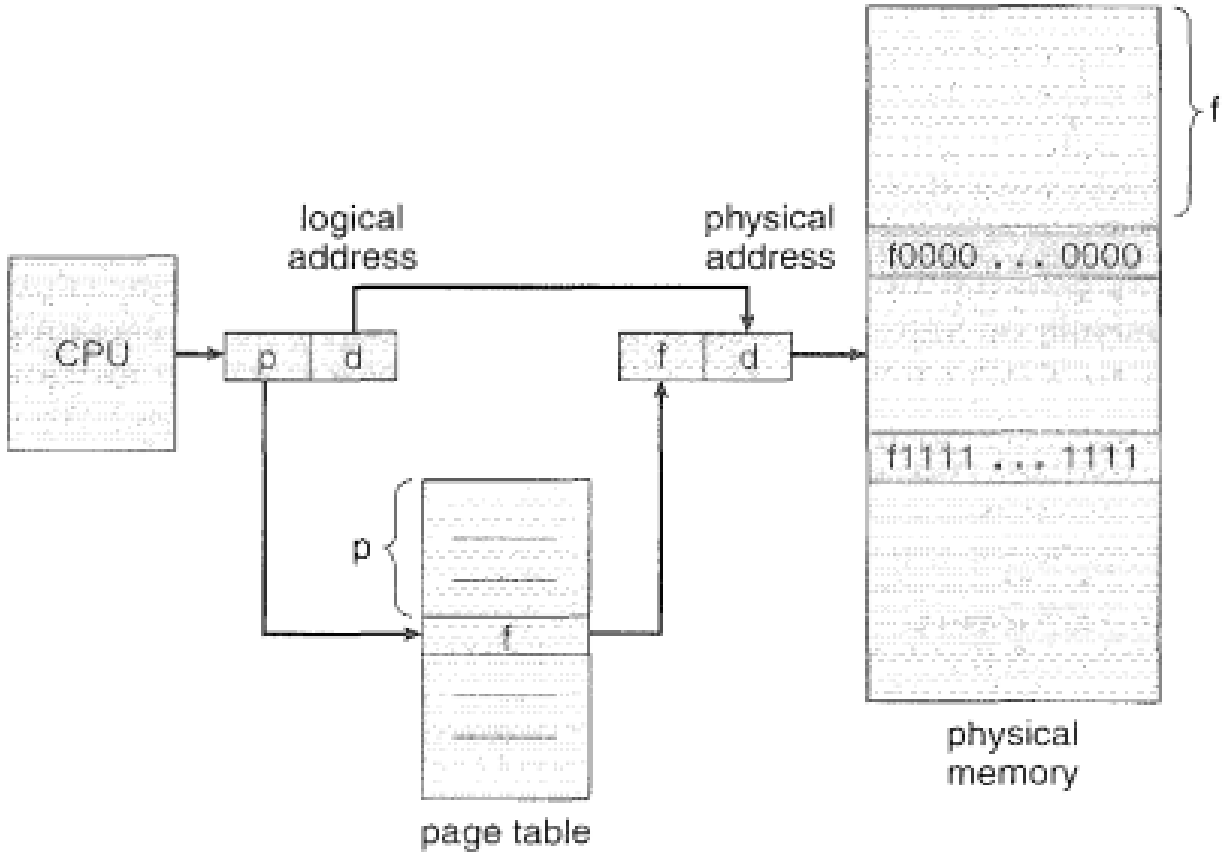


Figure 8.7 Paging hardware.

The basic method for implementing paging involves breaking physical memory into fixed-sized blocks called frames and breaking logical memory into blocks of the same size called pages. When a process is to be executed, its pages are loaded into any available memory frames from the backing store. The backing store is divided into fixed-sized blocks that are of the same size as the memory frames.

The hardware support for paging is illustrated in Figure 8.7. 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. The page table contains the base address of each page in physical memory. This base address is combined with the page offset to define the physical memory address that is sent to the memory unit. The paging model of memory is shown .in Figure 8.8

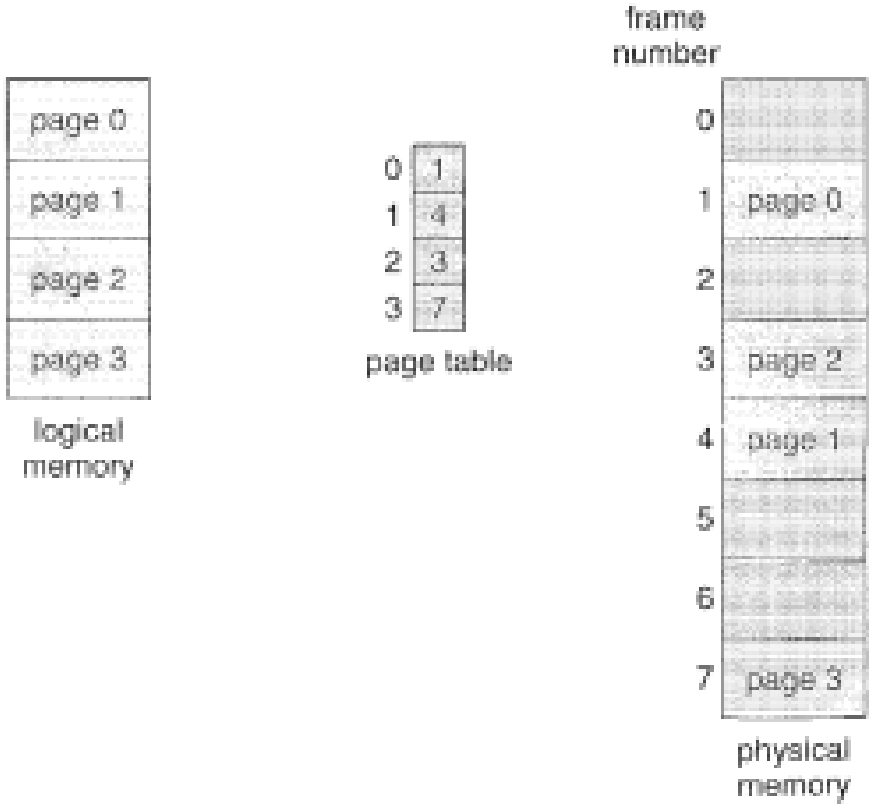


Figure 8.8 Paging model of logical and physical memory.

Segmentation

- **Memory-management scheme that supports user view of memory**
- A program is a collection of segments
  - A segment is a logical unit such as:

    main program

    procedure

    function

    method

    object

    local variables, global variables

    common block

    stack

    symbol table

    arrays

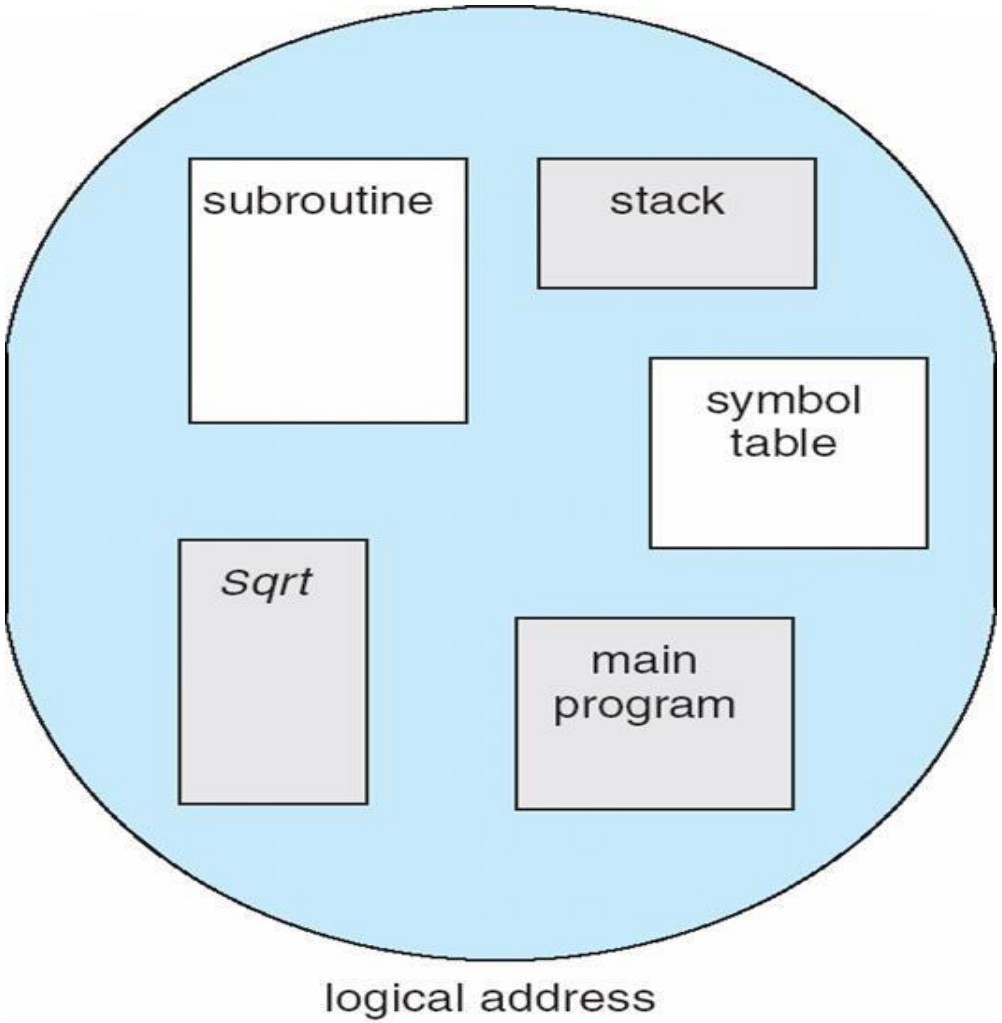
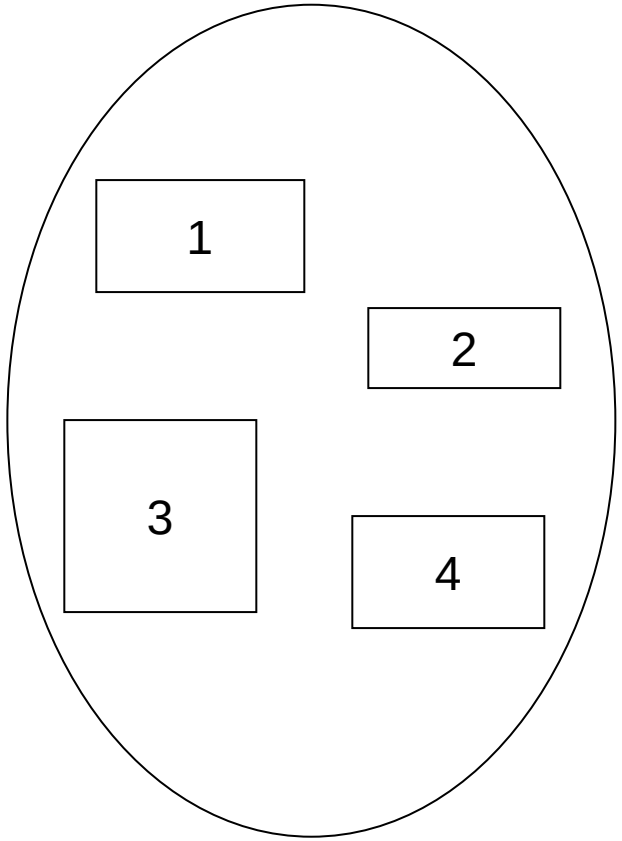


Figure 8.18 User's view of a program.

Logical View of Segmentation



user space

physical memory space

Segmentation Hardware

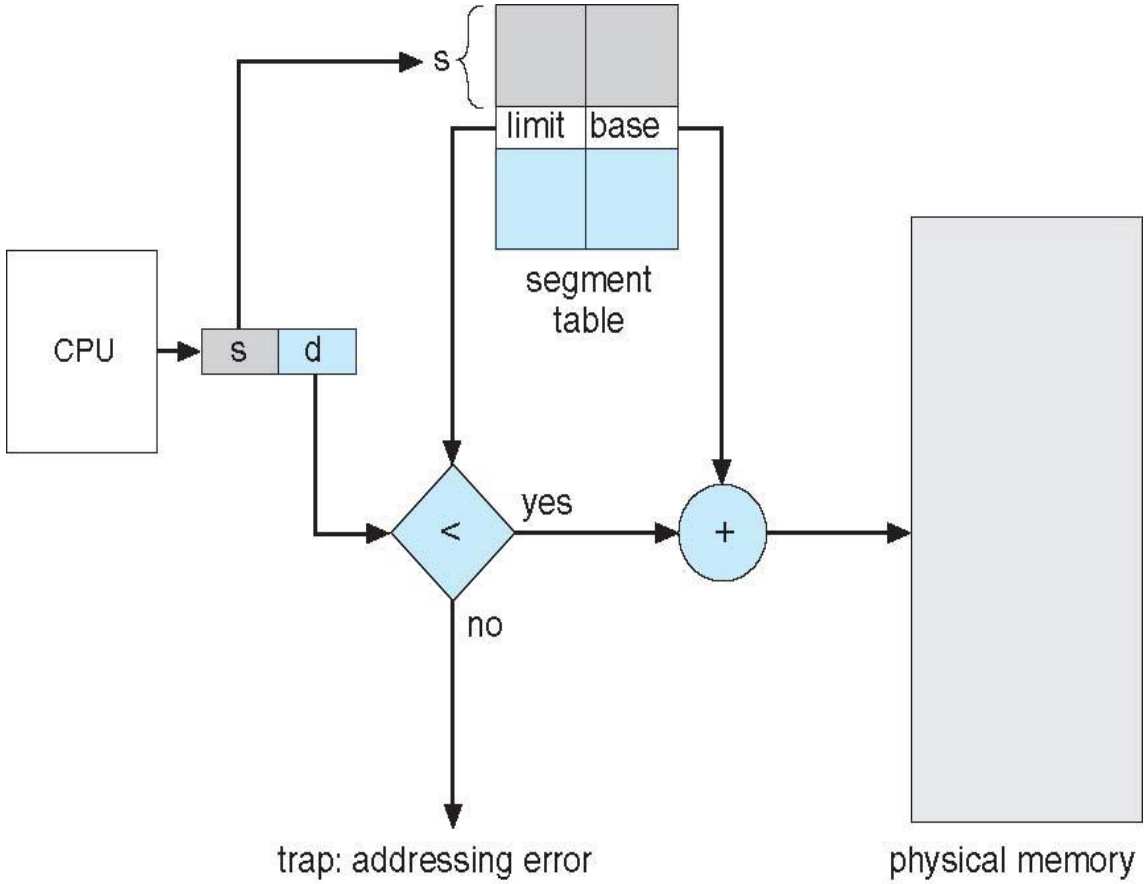


Figure 8.19 Segmentation hardware.

Segmentation Hardware Continued

For example, segment 2 is 400 bytes long and begins at location 4300. Thus, a reference to byte 53 of segment 2 is mapped onto location 4300 + 53 = 4353. A reference to segment 3, byte 852, is mapped to 3200 (the base of segment 3) + 852 = 4052. A reference to byte 1222 of segment 0 would result in a trap to the operating system, as this segment is only 1,000 bytes long

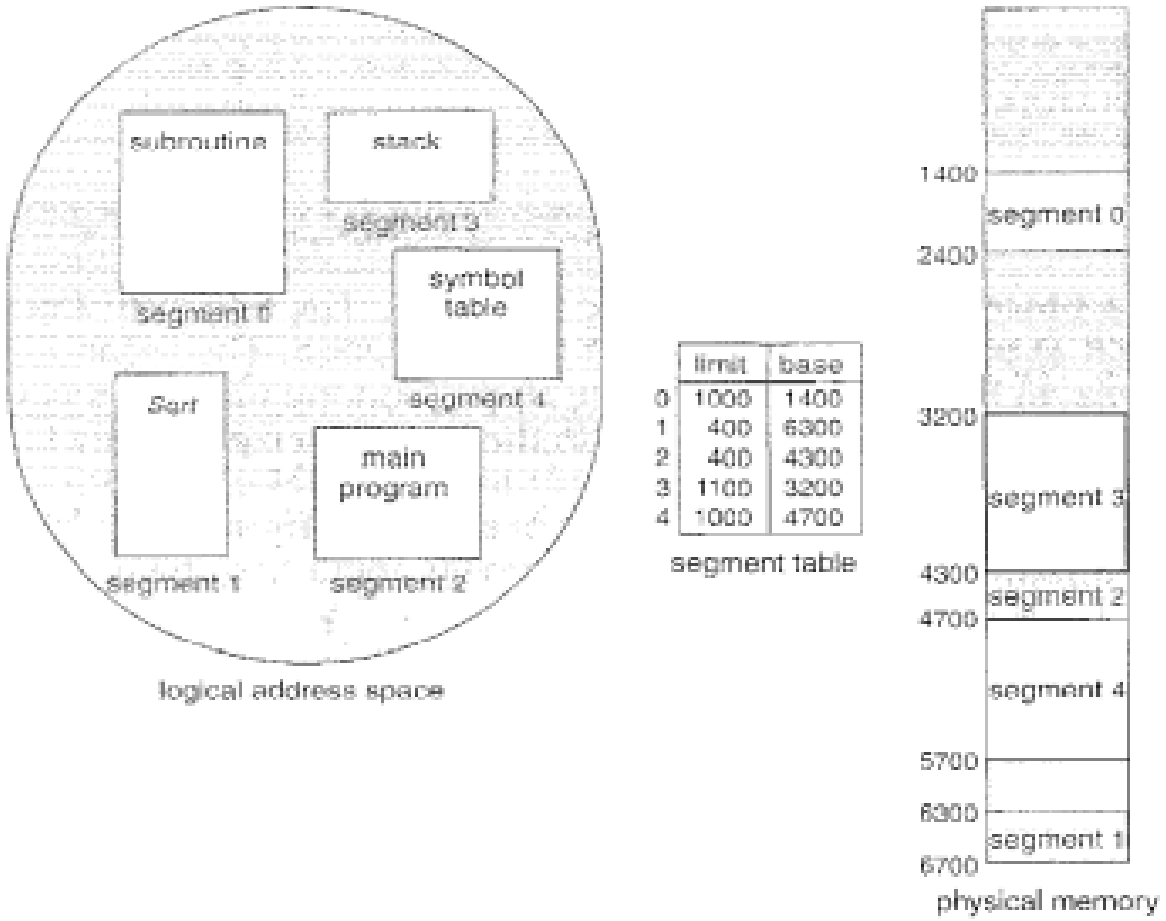


Figure 8.20 Example of segmentation.

Overview of Virtual Memory Management

- Background
- Demand paging; Copy-on-write;
- Page replacement; Allocation of frames; Thrashing.
- File System
- Implementation of File System: 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

Background

It is a technique that allows the execution of process that are not completely in memory.

Advantages

- Programmers should not worry about the available physical memory
- Large programs can now be easily executed
- Reduce the external fragmentation
- Degree of Multiprogramming increased
- Virtual Memory is implemented by Demand Paging and Demand Segmentation

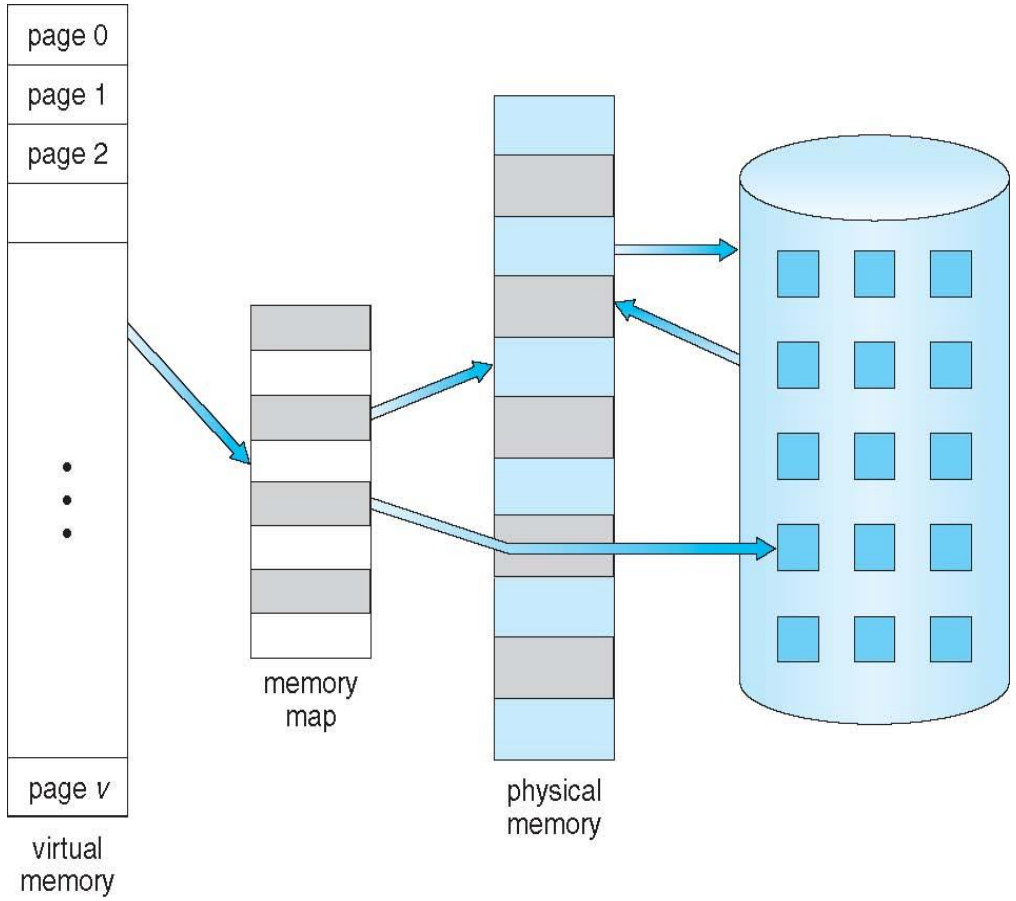


Figure 9.1 Diagram showing virtual memory that is larger than physical memory

Virtual Memory

Virtual memory – separation of user logical memory from physical memory

- Only part of the program needs to be in memory for execution
- Logical address space can therefore be much larger than physical address space
- Allows address spaces to be shared by several processes
- Allows for more efficient process creation
- More programs running concurrently
- Less I/O needed to load or swap processes.

Virtual/Logical Address space: logical view of how process is stored in memory

- Usually start at address 0, contiguous addresses until end of space
- Meanwhile, physical memory organized in page frames

- MMU must map logical to physical

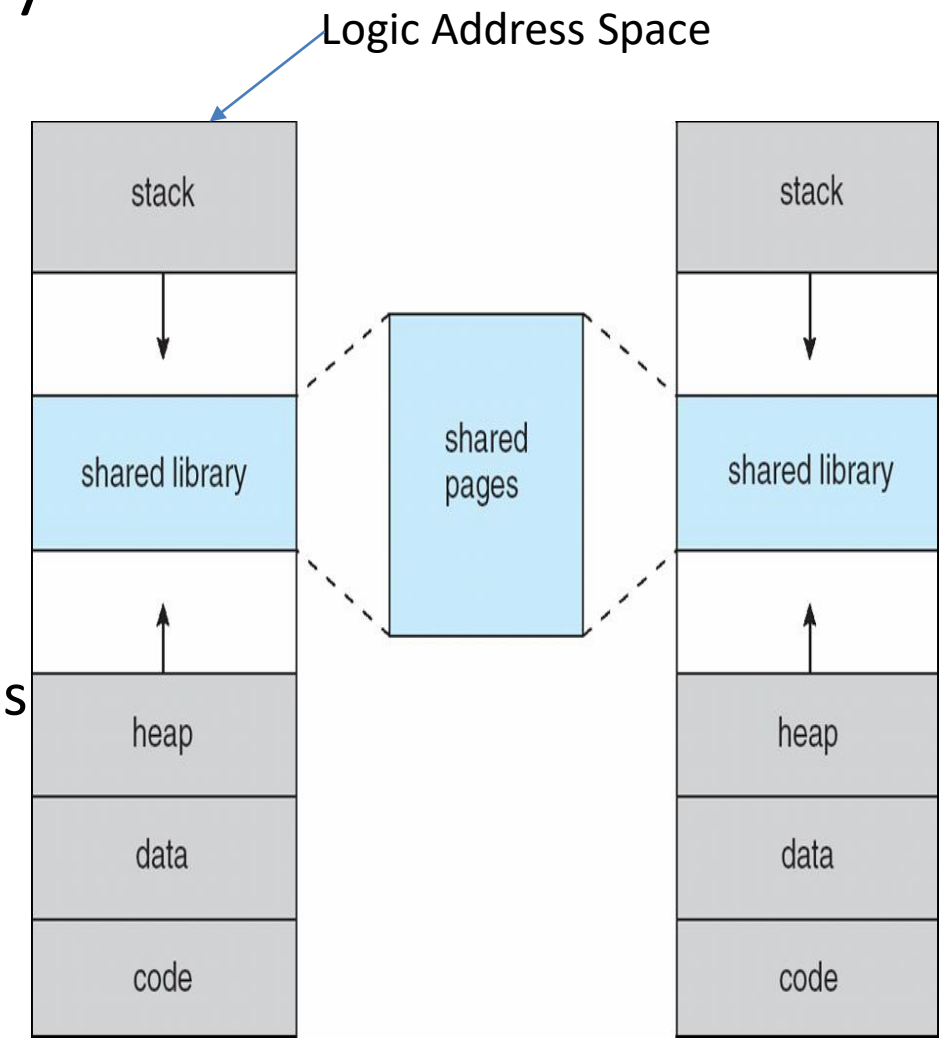


Figure 9.3 Shared library using virtual memory.

Demand Paging

Loading the **entire program into memory** results in loading the executable code for all options.
An alternative strategy is to **initially load pages only as they are needed**. This technique is known as demand paging and is commonly used in virtual memory systems.
With demand-paged virtual memory, pages are only loaded when they are demanded during program execution; pages that are never accessed are thus never loaded into physical memory.
A demand-paging system is similar to a paging system with swapping (Figure 9.4) where processes reside in secondary memory (usually a disk). When we want to execute a process, we swap it into memory. Rather than swapping the entire process into memory, however, we use a lazy swapper. A lazy swapper never swaps a page into memory unless that page will be needed

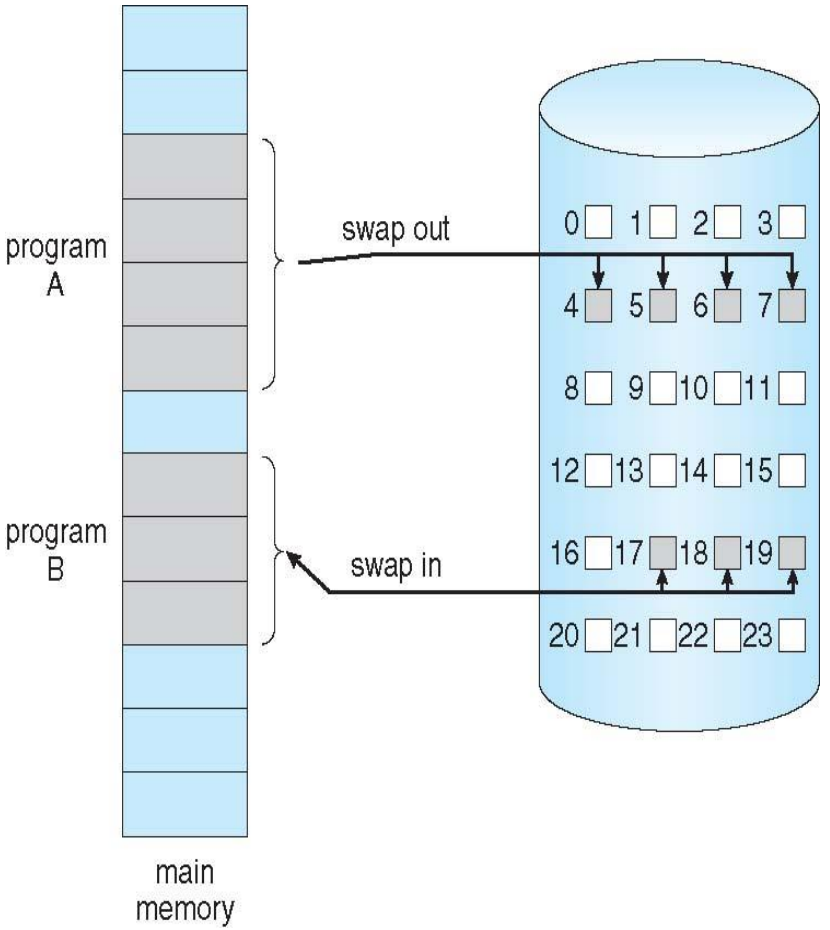


Figure 9.4 Transfer of a paged memory to contiguous disk space

Demand Paging Continued

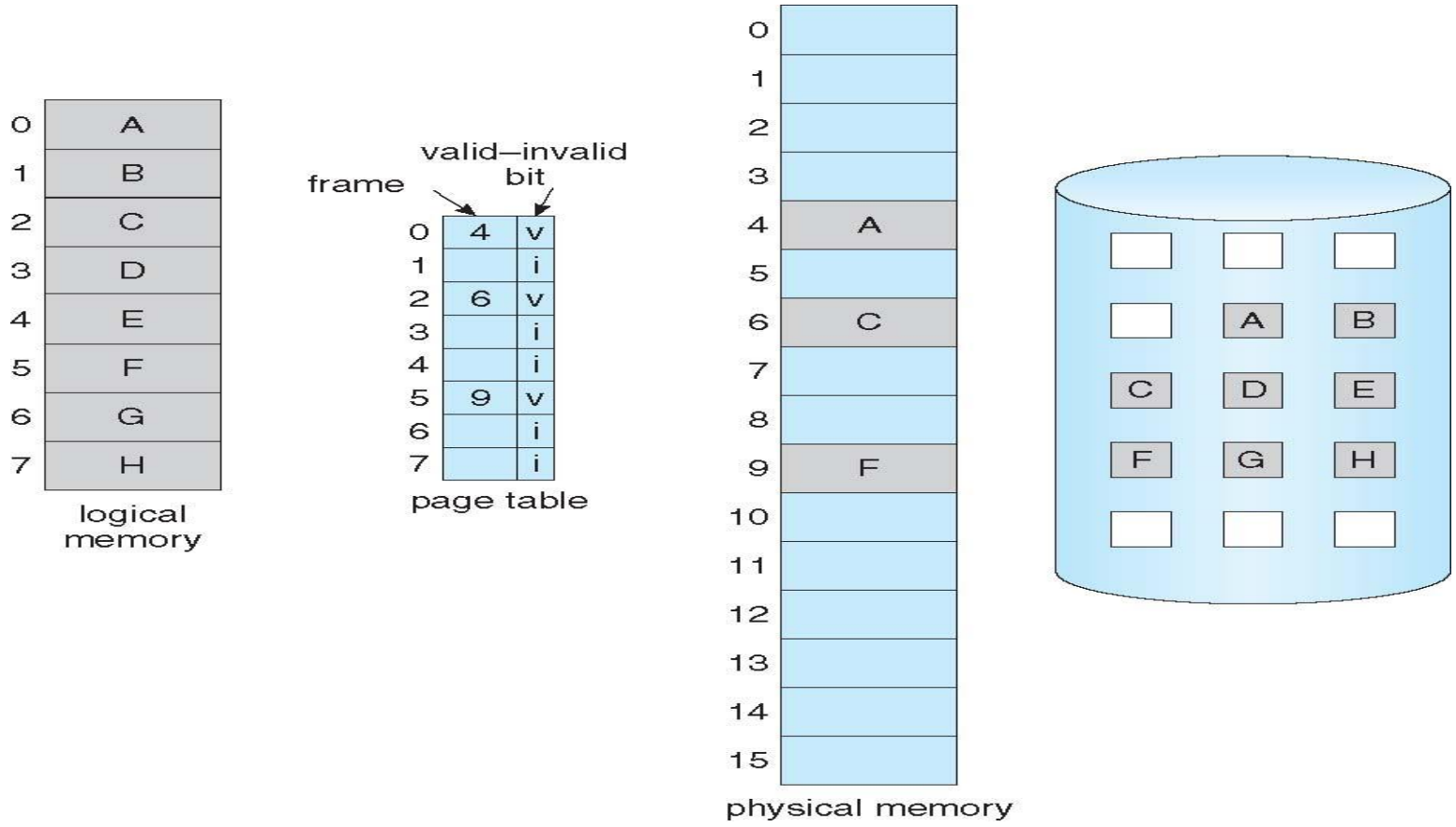


Figure 9.5 Page table when some pages are not in main memory.

Steps in Handling a Page Fault

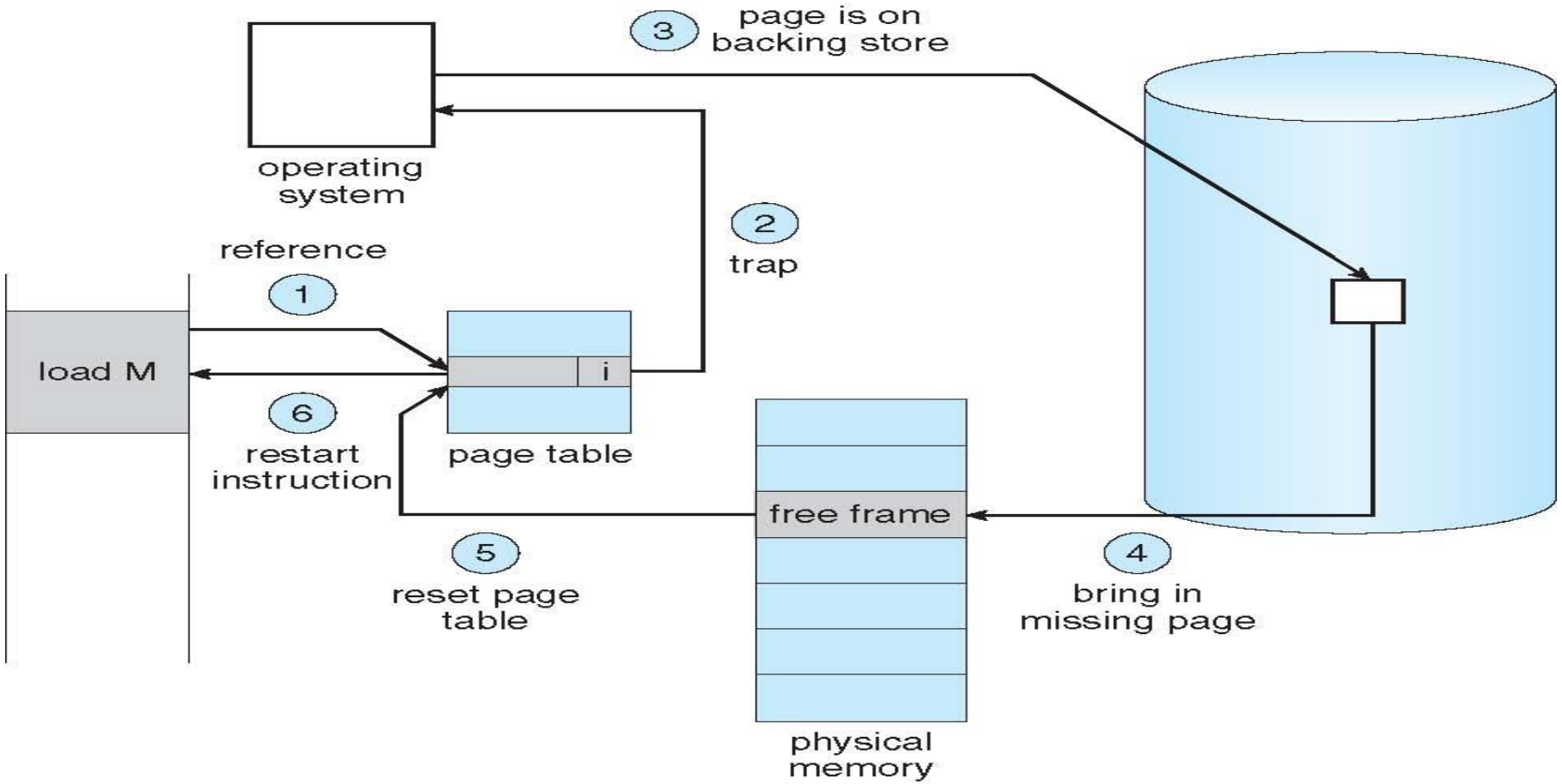


Figure 9.6 : Steps in handling a page fault.

Performance of Demand Paging

Let **p** be the probability of a page fault (0 <= p <= 1). We would expect p to be close to zero,that is, we would expect to have only a few page faults. The effective access time is then.

Effective access time(EAT) = (1 - p) x memory access time(ma) + p x page fault time(overhead).

(1-p)->Probability of no occurrence of page fault.

Page fault overhead-Swap page out, swap page in and Restart overhead.

Given ma=200,page fault time=8ms=8000000,if 1 access out of 1000 causes a page fault then p=1/1000=0.001

EAT=(1-P)*200+P*8000000

EAT=(1-0.001)*200+0.001*8000000=199.8+8000=8199.8=82Micro second.

Copy-on-Write

However, process creation using the fork () system call may initially bypass the need for demand paging by using a technique similar to page sharing.

Recall that the fork() system call creates a child process as a duplicate of its parent. Traditionally, fork() worked by creating a copy of the parent's address space for the child, duplicating the pages belonging to the parent. However, considering that many child processes invoke the exec() system call immediately after creation, the copying of the parent's address space may be unnecessary.

Alternatively, we can use a technique known as copy-an-write, which works by allowing the parent and child processes initially to share the same pages. These shared pages are marked as copy-an-write pages, meaning that if either process writes to a shared page, a copy of the shared page is created.

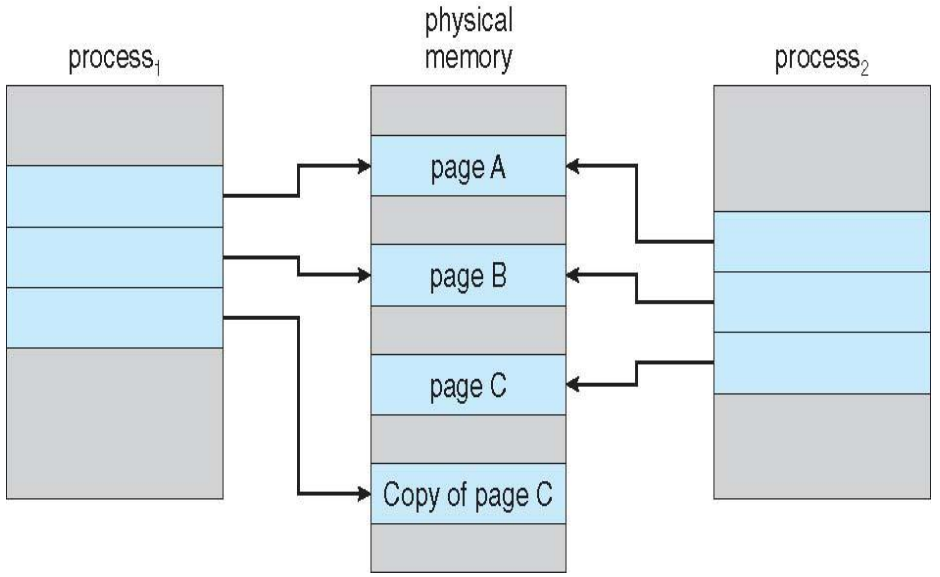


Figure 9.8 After process 1 modifies page C.

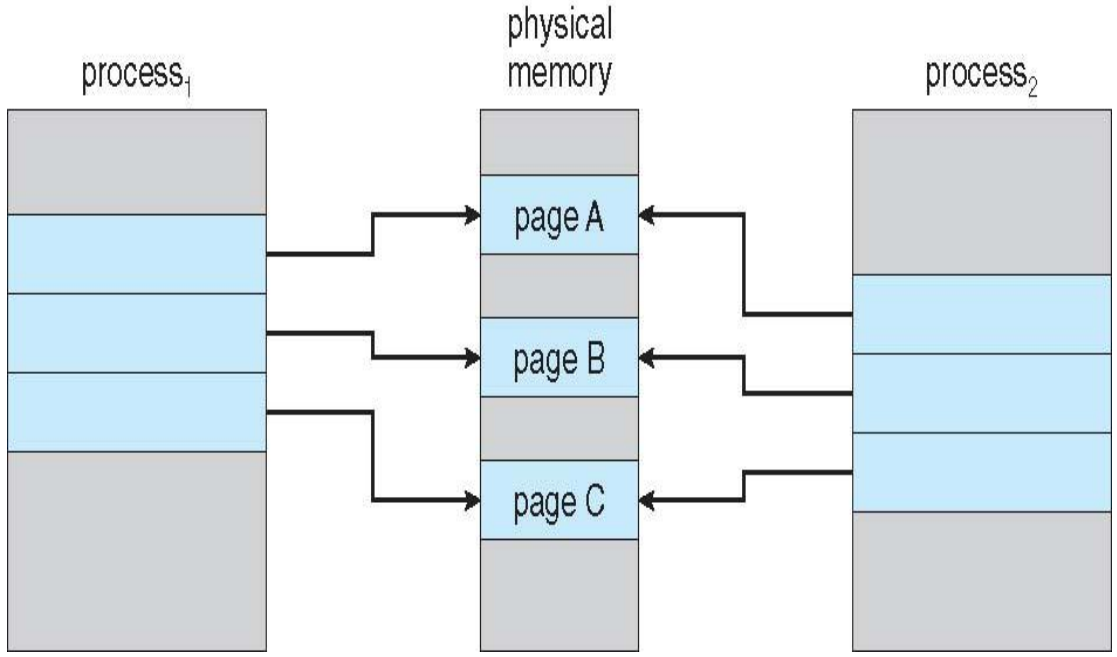


Figure 9.7 Before process 1 modifies page C

Page Replacement

What Happens if There is no Free Frame?

Used up by process pages

Also in demand from the kernel, I/O buffers, etc

How much to allocate to each?

Page replacement – find some page in memory, but not really in use, page it out

- Algorithm – terminate? swap out? replace the page?
- Performance – want an algorithm which will result in minimum number of page faults

Same page may be brought into memory several times

1. Find the location of the desired page on disk

2. Find a free frame:
   - If there is a free frame, use it
   - If there is no free frame, use a page replacement algorithm to select a **victim frame**
     - Write victim frame to disk if dirty

3. Bring the desired page into the (newly) free frame; update the page and frame tables

4. Continue the process by restarting the instruction that caused the trap

Page Replacement Algorithms

FIFO Page Replacement:A FIFO replacement algorithm associates with each page the time when that page was brought into memory. When a page must be replaced, the oldest page is chosen.

Optimal Page Replacement: This algorithms replaces the page which will not be referred for so long in future.

LRU (least-recently-used) **Page Replacement**:This algorithm replaces the page which has not been referred for a long time.

We next illustrate several page-replacement algorithms. In doing so, we use the reference string

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

for a memory with three frames.

Thrashing

In fact, look at any process that does not have "enough" frames. If the process does not have the number of frames it needs to support pages in active use, it will quickly page-fault. At this point, it must replace some page.

However, since all its pages are in active use, it must replace a page that will be needed again right away. Consequently, it quickly faults again, and again, and again, replacing pages that it must bring back in immediately.

This high paging activity is called thrashing. A process is thrashing if it is spending more time paging than executing.

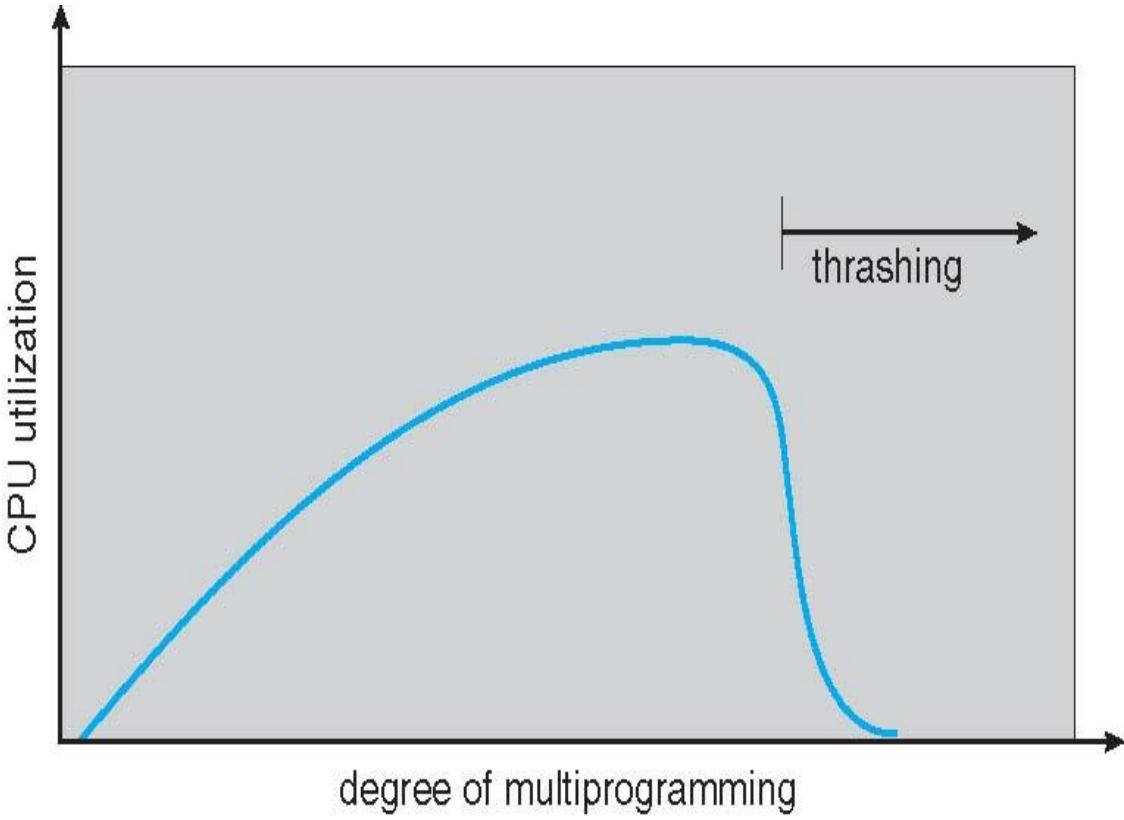


Figure 9.18 Thrashing.

At this point, to increase CPU utilization and stop thrashing, we must decrease the degree of multiprogramming.

File-System Interface

File Concept

Computers can store information on various storage media, such as magnetic disks, magnetic tapes, and optical disks. So that the computer system will be convenient to use, the operating system provides a uniform logical view of information storage.
The operating system abstracts from the physical properties of its storage devices to define a logical storage unit! The file.
Files are mapped by the operating system onto physical devices. These storage devices are usually nonvolatile, so the contents are persistent through power failures and system reboots.
A file is a named collection of related information that is recorded on secondary storage. From a user's perspective, **a file is the smallest allotment of logical secondary storage**; that is, data cannot be written to secondary storage unless they are within a file

File Attributes

Name – only information kept in human-readable form
Identifier – unique tag (number) identifies file within file system
Type – needed for systems that support different types
Location – pointer to file location on device
Size – current file size
Protection – controls who can do reading, writing, executing
Time, date, and user identification – data for protection, security, and usage monitoring
Information about files are kept in the directory structure, which is maintained on the disk

File Operations

File is an **abstract data type**
Create
Write – at **write pointer** location
Read – at **read pointer** location
Reposition within file - seek
Delete
Truncate-The user may want to erase the contents of a file but keep its attributes. Rather than forcing the user to delete the file and then recreate it
Open(F_i) – search the directory structure on disk for entry ***F_i***, and move the content of entry to memory
Close (F_i) – move the content of entry ***F_i*** in memory to directory structure on disk

File Types – Name, Extension

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

Figure 10.2 Common file types.

Internal File Structure

•Some files contain an internal structure, which may or may not be known to the OS. For the OS to support particular file formats increases the size and complexity of the OS.UNIX treats all files as sequences of bytes, with no further consideration of the internal structure.

•Disk files are accessed in units of physical blocks, typically 512 bytes or some power-of-two multiple thereof. (Larger physical disks use larger block sizes, to keep the range of block numbers within the range of a 32-bit integer.)

•Internally files are organized in units of logical units, which may be as small as a single byte, or may be a larger size corresponding to some data record or structure size.

•The number of logical units which fit into one physical block determines its ***packing***, and has an impact on the amount of internal fragmentation (wasted space) that occurs.

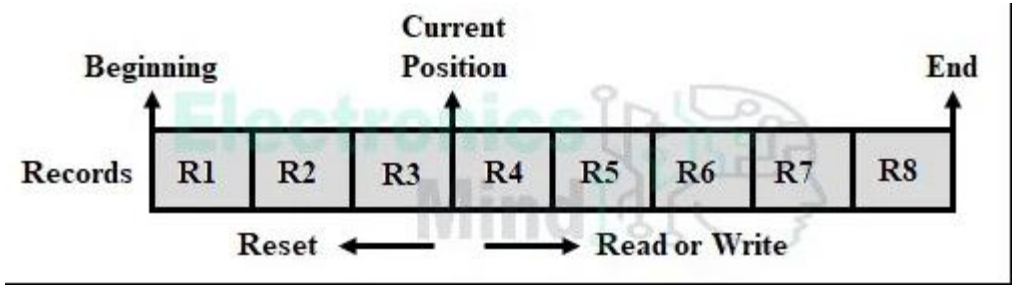
Access Methods

Files store information. When it is used, this information must be accessed and read into computer memory. The information in the file can be accessed in several ways.

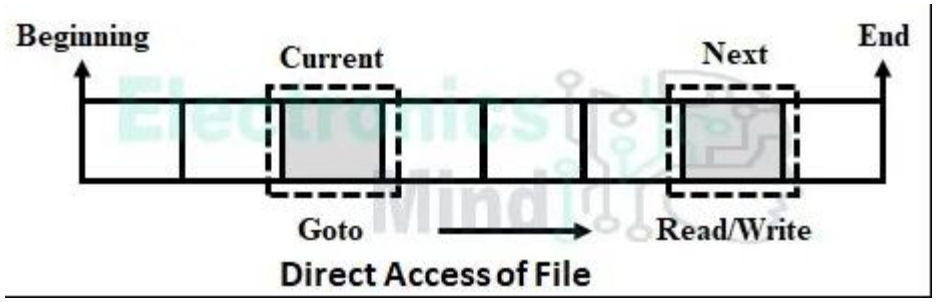
1. **Sequential Access Method**: it is the sequential (or series) processing of information present in a file. Due to its simplicity most of the compilers, editors, etc., use this method.
2. **Direct Access Method**:This access method is also called real-time access where the records can be read irrespective of their sequence.
3. **Indexed sequential access**:An index is created for each file which contains pointers to various blocks. Index is searched sequentially and its pointer is used to access the file directly

Access Methods Continued

Sequential Access

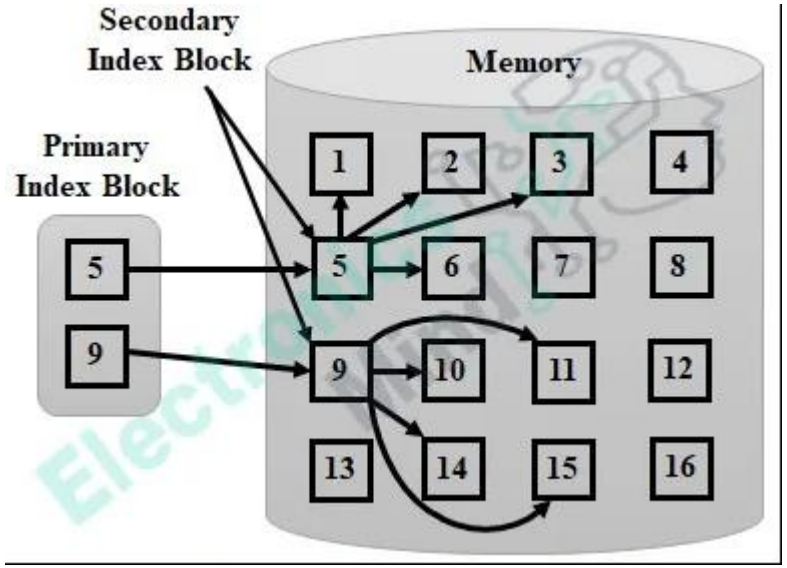


Direct Access



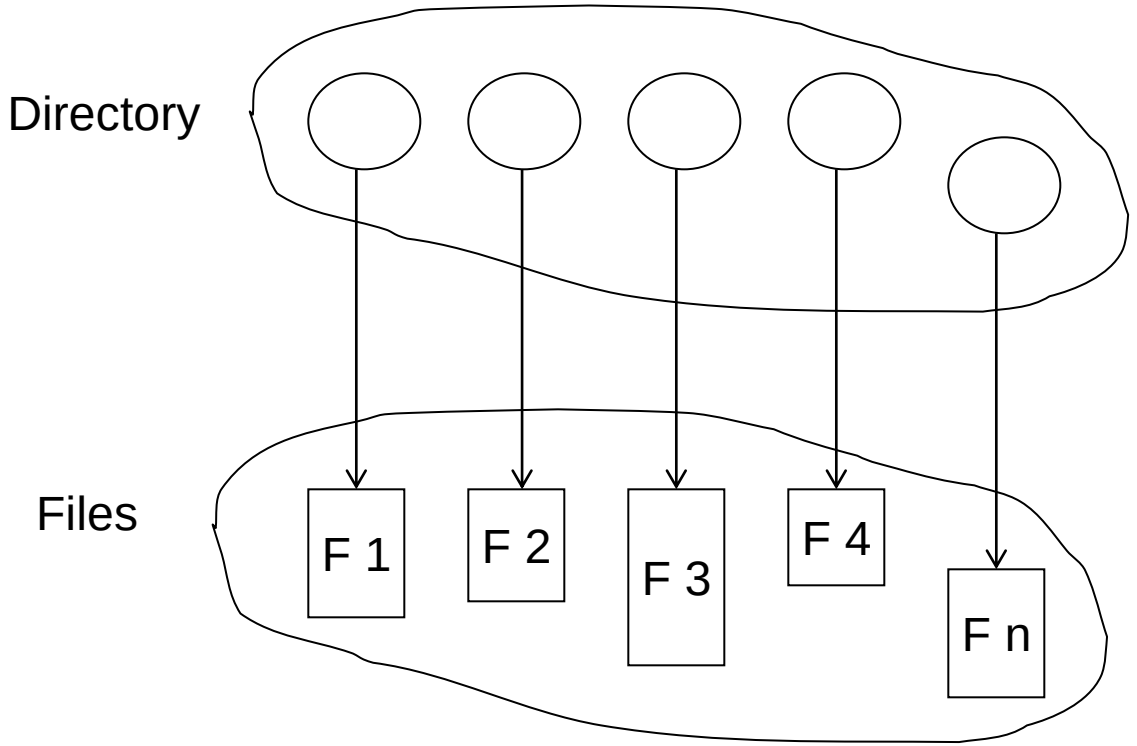
sequential access	implementation for direct access
reset	*cp* = 0;
read next	*read cp*; *cp* = *cp* + 1;
write next	*write cp*; *cp* = *cp* + 1;

Indexed sequential Access



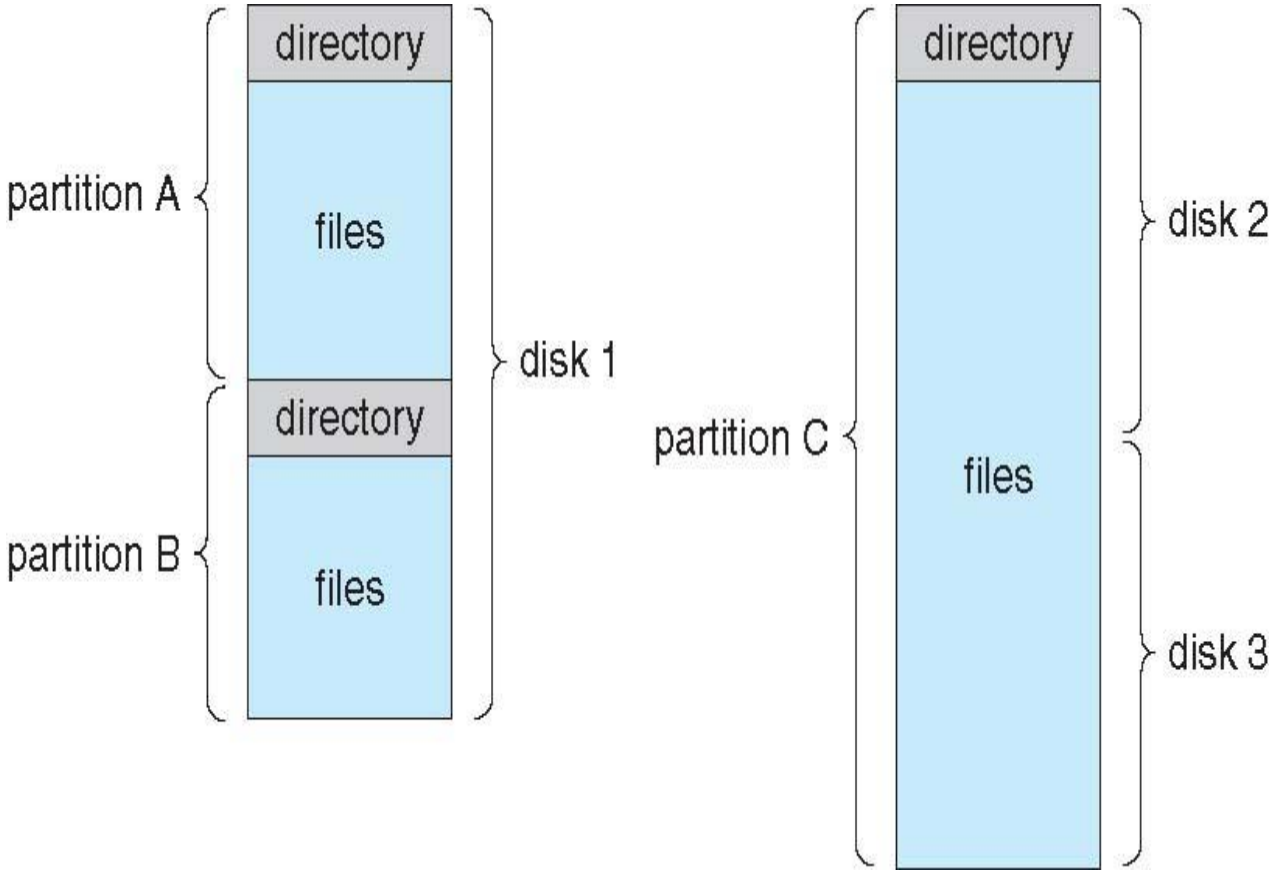
Directory Structure

A collection of nodes containing information about all files



Both the directory structure and the files reside on disk

Storage Structure



Disk can be subdivided into **partitions**
Disks or partitions can be **RAID** protected against failure
Disk or partition can be used **raw** – without a file system, or **formatted** with a file system
Partitions also known as minidisks, slices
Entity containing file system known as a **volume**
Each volume containing file system also tracks that file system's info in **device directory** or **volume table of contents**

Figure 10.6 A typical file-system organization.

Directory Overview

Operations Performed on Directory

Search for a file

Create a file

Delete a file

List a directory

Rename a file

Traverse the file system

Directory Organization

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

Single-Level Directory,

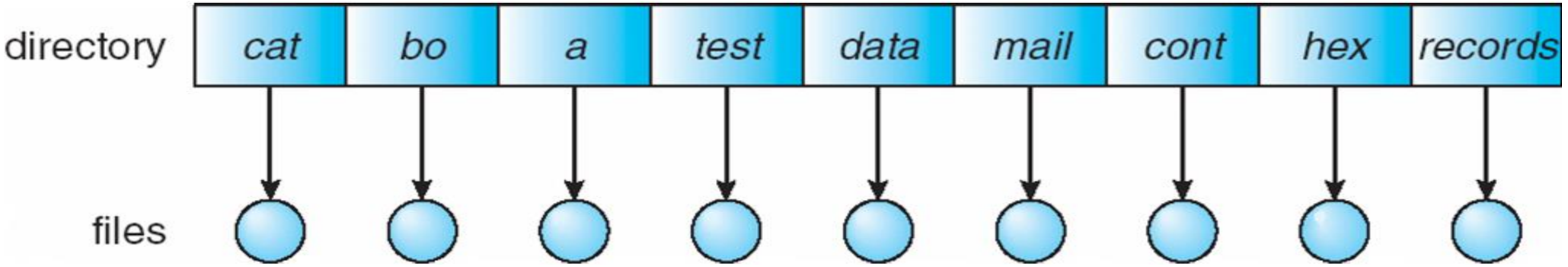


Figure 10.7 Single-level directory

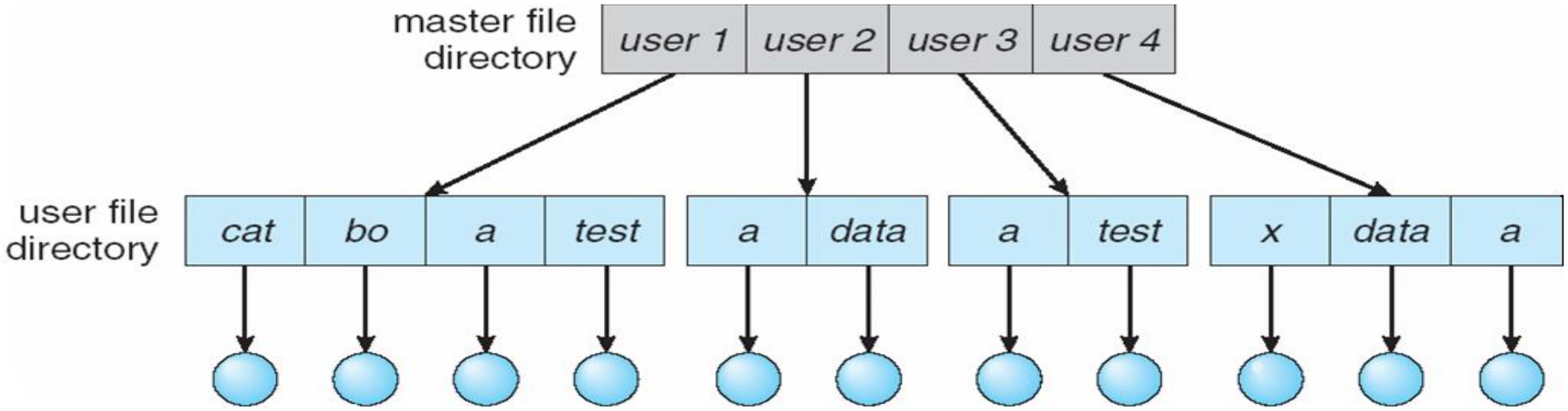


Figure 10.8 Two-level directory structure.

Tree-Structured Directories

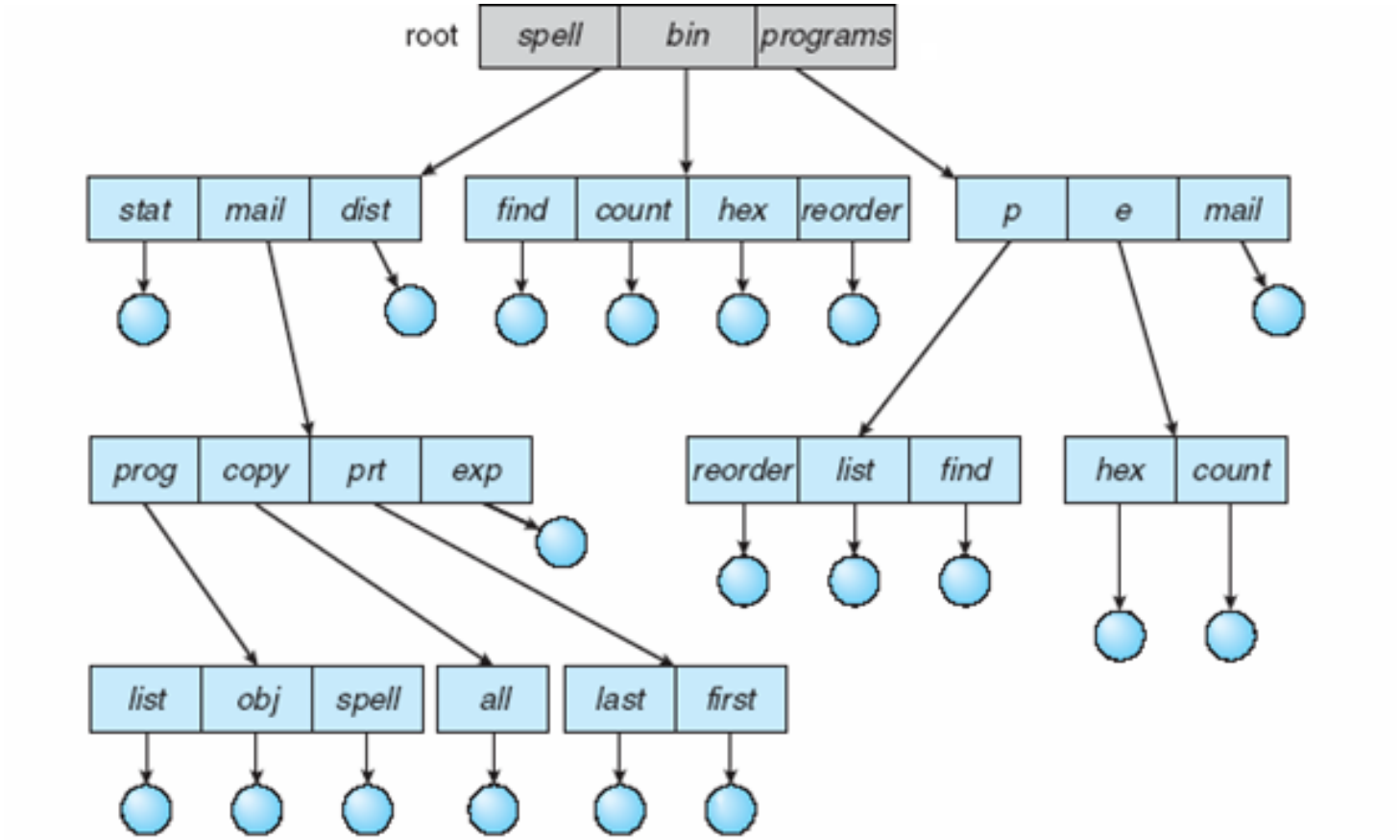


Figure 10.9 Tree-structured directory structure

Note: **Homework:Acyclic-graph directory structure and General Graph Directory**

File-System Mounting

Just as a file must be opened before it is used, a file system must be mounted before it can be available to processes on the system. More specifically, the directory structure can be built out of multiple volumes, which must be mounted to make them available within the file-system name space.

The mount procedure is straightforward. The operating system is given the name of the device and the mount point-the location within the file structure where the file system is to be attached. Typically, a mount point is an empty directory.

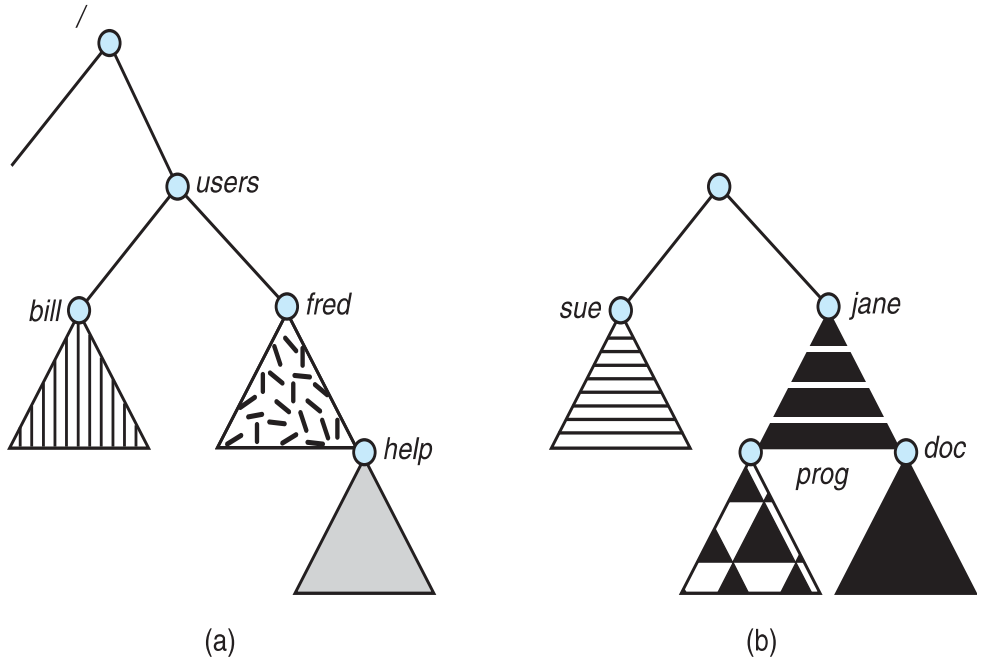


Figure 10.12 File system. (a) Existing system. (b) Unmounted volume.

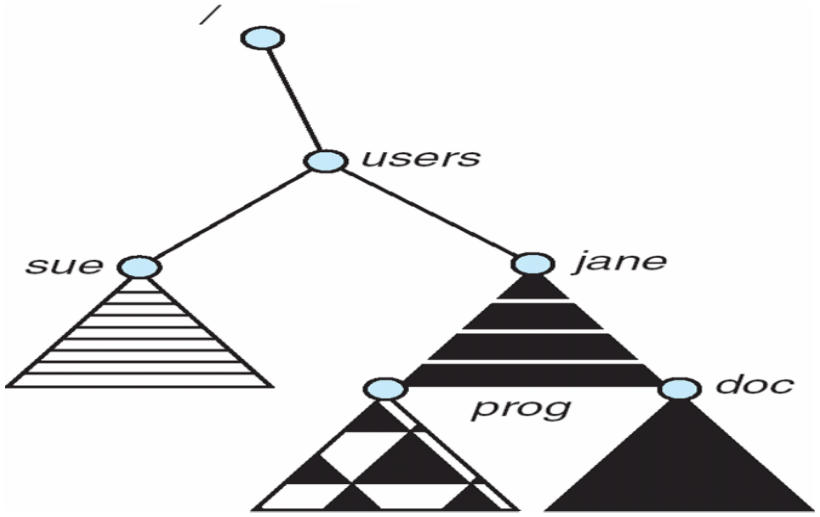


Figure 10.13 Mount point.

File Sharing and Protection

File Sharing

Sharing of files on multi-user systems is desirable

Sharing may be done through a **protection** scheme

On distributed systems, files may be shared across a network

Uses networking to allow file system access between systems

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

NFS is standard UNIX client-server file sharing protocol

CIFS is standard Windows protocol

Protection

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

File-System Structure

Disks provide the bulk of secondary storage on which a file system is maintained. They have two characteristics that make them a convenient medium for storing multiple files a) A disk can be rewritten in place b) A disk can access directly any given block of information it contains.

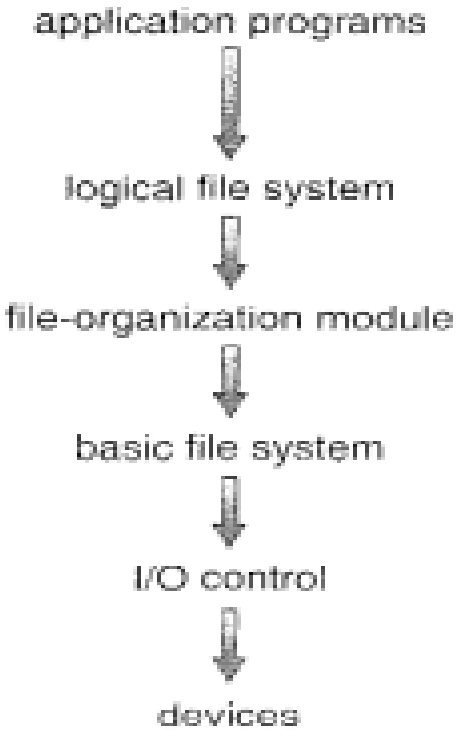


Figure 11.1 Layered file system.

File system Implementation

A boot control block (per volume) can contain information needed by the system to boot an operating system from that volume(In UFS, it is called the boot block; in. NTFS, it is the partition boot sector).

A volume control block (per volume) contains volume (or partition) details, such as the number of blocks in the partition, size of the blocks, freeblock count and free-block pointers, and free FCB count and FCB pointers. In UFS(Unix), this is called a superblock; in NTFS(New Technology), it is stored in the master file table.

A directory structure per file system is used to organize the files. In UFS-Inode Number,In NTFS-master file table.

A per-file FCB contains many details about the file, including file permissions, ownership, size, and location of the data blocks.

File system Implementation

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.

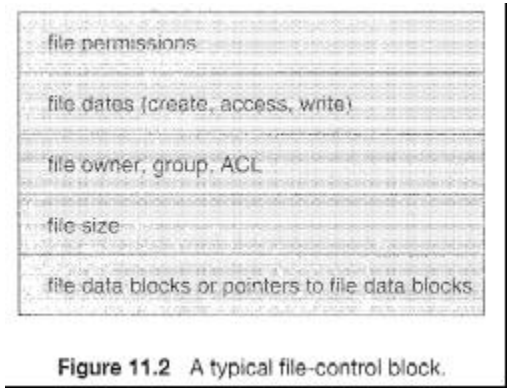


Figure 11.2 A typical file-control block.

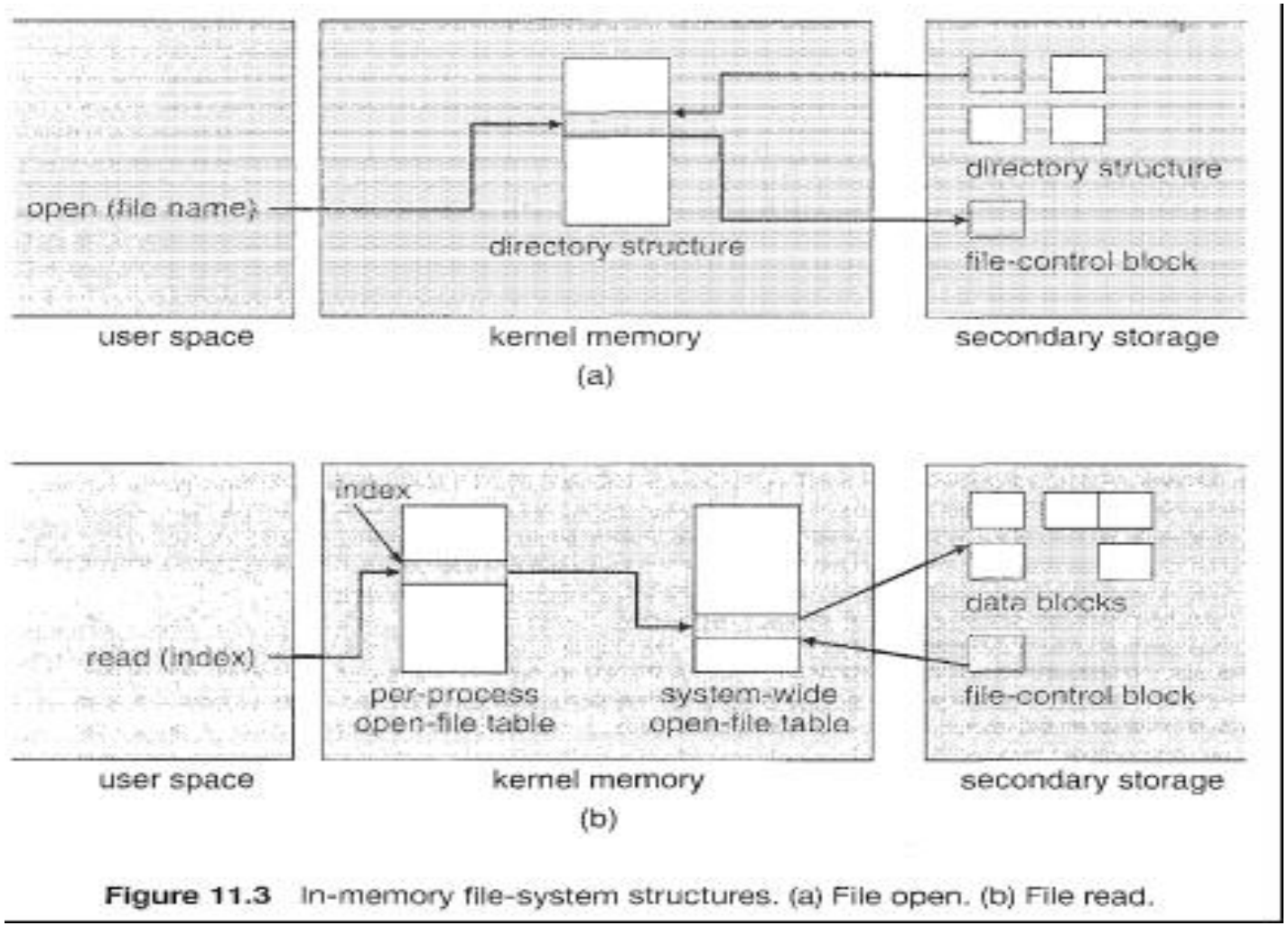


Figure 11.3 In-memory file-system structures. (a) File open. (b) File read.

The textbook

- **Operating System Concepts**, 7th Edition Abraham Silberschatz, Yale University Peter Baer Galvin, Corporate Technologies Greg Gagne, Westminster College
  ISBN: 0-471-69466-5

  http://he-cda.wiley.com/WileyCDA/HigherEdTitle/productCd-0471694665.html