UNIT-IV
Memory Management & Virtual Memory Management
& File Systems

Memory management in OS

What do you mean by memory management :

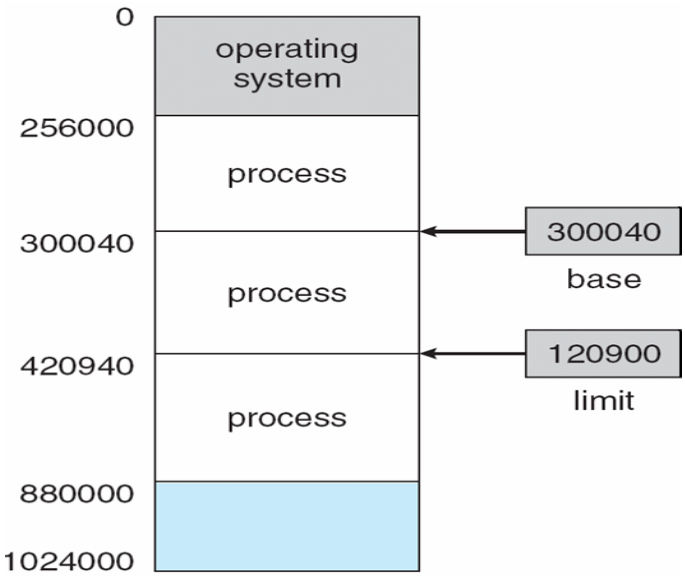
Memory is the important part of the computer that is used to store the data. Its management is critical to the computer system because the amount of main memory available in a computer system is very limited. At any time, many processes are competing for it. Moreover, to increase performance, several processes are executed simultaneously. For this, we must keep several processes in the main memory, so it is even more important to manage them effectively.

- 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. There are machine instructions that take memory addresses as arguments, but none that take disk addresses. Therefore, any instructions in execution, and any data being used by the instructions, must be in one of these direct-access storage devices.
- Memory unit only sees a stream of addresses + read requests, or address + data and write requests
- Registers that are built into the CPU are generally accessible within one cycle of the CPU clock. Most CPUs can decode instructions and perform simple operations on register contents at the rate of one or more operations per clock tick.
- The same cannot be said of main memory, which is accessed via a transaction on the memory bus. Memory access may take many cycles of the CPU clock to complete (processor stalls). Main memory can take many cycles, causing a **stall**
- **Cache** sits between main memory and CPU registers. The remedy is to add fast memory between the CPU and main memory (cache memory). Not only we are concerned with the relative speed of accessing physical memory, but we also must ensure correct operation has to protect the OS from access by user processes and, in addition, to protect user processes from one another.
- Protection of memory required to ensure correct operation

Base and Limit Registers:

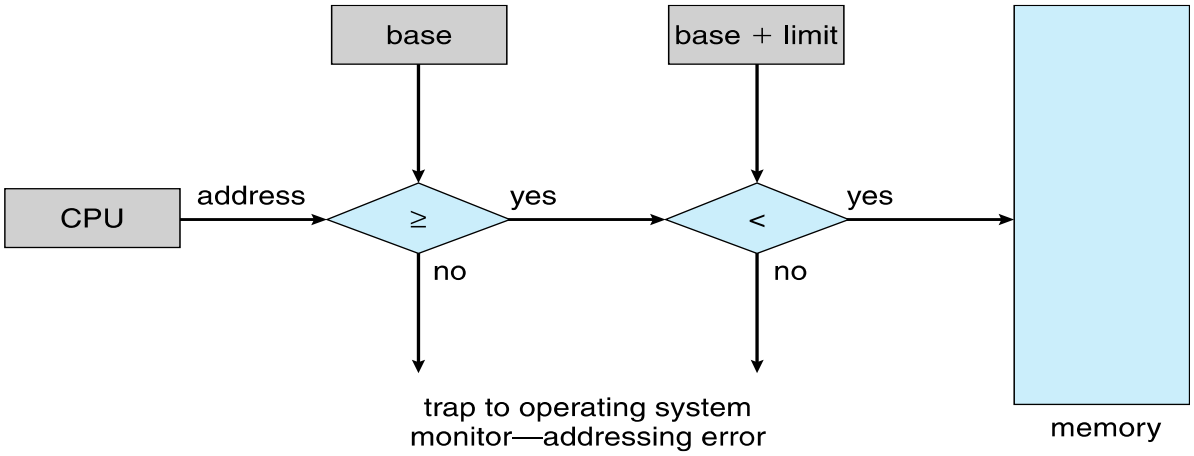
A base and limit register defines a logical address space.

- A pair of **base** and **limit registers** define the logical address space
- CPU must check every memory access generated in user mode to be sure it is between base and limit for that user



This protection must be provided by the hardware. We first need to make sure that each process has a separate memory space.

- We can provide this protection by using two registers, usually a base and a limit.
- 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 holds 300040 and limit register is 120900, then the program can legally access all addresses from 300040 through 420940 (inclusive).
- 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 OS, which treats the attempt as a fatal error.
- This scheme prevents a user program from (accidentally or deliberately) modifying the code or data structures of either the OS or other users.
- Hardware address protection with base and limit registers



Address space:

- Address uniquely identifies a location in the memory.
- We have two types of addresses that are logical address and physical address.
- The logical address is a virtual address and can be viewed by the user. The user can't view the physical address directly.
- The logical address is used like a reference, to access the physical address.
- The fundamental difference between logical and physical address is that logical address is generated by CPU during a program execution whereas, the physical address refers to a location in the memory unit.
- **Logical Address**: It is the virtual address generated by CPU.
- **Physical address:** The physical address is a location in a memory unit.

What is meant by Address Binding? Discuss briefly

- The process of associating program instructions and data to physical memory addresses is called address binding, or relocation.
- A user program will go through several steps -some of which may be optional-before being executed

Addresses may be represented in different ways during these steps.:

- Programs on disk, ready to be brought into memory to execute form an **input queue**
  - Without support, must be loaded into address 0000
- Inconvenient to have first user process physical address always at 0000
  - How can it not be?
- Further, addresses represented in different ways at different stages of a program's life
  - Source code addresses usually symbolic
  - Compiled code addresses **bind** to relocatable addresses
    - i.e. "14 bytes from beginning of this module"
  - Linker or loader will bind relocatable addresses to absolute addresses
    - i.e. 74014
  - Each binding maps one address space to another

Binding of Instructions and Data to Memory:

- Address binding is the process of mapping from one address space to another address space. Logical address is an address generated by the CPU during execution, whereas Physical Address refers to the location in the memory unit(the one that is loaded into memory).
- Address binding of instructions and data to memory addresses can happen at three different stages
- **Compile time**:

  The compiler translates symbolic addresses to absolute addresses. If you know at compile time where the process will reside in memory, then absolute

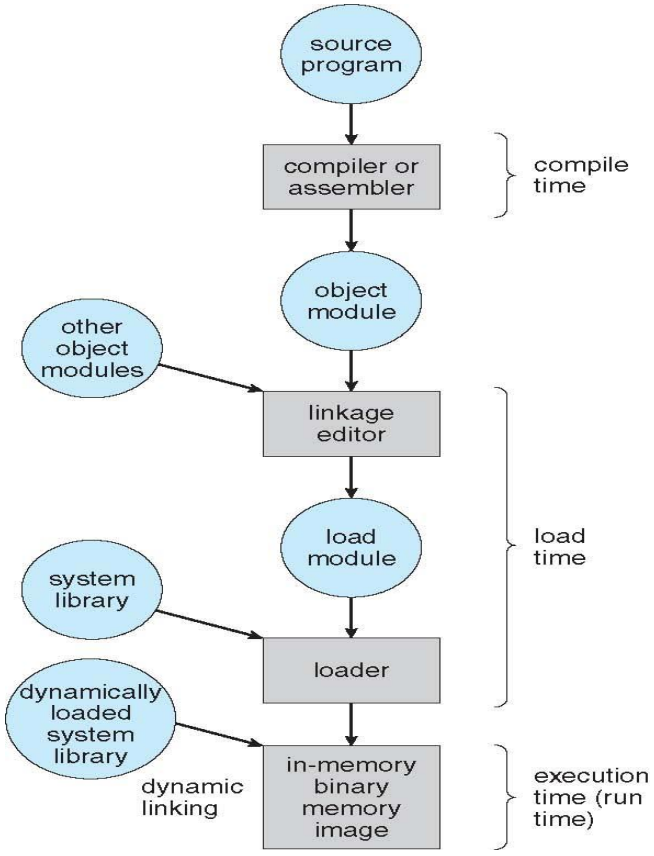
code can be generated (Static). It must recompile code if starting location changes

- **Load time**:
  - The compiler translates symbolic addresses to relative (relocatable) addresses. The loader translates these to absolute addresses. If it is not known at compile time where the process will reside in memory, then the compiler must generate relocatable code (Static). Must generate **relocatable code** if memory location is not known at compile time
- **Execution time or Run time**:

  If the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time. The absolute addresses are generated by hardware. Most general-purpose OSs use this method (Dynamic). Static-new locations are determined before execution. Dynamic-new locations are determined during execution.

  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)

Multistep Processing of a User Program:



In most cases, a user program will go through several steps-some of which may be optional-before being executed (Figure) Addresses may be represented in different ways during these steps.

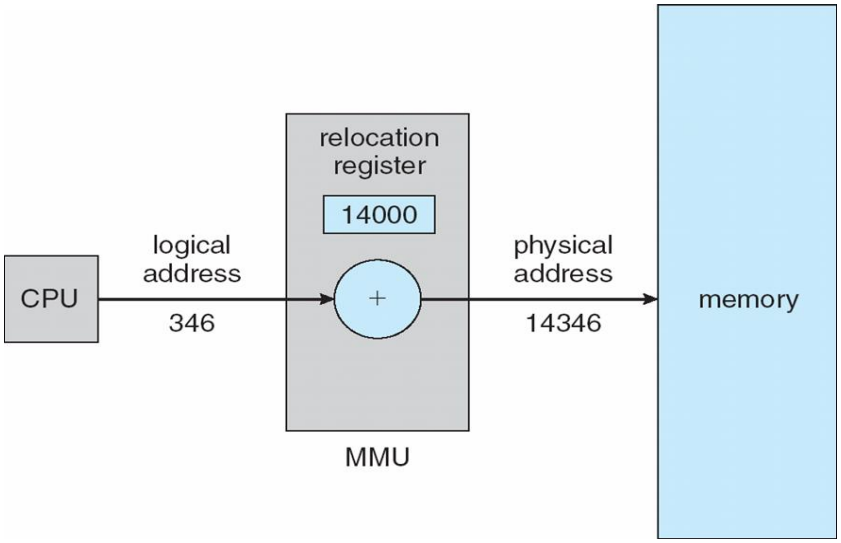
Logical vs. Physical Address Space:

- The concept of a logical address space that is bound to a separate **physical address space** is central to proper memory management
  - **Logical address** – 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. In this case, we usually refer to the logical address as a virtual address.
- **Logical address space** is the set of all logical addresses generated by a program
- **Physical address space** is the set of all physical addresses generated by a program

Memory-Management Unit (MMU):

- The run-time mapping from virtual to physical addresses is done by a hardware device called the Memory management unit.
- To start, consider simple scheme where the value in the relocation register is added to every address generated by a user process at the time it is sent to memory
  - Base register now called **relocation register.**
  - The value in the relocation register is added to every address generated by a user process at the time the address is sent to memory
  - MS-DOS on Intel 80x86 used 4 relocation registers
- The user program deals with *logical* addresses; it never sees the *real* physical addresses
  - Execution-time binding occurs when reference is made to location in memory
  - Logical address bound to physical addresses

Dynamic relocation using a relocation register:



Dynamic Loading:

- It has been necessary for the entire program and all data of a process to be in physical memory for the process to execute. The size of a process has thus been limited to the size of physical memory. To obtain better memory-space utilization, we can use dynamic loading.
- With Dynamic loading, Routine is not loaded until it is called
- Better memory-space utilization; unused routine is never loaded
- No special support from the operating system is required
- Implemented through program design
- OS can help by providing libraries to implement dynamic loading
- All routines are kept on disk in a relocatable load format.
- The main program is loaded into memory and is executed.
- When a routine needs to call another routine, the calling routine first checks to see whether the other routine has been loaded.
- If not, the relocatable linking loader is called to load the desired routine into memory and to update the program's address tables to reflect this change.
- Then control is passed to the newly loaded routine.
- The advantage of dynamic loading is that an unused routine is never loaded

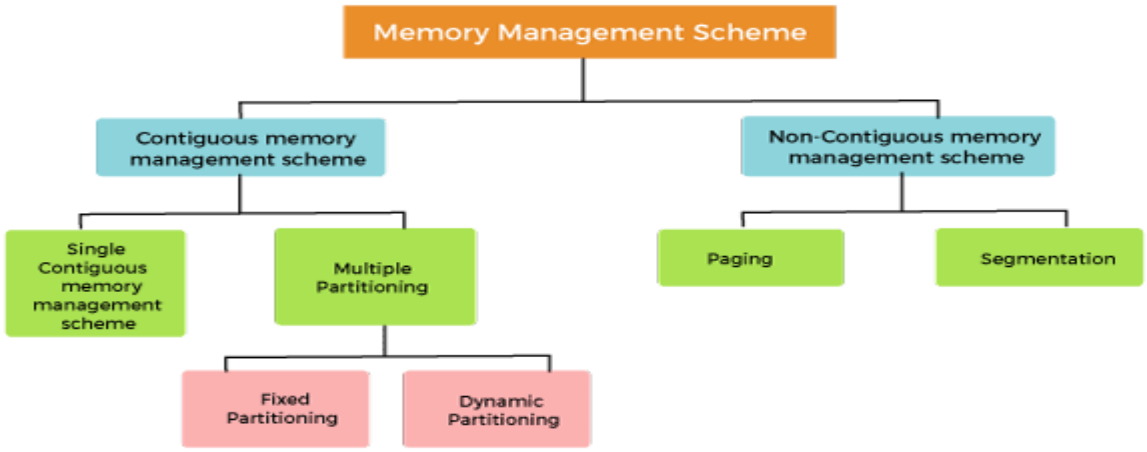
Dynamic linking:

- **Static linking** – system libraries and program code combined by the loader into the binary program image
- Dynamic linking, in contrast, is similar to dynamic loading.
- Dynamic linking is postponed until execution time
- With dynamic linking, a stub is included in the image for each library-routine reference.
- The stub is a small piece of code that indicates how to locate the appropriate memory-resident library routine or how to load the library if the routine is not already present.

- When the stub is executed, it checks to see whether the needed routine is already in memory.
- If not, the program loads the routine into memory.
- This feature can be extended to library updates (such as bug fixes). A library may be replaced by a new version, and all programs that reference the library will automatically use the new version.
- Dynamic linking is particularly useful for libraries
- System also known as **shared libraries**

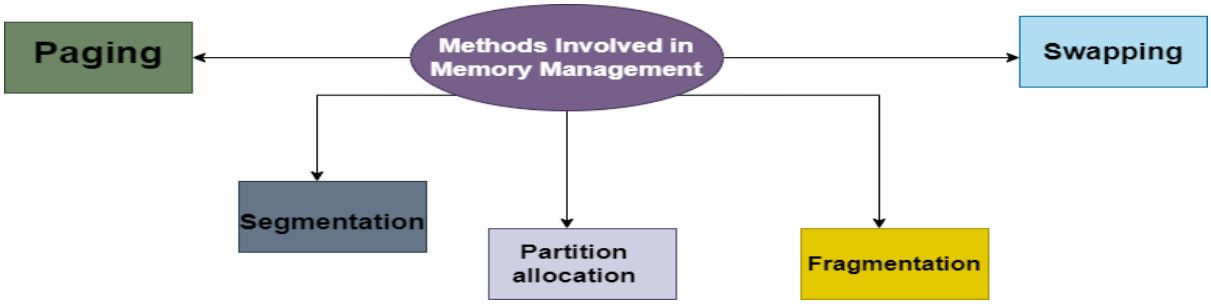
Memory management Techniques: : **can be classified into following main categories:**

1.Contiguous memory management schemes



Classification of memory management schemes

2.Non-Contiguous memory management schemes:



Contiguous memory management schemes:

In a Contiguous memory management scheme, each program occupies a single contiguous block of storage locations, i.e., a set of memory locations with consecutive addresses.

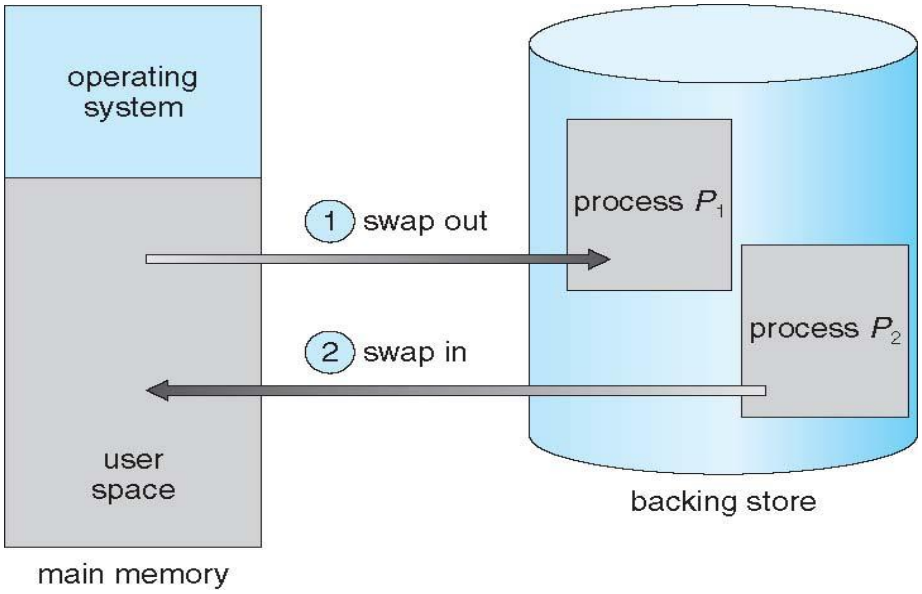
Non-Contiguous memory management schemes:

In a Non-Contiguous memory management scheme, the program is divided into different blocks and loaded at different portions of the memory that need not necessarily be adjacent to one another. This scheme can be classified depending upon the size of blocks and whether the blocks reside in the main memory or not.

Swapping in memory management.:

- Swapping is a mechanism in which a process can be swapped temporarily out of main memory (or move) to secondary storage (disk) and make that memory available to other processes. At some later time, the system swaps back the process from the secondary storage to main memory.
- Though performance is usually affected by swapping process but it helps in running multiple and big processes in parallel and that's the reason Swapping is also known as a technique for memory compaction.
- A process must be in memory to be executed
- 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
    - In Round robin Algorithm, swapping can be seen.
- The memory manager will start to swap out the process that just finished and to swap in another process into the memory space that has been freed. In the meantime, the CPU scheduler will allocate a time slice to some other process in memory. When each process finishes its quantum, it will be swapped with another process.
- **Backing store** is a usually a hard disk drive or any other secondary storage which fast in access and large enough to accommodate copies of all memory images for all users
- **Roll out, roll in** – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so that higher-priority process can be loaded and executed
- Major time consuming part of swapping 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

Swapping:



Contiguous Memory Allocation:

- In a Contiguous memory management scheme, each program occupies a single contiguous block of storage locations, i.e., a set of memory locations with consecutive addresses.

The memory is usually divided into two partitions:

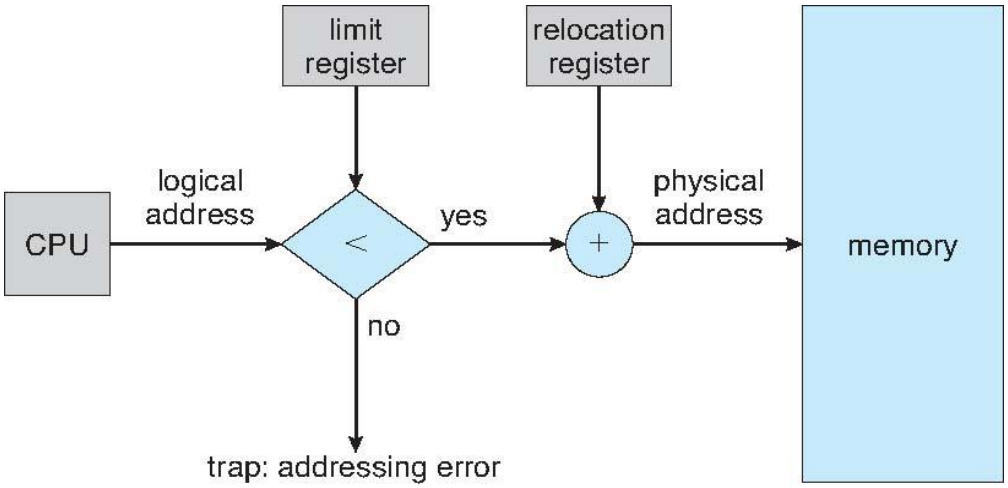
- one for the resident OS
- one for the user processes.
- We can place the OS in either low memory or high memory (depends on the location of the interrupt vector).
- We usually want several user processes to reside in memory at the same time. We therefore need to consider how to allocate available memory 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

- With relocation and limit registers, each logical address must be less than the limit register;
- The MMU maps the logical address dynamically by adding the value in the relocation register. This mapped address is sent to memory
- 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

Hardware Support for Relocation and Limit Registers:



- When the CPU scheduler selects a process for execution, the dispatcher loads the relocation and limit registers with the correct values as part of the context switch.
- The relocation-register scheme provides an effective way to allow the OS size to change dynamically.

Memory Allocation

- One of the simplest methods for allocating memory is to divide memory into several fixed-sized partitions. Each partition may contain exactly one process.
- Thus, the degree of multiprogramming is bound by the number of partitions.

Single partitioning

- The Single contiguous memory management scheme is the simplest memory management scheme used in the earliest generation of computer systems. In this scheme, the main memory is divided into two contiguous areas or partitions. The operating systems reside permanently in one partition, generally at the lower memory, and the user process is loaded into the other partition.

Advantages of Single contiguous memory management schemes:

1.Simple to implement.

2.Easy to manage and design.

- **Disadvantages of Single contiguous memory management schemes:**

1. Wastage of memory space
2. The CPU remains idle,
3. It does not support multiprogramming

Multiple Partitioning (or) Multiple-partition method

- The single Contiguous memory management scheme is inefficient as it limits computers to execute only one program at a time resulting in wastage in memory space and CPU time. The problem of inefficient CPU use can be overcome using multiprogramming that allows more than one program to run concurrently. To switch between two processes, the operating systems need to load both processes into the main memory.
- The operating system needs to divide the available main memory into multiple parts to load multiple processes into the main memory. Thus multiple processes can reside in the main memory simultaneously.
- The multiple partitioning schemes can be of two types:
- 1.Fixed Partitioning
- 2.Dynamic Partitioning

1.Fixed-size Partition Scheme(MFT)

- Fixed Partitioning
- The main memory is divided into several fixed-sized partitions in a fixed partition memory management scheme or static partitioning. These partitions can be of the same size or different sizes. Each partition can hold a single process. The number of partitions determines the degree of multiprogramming, i.e., the maximum number of processes in memory. These partitions are made at the time of system generation and remain fixed after that.

Advantages of Fixed Partitioning memory management schemes:

- 1.Simple to implement.
- 2.Easy to manage and design.

Disadvantages of Fixed Partitioning memory management schemes:

- 1.This scheme suffers from internal fragmentation.
  2.The number of partitions is specified at the time of system generation.

2. Variable-size Partition Scheme(MVT)

- This scheme is also known as Dynamic partitioning and is came into existence to overcome the drawback i.e internal fragmentation that is caused by Static partitioning. In this partitioning, scheme allocation is done dynamically.
- The dynamic partitioning was designed to overcome the problems of a fixed partitioning scheme. In a dynamic partitioning scheme, each process occupies only as much memory as they require when loaded for processing. Requested processes are allocated memory until the entire physical memory is exhausted or the remaining space is insufficient to hold the requesting process. In this scheme the partitions used are of variable size, and the number of partitions is not defined at the system generation time.

Advantages of Dynamic Partitioning memory management schemes:

- 1.Simple to implement.
- 2.Easy to manage and design.

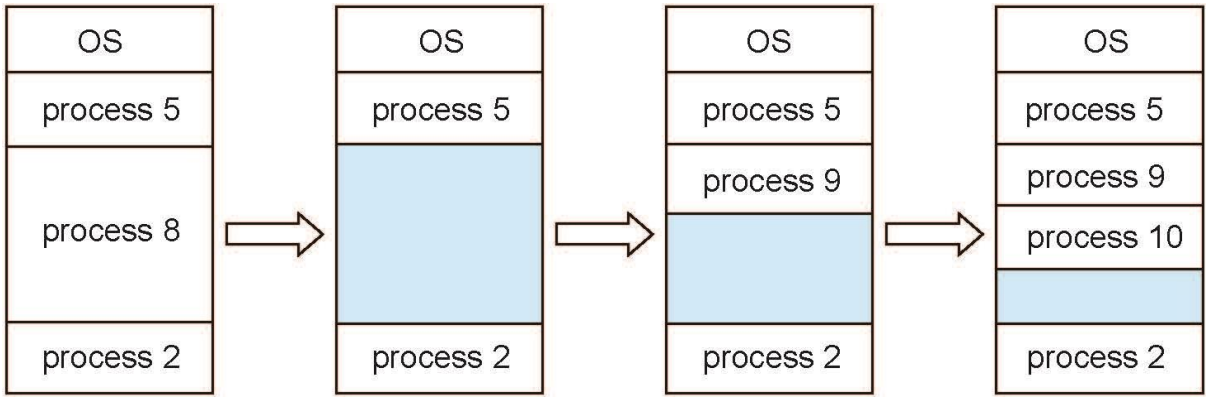
Disadvantages of Dynamic Partitioning memory management schemes:

- 1.This scheme also suffers from fragmentation.

- 2.The number of partitions is specified at the time of system segmentation.

Multiple-partition allocation

- Degree of multiprogramming limited by number of partitions
- **Variable-partition** sizes for efficiency (sized to a given process' needs)
- **Hole** – block of available memory; holes of various size are scattered throughout memory
- When a process arrives, it is allocated memory from a hole large enough to accommodate it
- Process exiting frees its partition, adjacent free partitions combined
- Operating system maintains information about:
  a) allocated partitions b) free partitions (hole)



Dynamic Storage-Allocation Problem:

- The OS keeps a table indicating which parts of memory are available and which are occupied.
- Initially, all memory is available for user processes and is considered one large block of available memory, a hole.
- When a process arrives and needs memory, we search for a hole large enough for this process.
- If we find one, we allocate only as much memory as is needed, keeping the rest available to satisfy future requests.
- At any given time, we have a list of available block sizes and the input queue. The OS can order the input queue according to a scheduling algorithm.
- When a process terminates, it releases its block of memory, which is then placed back in the set of holes. If the new hole is adjacent to other holes, these adjacent holes are merged to form one larger hole.
- This procedure is a particular instance of the general **dynamic storage-allocation** problem, which concerns how to satisfy a request of size from a list of free holes. There are many solutions to this problem.
- **First fit**:

Allocate the first hole that is big enough. Searching can start either at the beginning of the set of holes or where the previous first-fit search ended. We can stop searching as soon as we find a free hole that is large enough.

- **Best fit:**
  Allocate the smallest hole that is big enough. We must search the entire list, unless the list is ordered by size. This strategy produces the smallest leftover hole.
- **Worst fit:**
  Allocate the largest hole. Again, we must search the entire list, unless it is sorted by size. This strategy produces the largest leftover hole, which may be more useful than the smaller leftover hole from a best-fit approach.

Fragmentation

- Both the first-fit and best-fit strategies for memory allocation suffer from external fragmentation.
- **External fragmentation** exists when there is enough total memory space to satisfy a request, but the available spaces are not contiguous; storage is fragmented into a large number of small holes.
- **Internal Fragmentation** -Memory block assigned to process is bigger than requested. The difference between these two numbers is internal fragmentation; memory that is internal to a partition but is not being used.
- The general approach to avoiding this problem is to break the physical memory into fixed-sized blocks and allocate memory in units based on block size.
- One solution to the problem of external fragmentation is compaction. The goal is to shuffle the memory contents so as to place all free memory together in one large block.
- The simplest compaction algorithm is to move all processes toward one end of memory; all holes move in the other direction, producing one large hole of available memory. This scheme can be expensive.
- Another possible solution to the external-fragmentation problem is to **permit the logical address space of the processes to be non-contiguous**, thus allowing a process to be allocated physical memory wherever the latter is available.

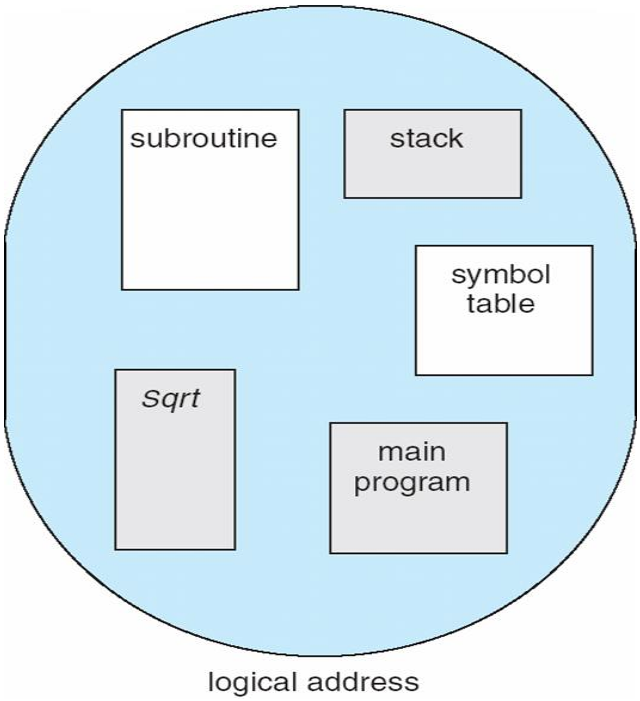
Segmentation.

- An important aspect of memory management that became unavoidable with paging is the separation of the user's view of memory and the actual physical memory.
- The user's view of memory is not the same as the actual physical memory. The user's view is mapped onto physical memory.
- It is a Memory-management scheme that supports user view of memory .
- It is used to show the modules of the program
- A program is a collection of segments

Basic Method

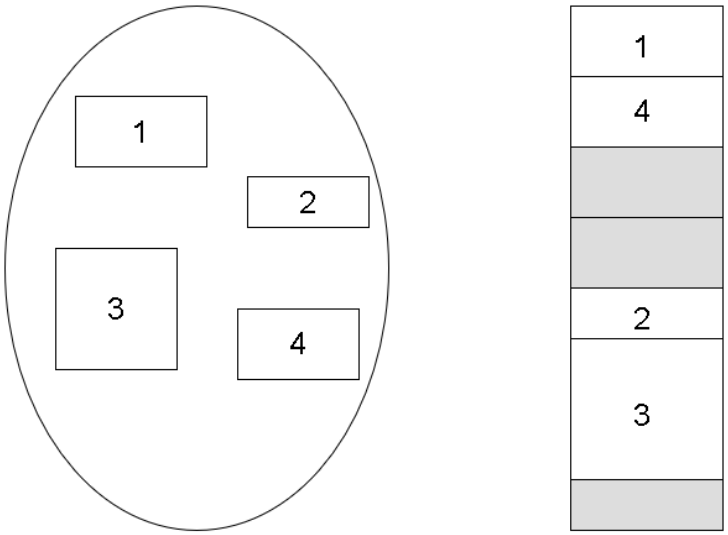
- Users prefer to view memory as a collection of variable-sized segments, with no necessary ordering among segments
- Consider how you think of a program when you are writing it. You think of it as a main program with a set of methods, procedures, or functions.
- Segmentation is a memory-management scheme that supports this user view of memory. A logical address space is a collection of segments.
- Each segment has a name and a length. The addresses specify both the segment name and the offset within the segment.
- The user therefore specifies each address by two quantities:
  - a segment name
  - an offset

User's View of a Program:



Logical View of Segmentation:

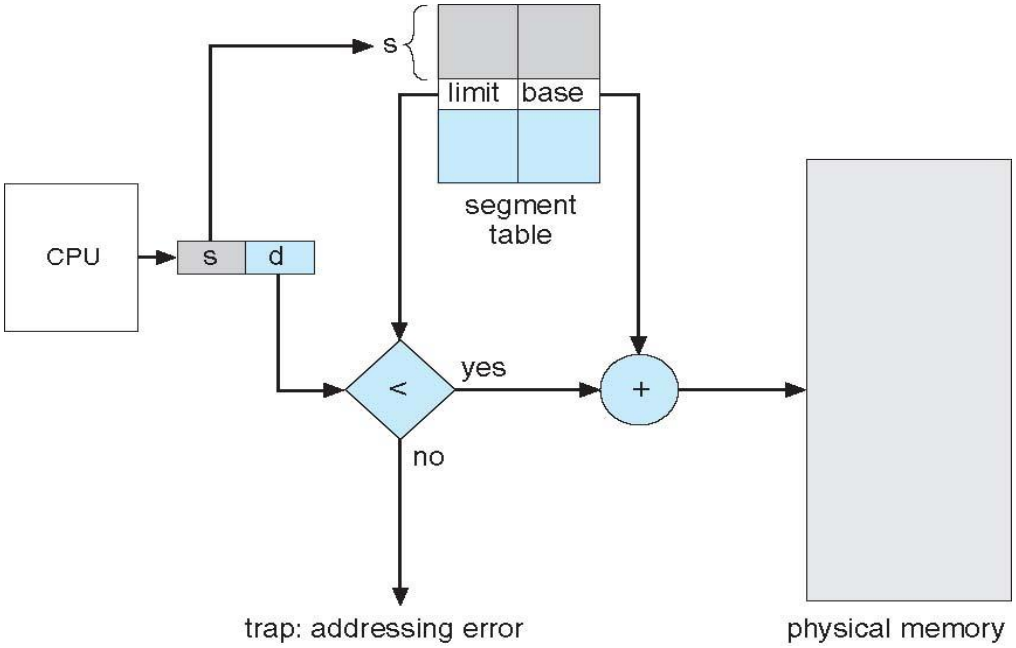
Logical View of Segmentation



Segmentation Architecture:

- For simplicity of implementation, segments are numbered and are referred to by a segment number, rather than by a segment name.
- Thus, a logical address consists of a two tuple:
  <segment-number, offset>,
- **Segment table** – maps two-dimensional physical addresses; each table entry has:
  - **base** – contains the starting physical address where the segments reside in memory
  - **limit** – specifies the length of the segment
- **Segment-table base register (STBR)** points to the segment table's location in memory
- **Segment-table length register (STLR)** indicates number of segments used by a program;
  segment number ***s*** is legal if ***s*** < **STLR**
- Protection
  - With each entry in segment table associate:
    - validation bit = 0 ⇒ illegal segment
    - read/write/execute privileges
- Protection bits associated with segments; code sharing occurs at segment level
- Since segments vary in length, memory allocation is a dynamic storage-allocation problem
- A segmentation example is shown in the following diagram

Segmentation Hardware:



- A logical address consists of two parts: a segment number, “s” and an offset into that segment, “d”
- The segment number is used as an index to the segment table. The offset d of the logical address must be between 0 and the segment limit. If it is not, we trap to the OS (logical addressing attempt beyond end of segment).
- When an offset is legal, it is added to the segment base to produce the address in physical memory of the desired byte. The segment table is thus essentially an array of base-limit register pairs.

Example of Segmentation:

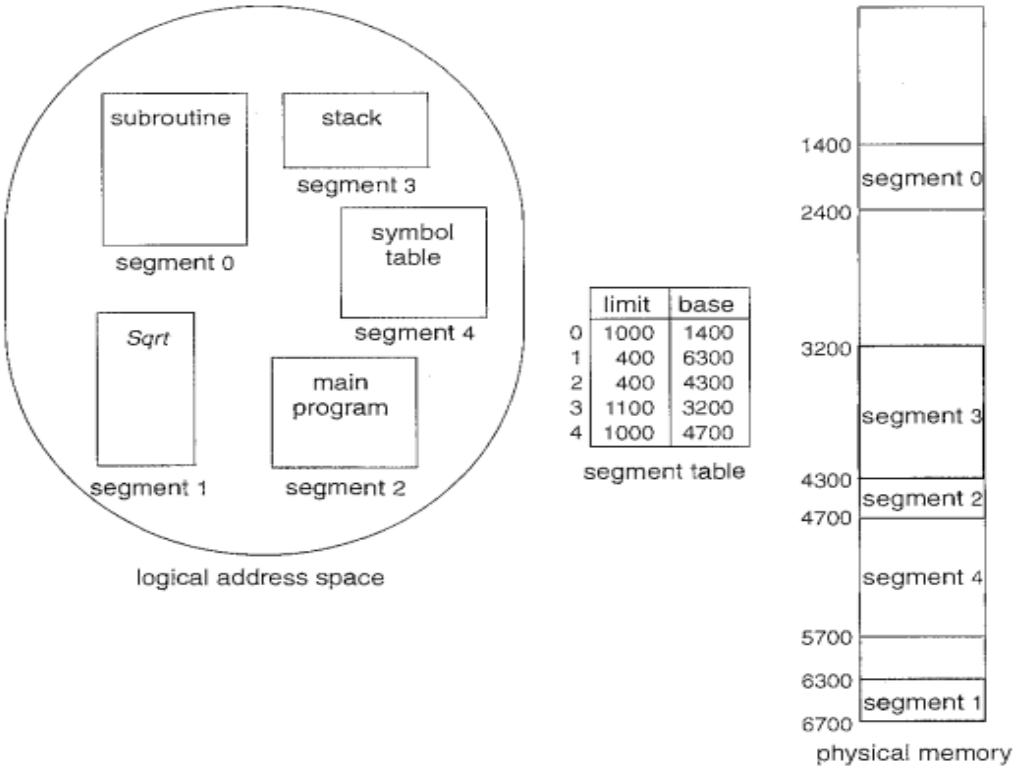


Figure 8.20 Example of segmentation.

- We have five segments numbered from 0 through 4. The segments are stored in physical memory as shown.
- The segment table has a separate entry for each segment, giving the beginning address of the segment in physical memory (or base) and the length of that segment (or limit).
- 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 would result in a trap to the OS, as this segment is only 1,000 bytes long.

Paging:

Paging memory management scheme

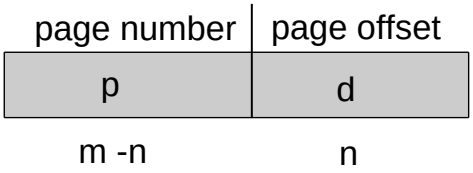
- Paging is a memory-management scheme that permits the physical address space of a process to be non-contiguous.

Basic Method :

- 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.
- In paging, divide physical memory into fixed-sized blocks called **frames**
- Size is power of 2, between 512 bytes and 16 Mbytes
- Divide logical memory into blocks of same size called **pages**
- Keep track of all free frames
- 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.
- Physical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
- Paging avoids external fragmentation
- It also Avoids problem of varying sized memory chunks
- To run a program of size ***N*** pages, need to find ***N*** free frames and load program
- Set up a **page table** to translate logical to physical addresses
- Backing store likewise split into pages
- Still have Internal fragmentation

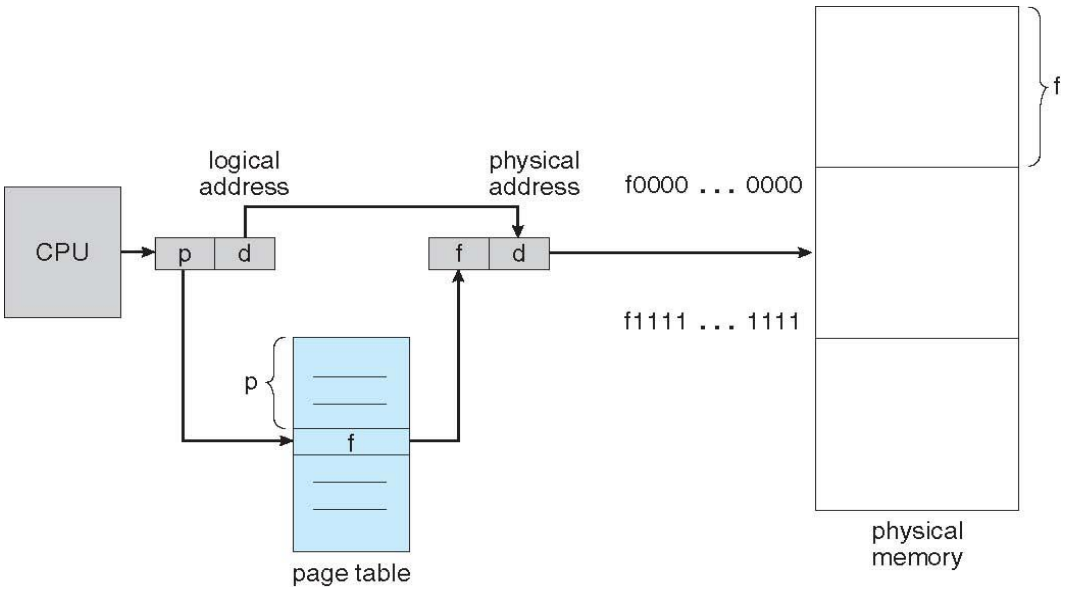
Address Translation Scheme:

- Every address generated by the CPU is divided into two parts: a page number (p) and a page offset (d).
- Address generated by CPU is divided into:
  - **Page number** (***p***) – it is used as an index into a **page table** which contains base address of each page in physical memory
  - **Page offset** (***d***) – combined with base address to define the physical memory address that is sent to the memory unit
  - For given logical address space 2^m and page size 2^n



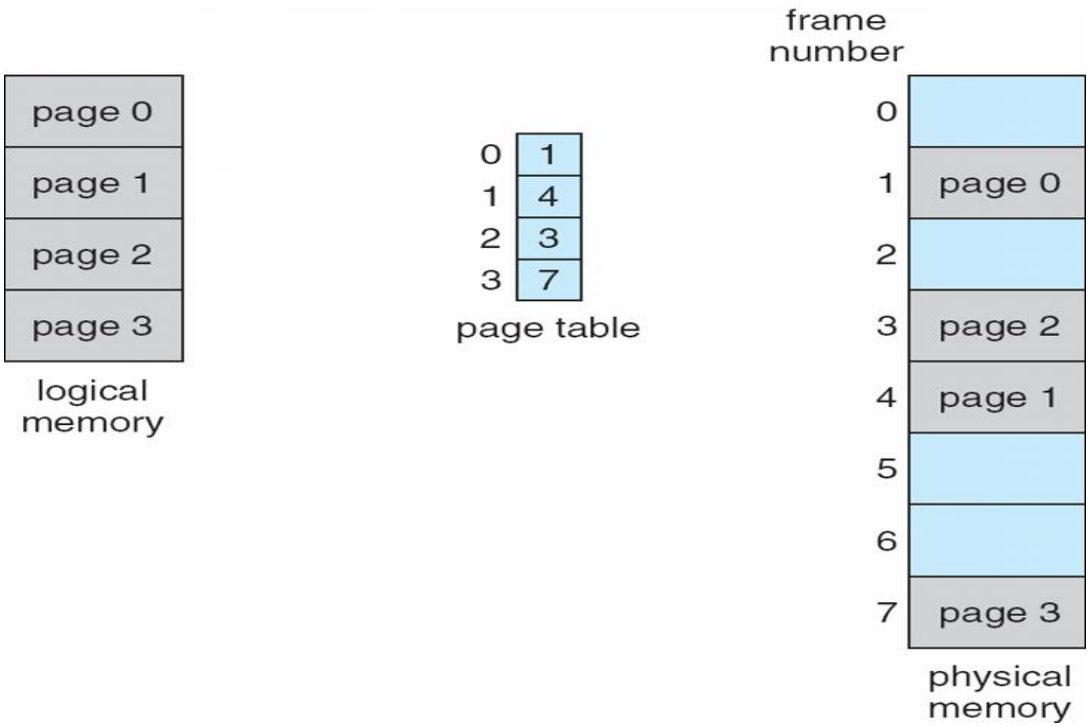
This base address is combined with the page offset to define the physical memory address that is sent to the memory unit.

Paging Hardware:



- 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.

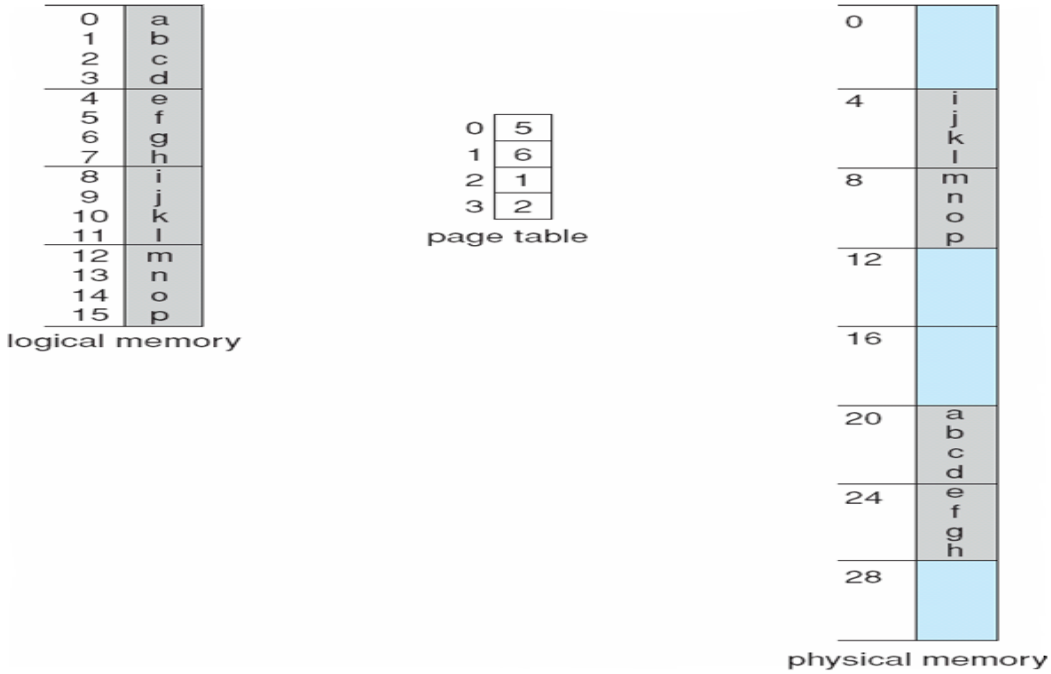
Paging Model of Logical and Physical Memory:



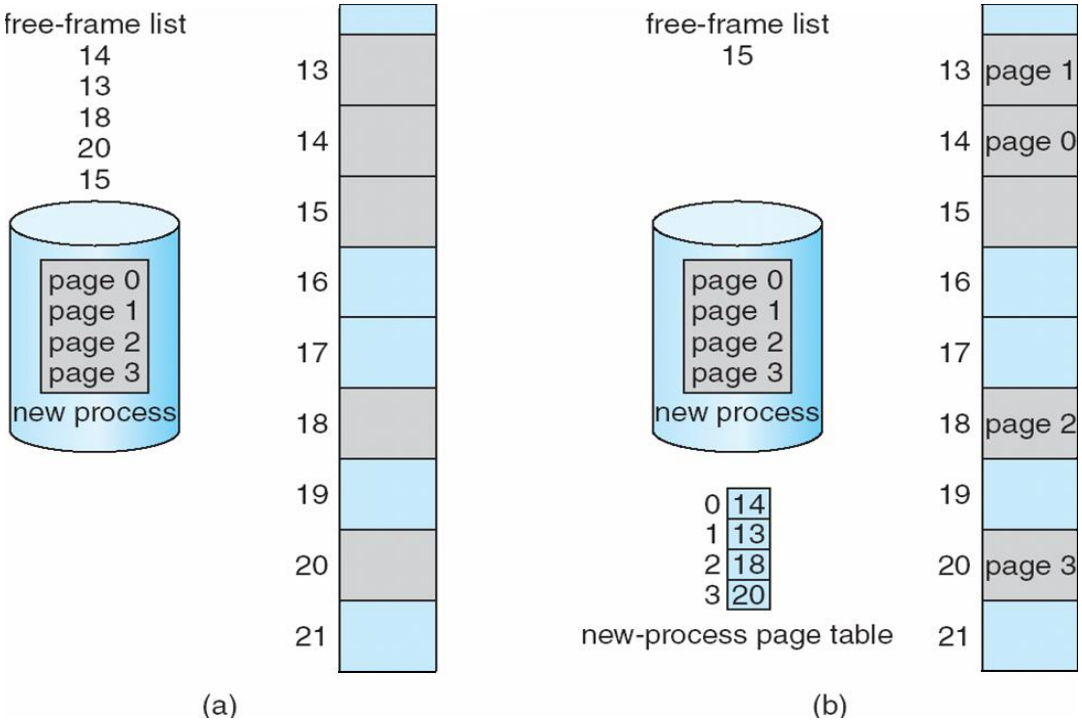
The page size (like the frame size) is defined by the hardware. The size of a page is typically a power of 2, varying between 512 bytes and 16 MB per page, depending on the computer architecture.

Paging Example:

Paging example for a 32-byte memory with 4-byte pages



Free Frames:



- An important aspect of paging is the clear separation between the user's view of memory and the actual physical memory.
- The user program views memory as one single space, containing only this one program. In fact, the user program is scattered throughout physical memory, which also holds other programs.
- The logical addresses are translated into physical addresses by the address-translation hardware. This mapping is hidden from the user and is controlled by the OS.

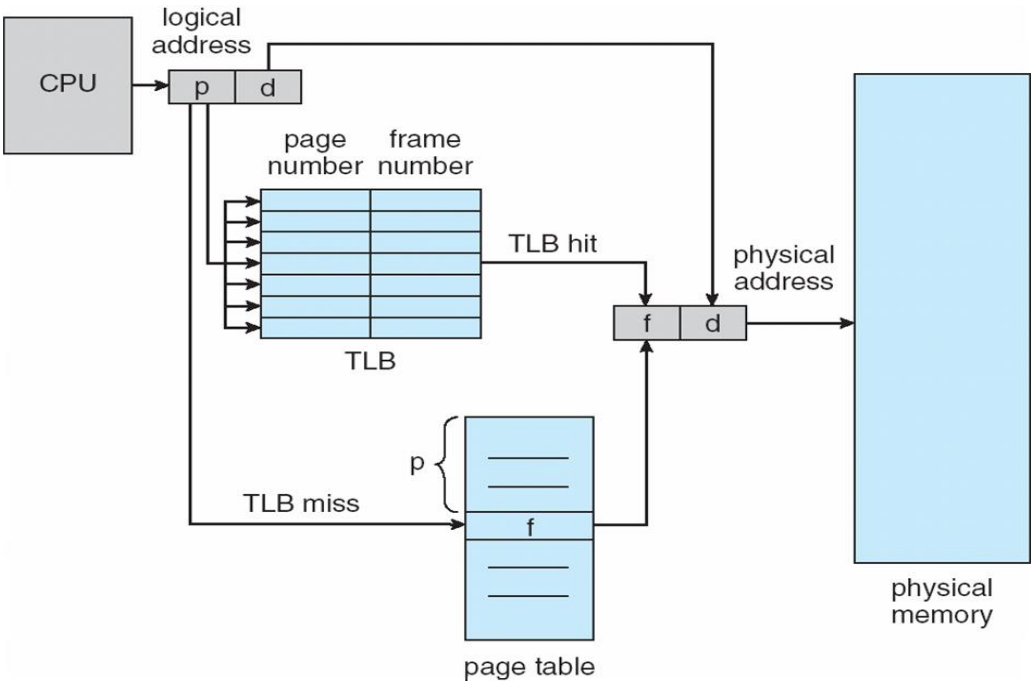
- The user process has no way of addressing memory outside of its page table, and the table includes only those pages that the process owns.
- Since the OS is managing physical memory, it must be aware of the allocation details of physical memory-which frames are allocated, which frames are available, how many total frames there are, and so on.
- This information is generally kept in a data structure called a frame table. The frame table has one entry for each physical page frame, indicating whether the latter is free or allocated and, if it is allocated, to which page of which process or processes.

Hardware Support for Paging:

Implementation of Page Table

- Page table is kept in main memory
- **Page-table base register** (**PTBR**) points to the page table
- **Page-table length register** (**PTLR**) indicates size of the page table
- In this scheme every data/instruction access requires two memory accesses
  - One for the page table and one for the data / instruction
  - two **memory access** problem can be solved by the use of a special fast-lookup hardware cache called **associative memory** or **translation look-aside buffers** (**TLBs**)

- Some TLBs store **address-space identifiers** (**ASIDs**) in each TLB entry – uniquely identifies each process to provide address-space protection for that process
- TLBs typically small (64 to 1,024 entries)
- On a TLB miss, value is loaded into the TLB for faster access next time



TLB :

- The TLB is associative, high-speed memory.
- Each entry in the TLB consists of two parts:
    - a key (or tag) and a value.
- When the associative memory is presented with an item, the item is compared with all keys simultaneously.
- If the item is found, the corresponding value field is returned.
- The TLB contains only a few of the page-table entries.
- When a logical address is generated by the CPU, its page number is presented to the TLB.
- If the page number is not in the TLB (known as a TLB miss), a memory reference to the page table must be made.
- Depending on the CPU, this may be done automatically in hardware or via an interrupt to the operating system.
- If the page number is found, its frame number is immediately available and is used to access memory.

Hit Ratio-The percentage of times that the page number of interest is found in the TLB is called the hit ratio.

Effective Memory Access Time

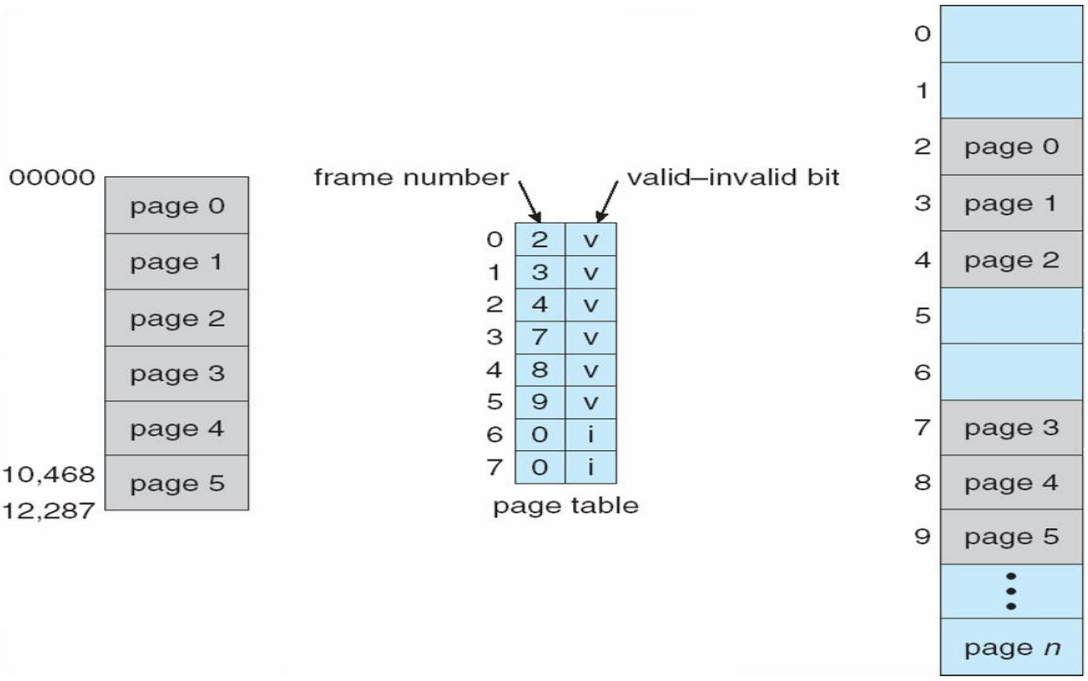
- An 80-percent hit ratio, for example, means that we find the desired page number in the TLB 80 percent of the time. If it takes 100 nanoseconds to access memory, then a mapped-memory access takes 100 nanoseconds when the page number is in the TLB.
- If we fail to find the page number in the TLB then we must first access memory for the page table and frame number (100 nanoseconds) and then access the desired byte in memory (100 nanoseconds), for a total of 200 nanoseconds. effective access time = 0.80 × 100 + 0.20 × 200 = 120 nanoseconds
- For a 99-percent hit ratio, which is much more realistic, we have effective access time = 0.99 × 100 + 0.01 × 200 = 101 nanoseconds

Protection

- Memory protection in a paged environment is accomplished by protection bits associated with each frame. Normally, these bits are kept in the page table. One bit can define a page to be read write or read-only.
- Every reference to memory goes through the page table to find the correct frame number. At the same time that the physical addres s is being computed, the protection bits can be checked to verify that no writes are being made to a read-only page.
- An attempt to write to a read-only page causes a hardware trap to the operating system (or memory-protection violation).
- One additional bit is generally attached to each entry in the page table: a valid-invalid bit.

- When this bit is set to ``valid'', the associated page is in the process's logical address space and is thus a legal (or valid) page.
- When the bit is set to ``invalid'', the page is not in the process's logical address space.
- Illegal addresses are trapped by use of the valid-invalid bit. The OS sets this bit for each page to allow or disallow access to the page.

Valid (v) or Invalid (i) Bit In A Page Table:



- Item Addresses in pages 0, 1, 2, 3, 4, and 5 are mapped normally through the page table.
- Any attempt to generate an address in pages 6 or 7, however, will find that the valid-invalid bit is set to invalid, and the computer will trap to the OS

Shared Pages

- An advantage of paging is the possibility of sharing common code. This consideration is particularly important in a time-sharing environment
- Re-entrant code is non-self-modifying code; it never changes during execution. Thus, two or more processes can execute the same code at the same time.
- Each process has its own copy of registers and data storage to hold the data for the process's execution. The data for two different processes will, of course, be different.
- Only one copy of the editor need be kept in physical memory. Each user's page table maps onto the same physical copy of the editor, but data pages are mapped onto different frames.

Shared pages example: Sharing of code in a paging environment.

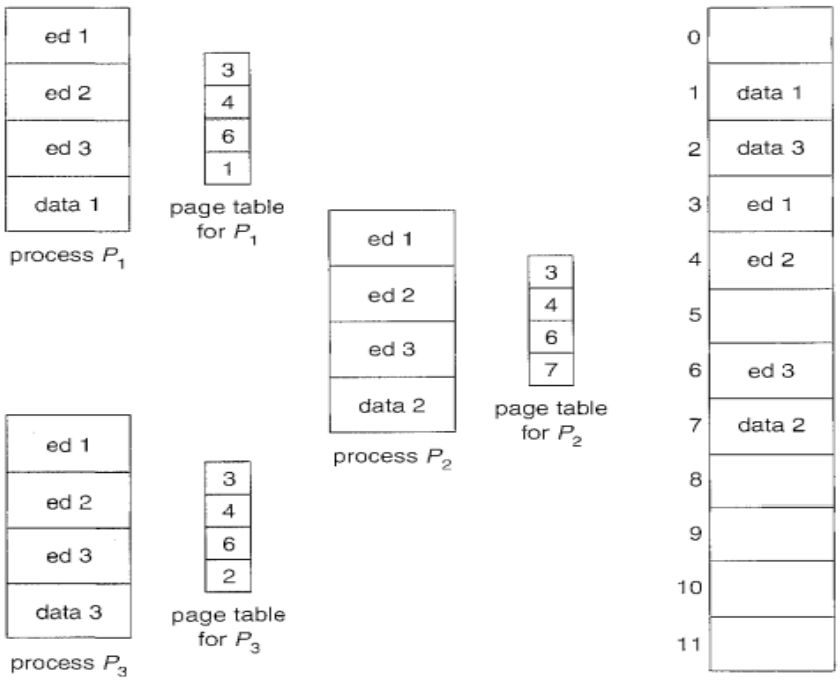


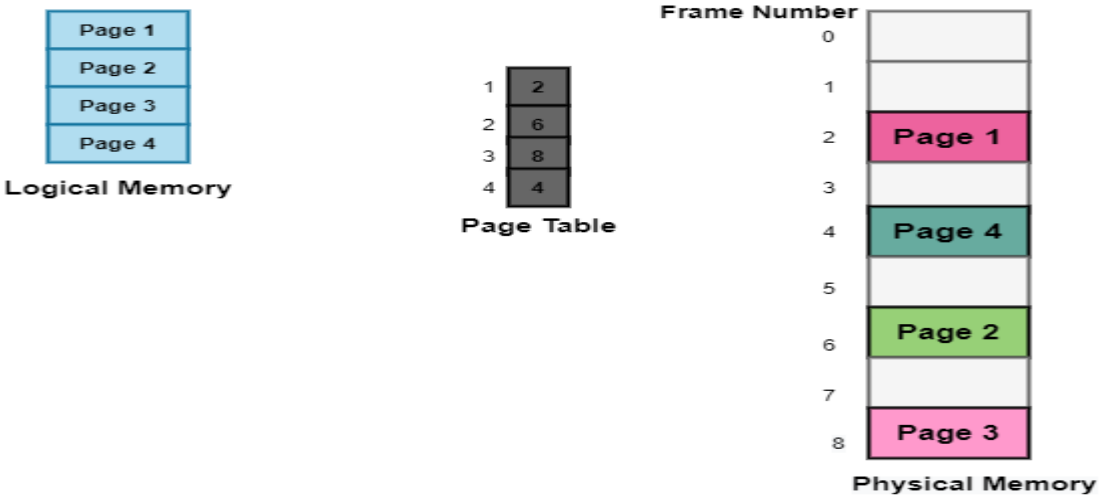
Figure 8.13 Sharing of code in a paging environment.

Structure of the Page Table:

The data structure that is used by the virtual memory system in the operating system of a computer in order to store the mapping between physical and logical addresses is commonly known as **Page Table**.

As we had already told you that the logical address that is generated by the CPU is translated into the physical address with the help of the page table.

- Thus page table mainly provides the corresponding frame number (base address of the frame) where that page is stored in the main memory.



Characteristics of the Page Table :

Some of the characteristics of the Page Table

- It is stored in the main memory.
- Generally; the Number of entries in the page table = the Number of Pages in which the process is divided.
- **PTBR** means page table base register and it is basically used to hold the base address for the page table of the current process.
- Each process has its own independent page table.

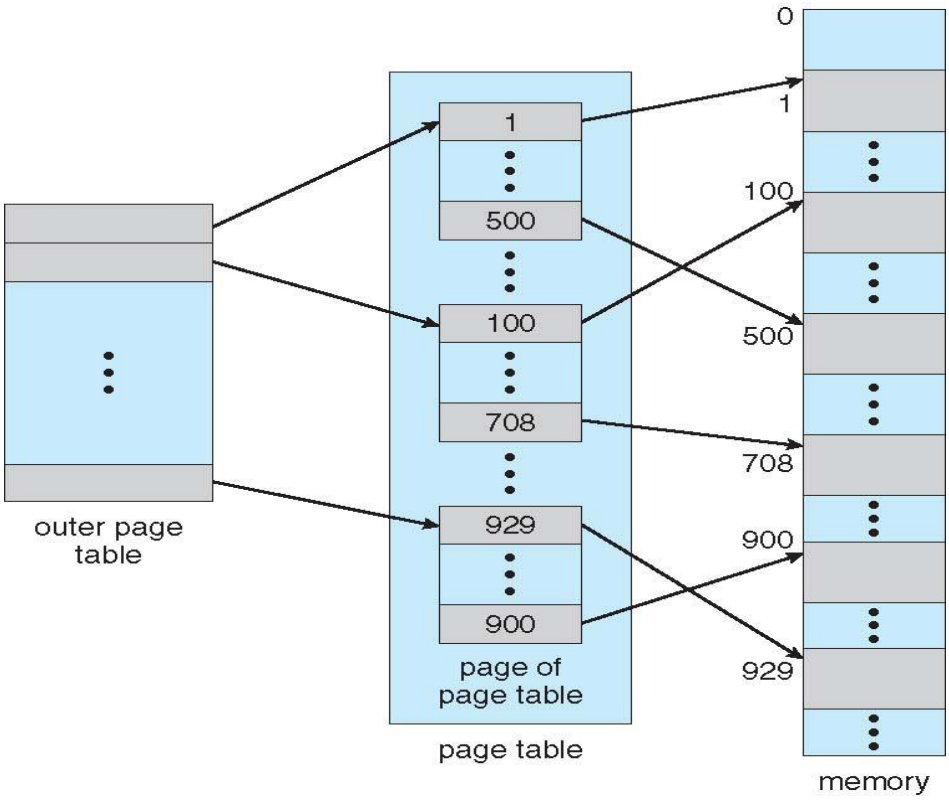
The most common techniques for structuring the page table are

- Hierarchical page tables
- Hashed page tables
- Inverted page tables

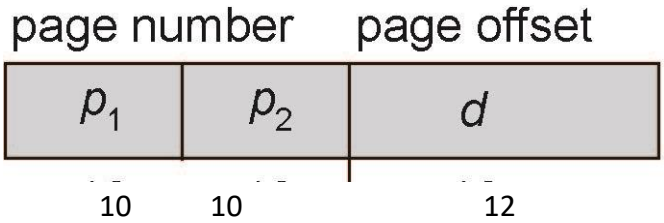
Hierarchical page tables :

- Break up the logical address space into multiple page tables
- A simple technique is a two-level page table
- We then page the page table
- The page table itself becomes large for computers with large logical address space (232 to 264).
- Example: • Consider a system with a 32-bit logical address space. If the page size in such a system is 4 KB (212), then a page table may consist of up to 1 million entries (232/212).
- Assuming that each entry consists of 4 bytes, each process may need up to 4 MB of physical address space for the page table alone.
- The page table should be allocated contiguously in main memory.
- The solution to this problem is to divide the page table into smaller pieces.
- One way of dividing the page table is to use a two-level paging algorithm, in which the page table itself is also paged as in the figure:

Two-Level Page-Table Scheme:



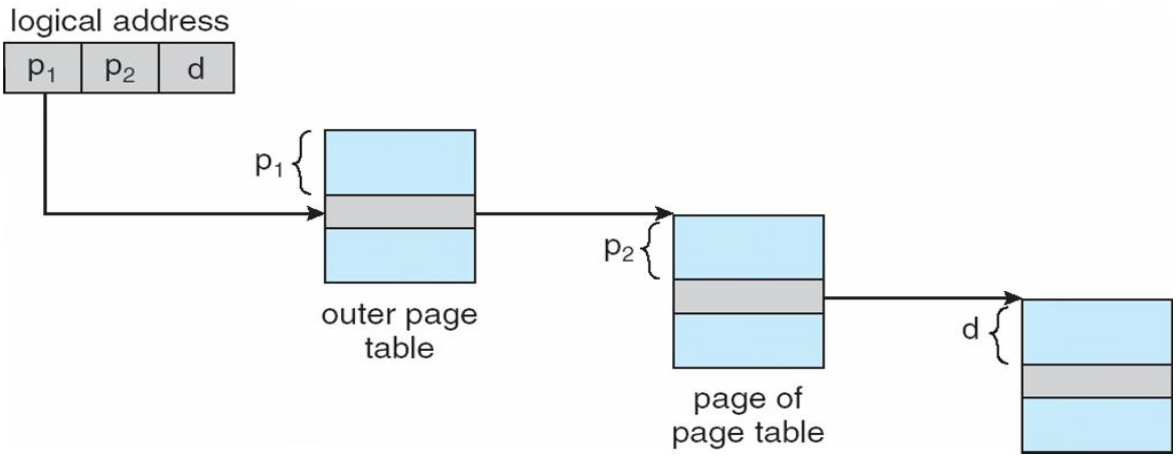
- A logical address (on 32-bit machine with 1K page size) is divided into:
  - a page number consisting of 20 bits
  - a page offset consisting of 12 bits
  - Since the page table is paged, the page number is further divided into:
  - a 10-bit page number
  - a 12-bit page offset
  - Thus, a logical address is as follows:



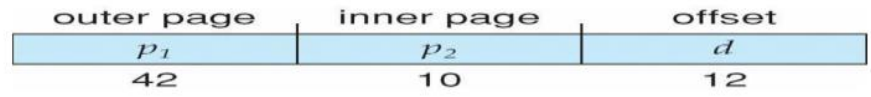
where p_1 is an index into the outer page table, and p_2 is the displacement within the page of the inner page table

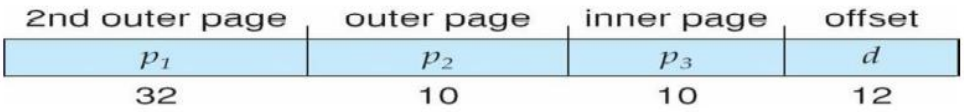
- Known as **forward-mapped page table**

Address-Translation Scheme:



Three-level Paging Scheme

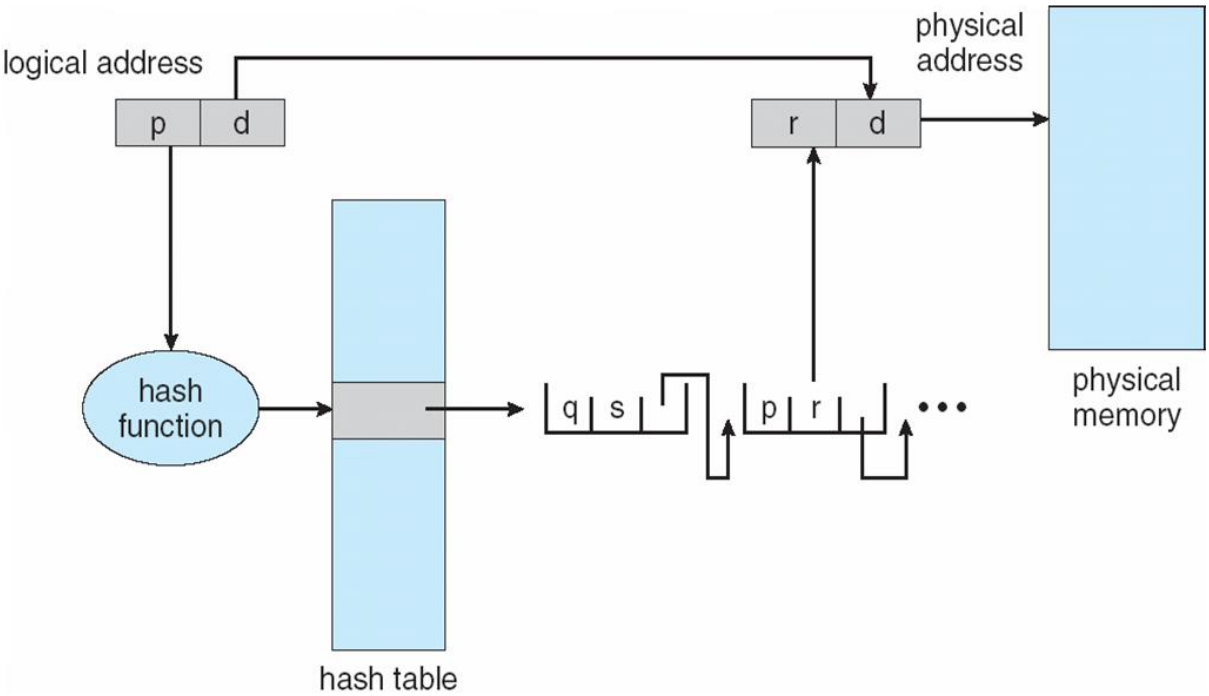




Hashed Page Tables

- A common approach for handling address spaces larger than 32 bits is to use a hashed page table, with the hash value being the virtual page number.
- Each entry in the hash table contains a linked list of elements that hash to the same location (to handle collisions). •
- Each element consists of three fields:
  - the virtual page number
  - the value of the mapped page frame
  - a pointer to the next element in the linked list.
- Common in address spaces > 32 bits
- The virtual page number is hashed into a page table
- This page table contains a chain of elements hashing to the same location
- Each element contains (1) the virtual page number (2) the value of the mapped page frame (3) a pointer to the next element
- Virtual page numbers are compared in this chain searching for a match
- If a match is found, the corresponding physical frame is extracted
- Variation for 64-bit addresses is **clustered page tables**
- Similar to hashed but each entry refers to several pages (such as 16) rather than 1
- Especially useful for **sparse** address spaces (where memory references are non-contiguous and scattered)

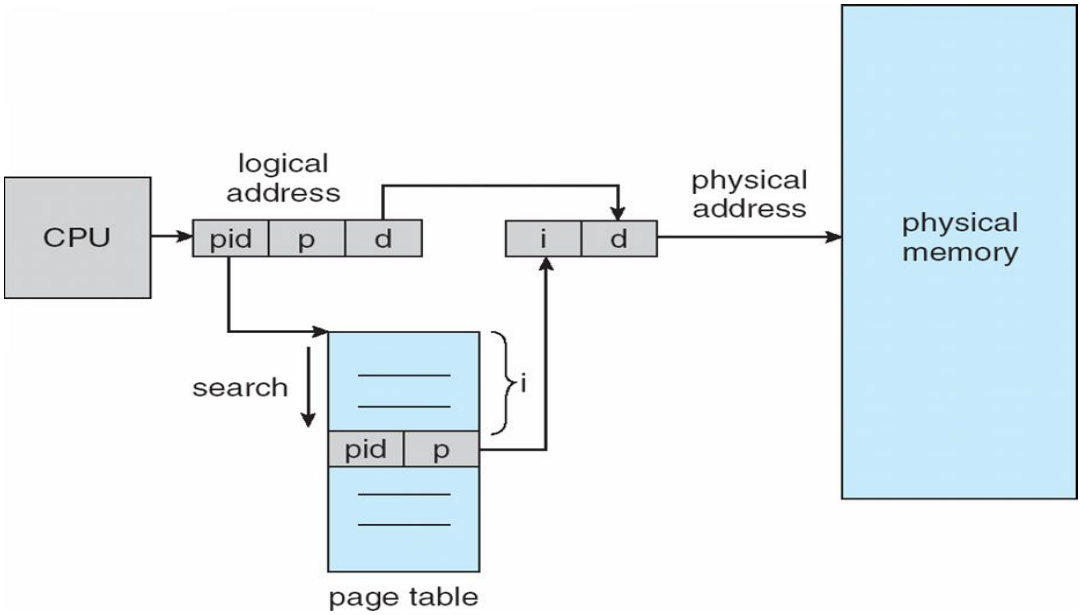
Hashed page table:



Inverted Page Table:

- Each process has an associated page table.
- The page table has one entry for each page that the process is using. This table representation is a natural one, since processes reference pages through the pages' virtual addresses.
- The operating system must then translate this reference into a physical memory address. Since the table is sorted by virtual address, the operating system is able to calculate where in the table the associated physical address entry is located and to use that value directly.
- Rather than each process having a page table and keeping track of all possible logical pages, track all physical pages
- One entry for each real page of memory
- Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page
- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs

Inverted Page Table Architecture



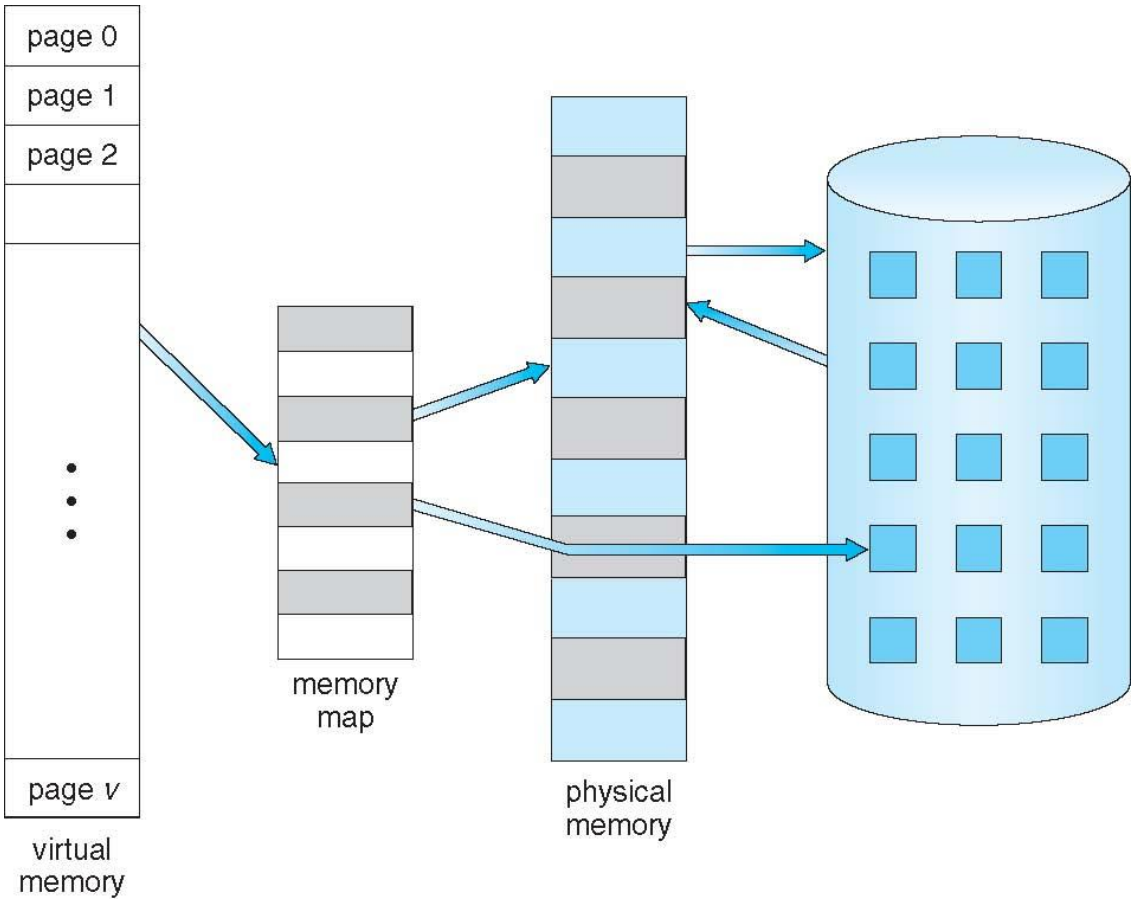
Virtual Memory:

- Virtual memory is a technique that allows the execution of processes that are not completely in memory.
- One major advantage of this scheme is that programs can be larger than physical memory.
- Further, virtual memory abstracts main memory into an extremely large, uniform array of storage, separating logical memory as viewed by the user from physical memory.
- Virtual memory also allows processes to share files easily and to implement shared memory.
- Virtual memory is not easy to implement, however, and may substantially decrease performance if it is used carelessly.

    - **Virtual memory** –involves separation of user logical memory from physical memory. This separation allows an extremely large virtual memory to be provided for programmers when only a smaller physical memory is available
        - 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 address space** – it is a 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

- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation

Diagram showing Virtual Memory That is Larger Than Physical Memory:

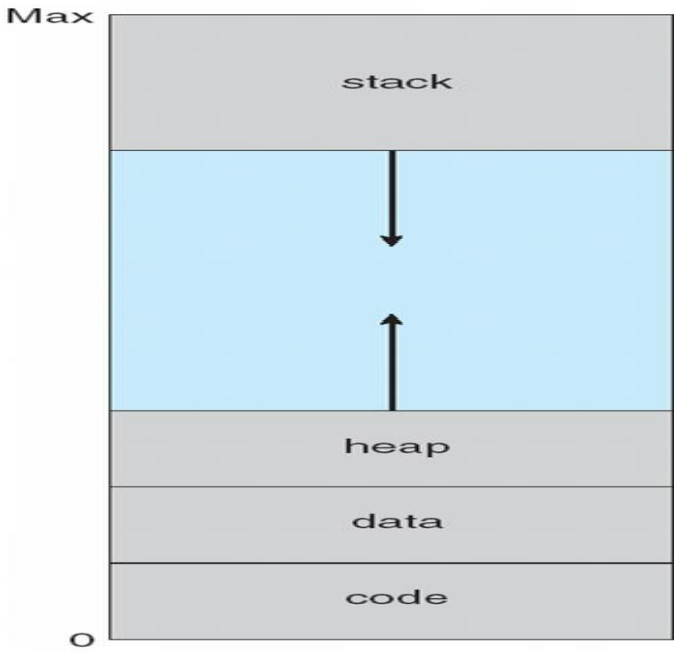


The virtual address space of a process refers to the logical (or virtual) view of how a process is stored in memory. Typically, this view is that a process begins at a certain logical address-say, address O -and exists in contiguous memory.

Virtual-address Space:

- Usually design logical address space for stack to start at Max logical address and grow “down” while heap grows “up”
  - Maximizes address space use
  - Unused address space between the two is hole
    - No physical memory needed until heap or stack grows to a given new page
- Enables **sparse** address spaces with holes left for growth, dynamically linked libraries, etc.
- Virtual address spaces that include holes are known as sparse address spaces.

- System libraries shared via mapping into virtual address space
- Shared memory by mapping pages read-write into virtual address space
- Pages can be shared during fork(), speeding process creation



Advantages of Virtual Memory

1. The degree of Multiprogramming will be increased.
2. User can run large application with less real RAM.
3. There is no need to buy more memory RAMs.

Disadvantages of Virtual Memory

1. The system becomes slower since swapping takes time.
2. It takes more time in switching between applications.
3. The user will have the lesser hard disk space for its use.

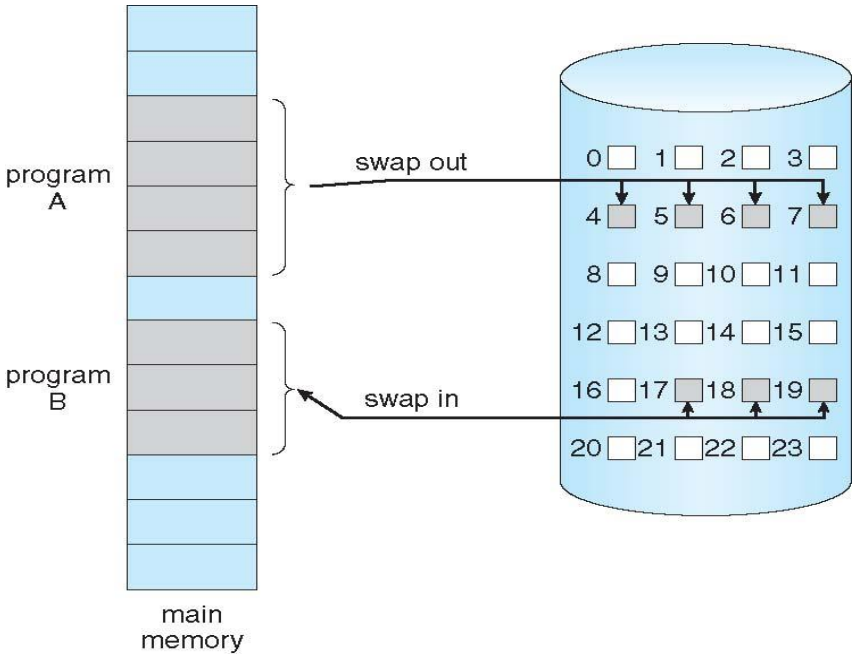
Explain Demand Paging with neat diagram:

Demand Paging:

- A demand-paging system is similar to a paging system with swapping where processes reside in secondary memory (usually a disk). Transfer of a paged memory to contiguous disk space
- 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.
- A swapper manipulates entire processes, whereas a pager is concerned with the individual pages of a process. We thus use pager, rather than swapper, in connection with demand paging.

- Demand Paging is a popular method of virtual memory management. In demand paging, the pages of a process which are least used, get stored in the secondary memory.
- A page is copied to the main memory when its demand is made or page fault occurs. There are various page replacement algorithms which are used to determine the pages which will be replaced.
- Consider how an executable program might be loaded from disk into memory. One option is to load the entire program in physical memory at program execution time. An alternative strategy is to 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.

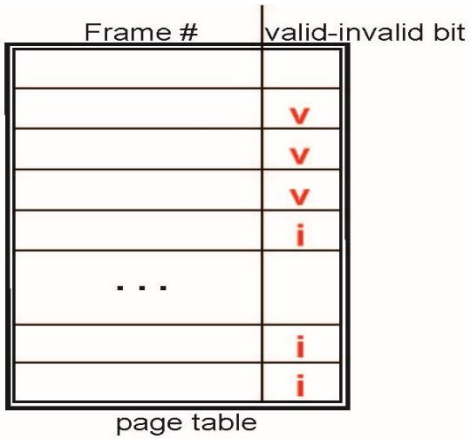
  - Could bring entire process into memory at load time
  - Or bring a page into memory only when it is needed
    - Less I/O needed, no unnecessary I/O
    - Less memory needed
    - Faster response
    - More users
  - Similar to paging system with swapping (diagram on right)
  - Page is needed ⇒ reference to it
    - invalid reference ⇒ abort
    - not-in-memory ⇒ bring to memory
  - **Lazy swapper** – it never swaps a page into memory unless page will be needed
    - Swapper that deals with pages is a **pager**



Valid-Invalid Bit:

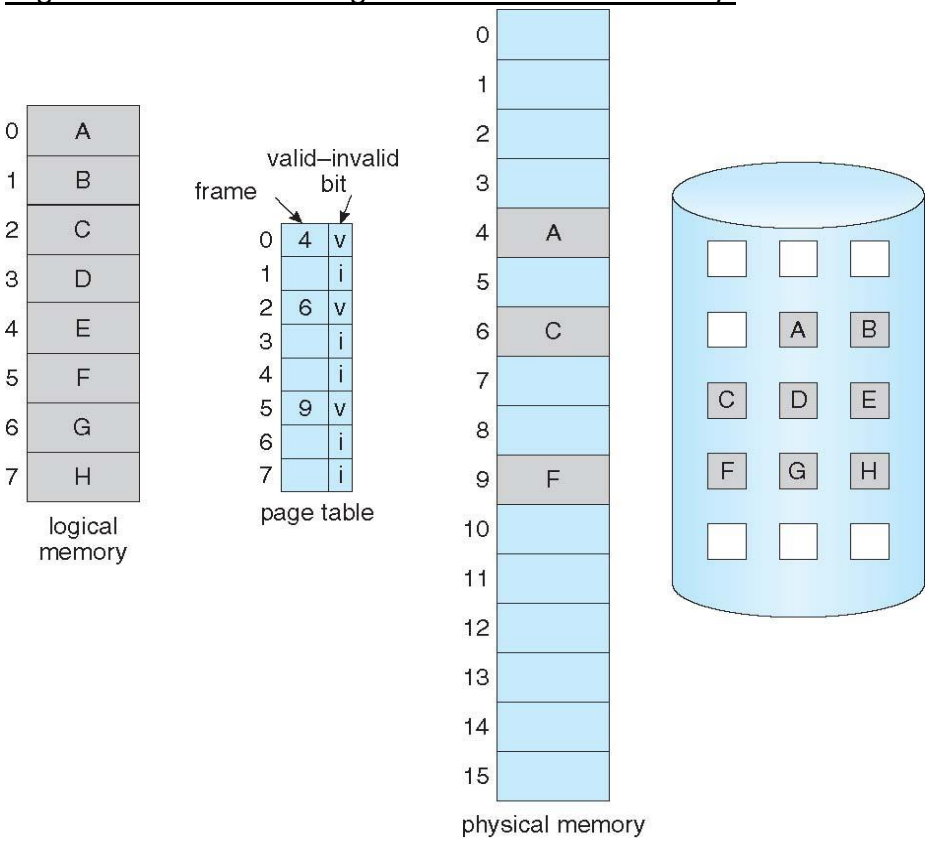
- With each page table entry a valid–invalid bit is associated (**v** ⇒ in-memory – **memory resident**, **i** ⇒ not-in-memory)
- Initially valid–invalid bit is set to **i** on all entries

Example of a page table snapshot:



- During MMU address translation, if valid–invalid bit in page table entry is **i** ⇒ page fault

Page Table When Some Pages Are Not in Main Memory:



- When a process is to be swapped in, the pager guesses which pages will be used before the process is swapped out again. It avoids reading into memory pages that will not be used anyway, decreasing the swap time and the amount of physical memory needed.

- Some form of hardware support is needed to distinguish between the pages that are in memory and the pages that are on the disk.
- The valid -invalid bit scheme can be used for this purpose.
- This time however, when this bit is set to ``valid'', the associated page is both legal and in memory.
- If the bit is set to ``invalid'', the page either is not valid (that is, not in the logical address space of the process) or is valid but is currently on the disk.
- The page-table entry for a page that is brought into memory is set as usual, but the page-table entry for a page that is not currently in memory is either simply marked invalid or contains the address of the page on disk
- While the process executes and accesses pages that are memory resident, execution proceeds normally.
- Access to a page marked invalid causes a page-fault trap.
- The paging hardware, in translating the address through the page table, will notice that the invalid bit is set, causing a trap to the OS.
- The procedure for handling this page fault is straightforward.

Steps in Handling a Page Fault:

- If there is a reference to a page, first reference to that page will trap to operating system: **page fault**

1. Operating system looks at another table to decide:
   - Invalid reference $\Rightarrow$ abort
   - Just not in memory
2. Find free frame
3. Swap page into frame via scheduled disk operation
4. Reset tables to indicate page now in memory
   Set validation bit = **v**
5. Restart the instruction that caused the page fault

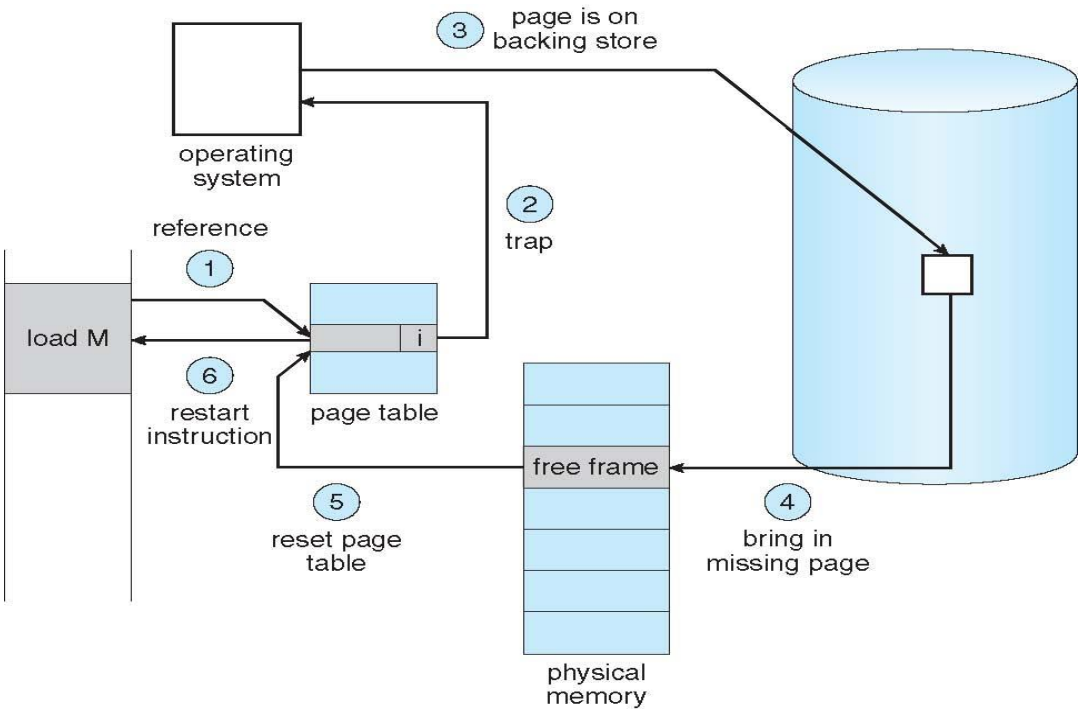
1. We check an internal table (in PCB) for this process to determine whether the reference was a valid or an invalid memory access.
2. If the reference was invalid, we terminate the process. If it was valid, but we have not yet brought in that page, we now page it in.
3. We find a free frame.
4. We schedule a disk operation to read the desired page into the newly allocated frame.
5. When the disk read is complete, we modify the internal table kept with the process and the page table.
6. We restart the instruction that was interrupted by the trap.

In the extreme case, we can start executing a process with no pages in memory.

- When the OS sets the instruction pointer to the first instruction of the process, which is on a non-memory-resident page, the process immediately faults for the page.

- After this page is brought into memory, the process continues to execute, faulting as necessary until every page that it needs is in memory.
- At that point, it can execute with no more faults. This scheme is pure demand paging: Never bring a page into memory until it is required.

Steps in Handling a Page Fault:



Performance of Demand Paging

- Demand paging can significantly affect the performance of a computer system. Let's compute the effective access time for a demand-paged memory.
- For most computer systems, the memory-access time, denoted ma, ranges from 10 to 200 nanoseconds.
- As long as we have no page faults, the effective access time is equal to the memory access time.
- If, however a page fault occurs, we must first read the relevant page from disk and then access the desired word.
- Let p be the probability of a page fault(o<=p<=1). We would expect p to be close to zero -that is, we would expect to have only a few page faults.
  - Three major activities
    - Service the interrupt – careful coding means just several hundred instructions needed
    - Read the page – lots of time
    - Restart the process – again just a small amount of time
  - Page Fault Rate $0 \le p \le 1$
    - if $p = 0$ no page faults
    - if $p = 1$, every reference is a fault

- Effective Access Time (EAT)

  EAT = (1 – *p*) x memory access + *p* * (page fault time)

We see, then, that the effective access time is directly proportional to the page-fault rate. It is important to keep the page-fault rate low in a demand-paging system. Otherwise, the effective access time increases, slowing process execution dramatically.

- An additional aspect of demand paging is the handling and overall use of swap space.
- Disk I/O to swap space is generally faster than that to the file system. It is faster because swap space is allocated in much larger blocks, and file lookups and indirect allocation methods are not used.
- The system can therefore gain better paging throughput by copying an entire file image into the swap space at process start up and then performing demand paging from the swap space.
- Another option is to demand pages from the file system initially but to write the pages to swap space as they are replaced.

Copy-on-Write:

- **Copy-on-Write** (COW) allows both parent and child processes to initially ***share*** the same pages in memory
  - If either process modifies a shared page, only then is the page copied
- COW allows more efficient process creation as only modified pages are copied
- In general, free pages are allocated from a **pool** of **zero-fill-on-demand** pages
  - Pool should always have free frames for fast demand page execution
- The main intention behind the CoW technique is that whenever a parent process creates a child process both parent and child process initially will share the same pages in the memory.
- These shared pages between parent and child process will be marked as copy-on-write which means that if the parent or child process will attempt to modify the shared pages then a copy of these pages will be created and the modifications will be done only on the copy of pages by that process and it will not affect other processes.
- Now its time to take a look at the basic example of this technique:
- Let us take an example where Process A creates a new process that is Process B, initially both these processes will share the same pages of the me

Before Process 1 Modifies Page C:

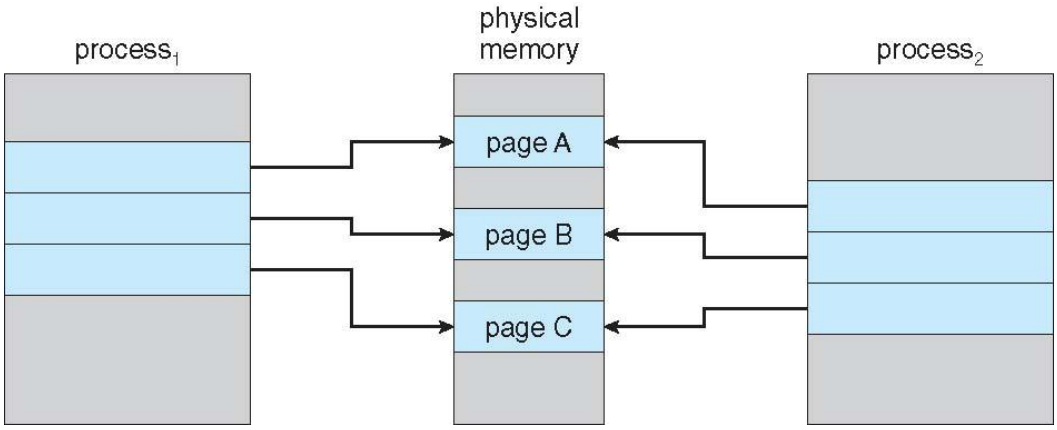


Figure: Above figure indicates parent and child process sharing the same pages
Now, let us assume that process A wants to modify a page in the memory. When the **Copy-on-write(CoW)** technique is used, only those pages that are modified by either process are copied; all the unmodified pages can be easily shared by the parent and child process.

After Process 1 Modifies Page C:

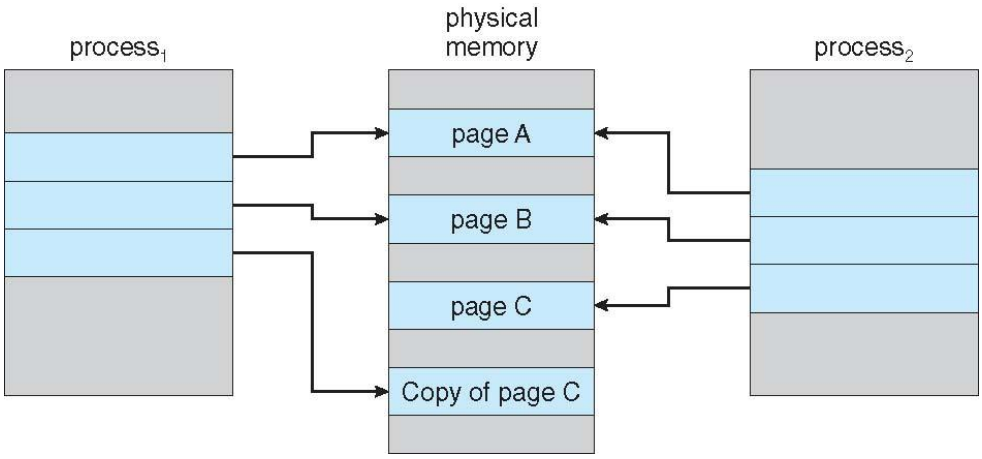


Figure: After Page C is modified by Process1
Whenever it is determined that a page is going to be duplicated using the copy-on-write technique, then it is important to note the location from where the free pages will be allocated. There is a pool of free pages for such requests; provided by many operating systems.
And these free pages are allocated typically when the stack/heap for a process must expand or when there are copy-on-write pages to manage.

Page Fault:

- A Page fault is another important concept in page replacement algorithms. A page fault occurs when a page requested by a program is not present in the main memory.
- Page hit: If a requested page is present in memory, it is called as Page – hit.

- Page fault dominates more like an error. It mainly occurs when any program tries to access the data or the code that is in the address space of the program, but that data is not currently located in the RAM of the system.

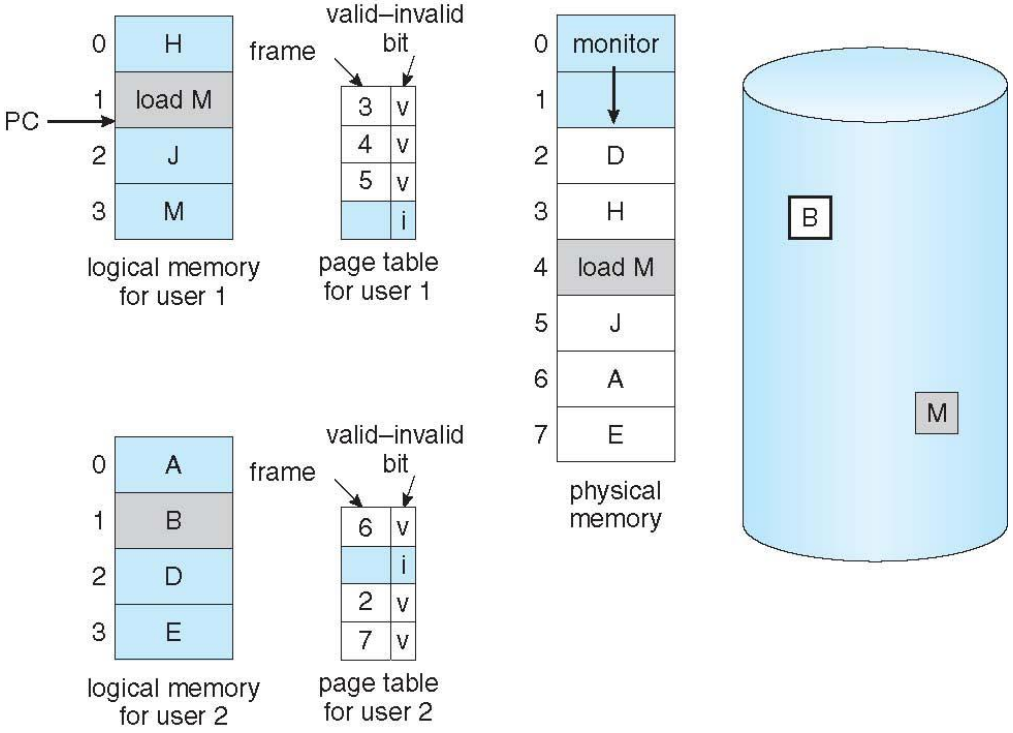
• So basically when the page referenced by the CPU is not found in the main memory then the situation is termed as Page Fault.

• Whenever any page fault occurs, then the required page has to be fetched from the secondary memory into the main memory.

Need For Page Replacement:

- While a user process is executing, a page fault occurs. The operating system determines where the desired page is residing on the disk but then finds that there are no free frames on the free-frame list; all memory is in use.
- At that time Page replacement is good solution
- Page replacement takes the following approach.
- If no frame is free, we find one that is not currently being used and free it.
- We can free a frame by writing its contents to swap space and changing the page table(and all other tables) to indicate that the page is no longer in memory.
- We can now use the freed frame to hold the page for which the process faulted.

Need For Page Replacement:



Page Replacement:

Page replacement is done when the requested page is not found in the main memory (page fault).

The page replacement algorithm decides which memory page is to be replaced. The process of replacement is sometimes called swap out or write to disk.

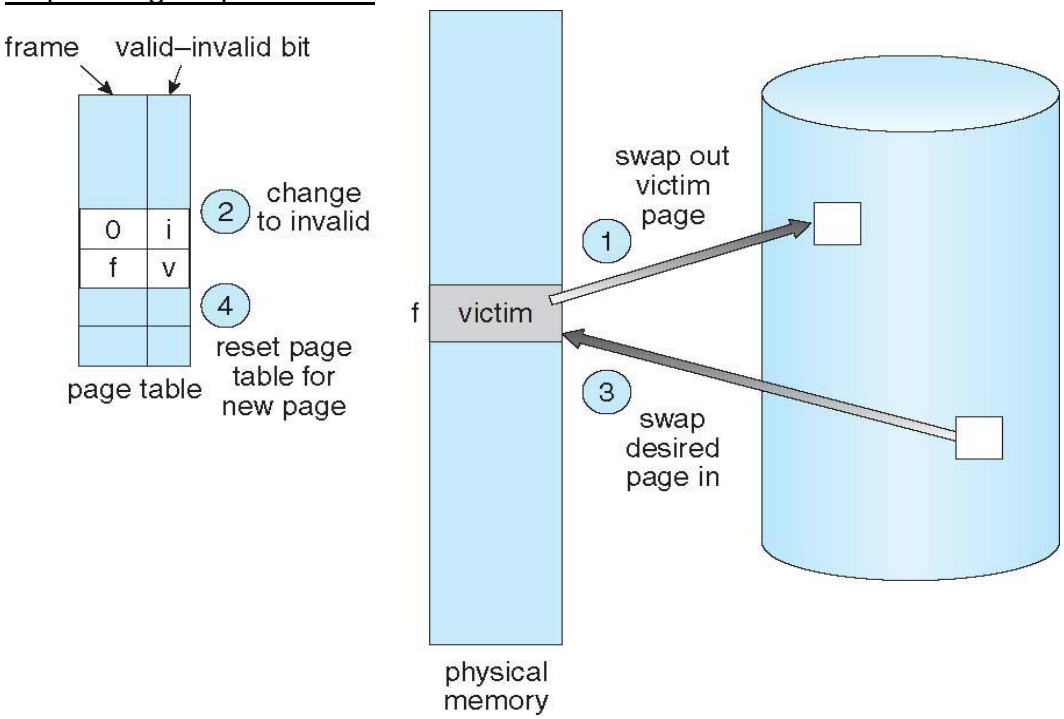
Basic Page Replacement

- Page replacement takes the following approach.
- If no frame is free, we find one that is not currently being used and free it.
- We can free a frame by writing its contents to swap space and changing the page table(and all other tables) to indicate that the page is no longer in memory.
- We can now use the freed frame to hold the page for which the process faulted.

The Steps of Page Replacement are:

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

Steps of Page Replacement :



- Page replacement is basic to demand paging. It completes the separation between logical memory and physical memory.
- We must solve two major problems to implement demand paging:
- 1. develop a frame-allocation algorithm- If we have multiple processes in memory, we must decide how many frames to allocate to each process.
- 2. develop a page-replacement algorithm. When page replacement is required, we must select the frames that are to be replaced.
- Designing appropriate algorithms to solve these problems is an important task, because disk I/O is so expensive. Even slight improvements in demand-paging methods yield large gains in system performance.

Page Replacement Algorithms:

There are 3 page replacement algorithms:

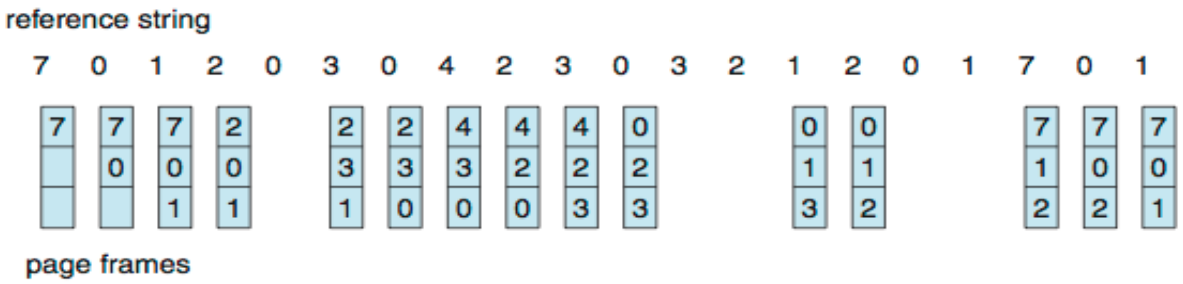
1. FIFO Page Replacement
2. Optimal Page Replacement
3. LRU Page Replacement

FIFO Page Replacement algorithm:

- The simplest page-replacement algorithm is a first-in, first-out (FIFO) algorithm.
- 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.
- The first-in, first-out (FIFO) page replacement algorithm is a low-overhead algorithm that requires little book keeping on the part of the operating system. In FIFO, the page which comes in first is replaced first.
- 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 or first page is chosen. We can create a FIFO queue to hold all pages in memory. We replace the page at the head of the queue. When a page is brought into memory, we insert it at the tail of the queue.
- We can get Beladys anomaly for FIFO algorithm.

For our example reference string, our three frames are initially empty.

- Reference string: **7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1**
- 3 frames (3 pages can be in memory at a time per process)



Here, there are 15 page faults

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

- The next reference (2) replaces page 7, because page 7 was brought in first.
- Since 0 is the next reference and 0 is already in memory, we have no fault for this reference.
- The first reference to 3 results in replacement of page 0, since it is now first in line.
- Because of this replacement, the next reference, to 0, will fault.
- Page 1 is then replaced by page O. This process continues and there are 15 faults altogether.
- The FIFO page-replacement algorithm is easy to understand and program. However, its performance is not always good.
- On the one hand, the page replaced may be an initialization module that was used a long time ago and is no longer needed.
- On the other hand, it could contain a heavily used variable that was initialized early and is in constant use.
- Notice that, even if we select for replacement a page that is in active use, everything still works correctly.
- After we replace an active page with a new one, a fault occurs almost immediately to retrieve the active page. Some other page will need to be replaced to bring the active page back into memory.
- Thus, a bad replacement choice increases the page-fault rate and slows process execution. It does not cause incorrect execution.

1. Apply FIFO algorithm for reference string: consider 1,2,3,4,1,2,5,1,2,3,4,5

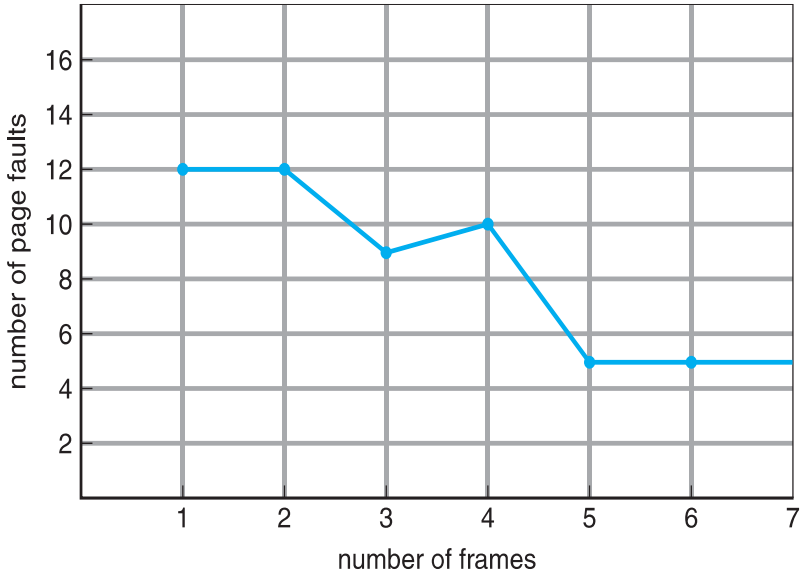
Take 3 frames and do this and find the number of page faults?

Belady's anomaly

- Belady's anomaly proves that it is possible to have more page faults when increasing the number of page frames while using the First in First Out (FIFO) page replacement algorithm.
- That means -Adding more frames can cause more page faults in FIFO algorithm.

- Beladys anomaly is not applicable to Optimal and LRU page replacement algorithms.

FIFO Illustrating Belady's Anomaly:

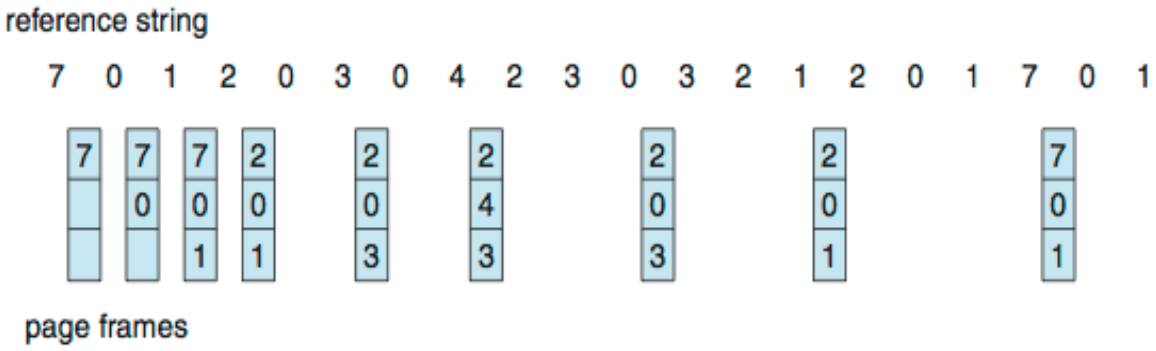


Optimal Page Replacement:

- An optimal page-replacement algorithm has the lowest page-fault rate of all algorithms (called OPT or MIN). It is simply this:
  Replace the page that will not be used for the longest period of time.
- Use of this page-replacement algorithm guarantees the lowest possible page-fault rate for a fixed number of frames.

Example:
For our example reference string, our three frames are initially empty. Apply Optimal algorithm:

- Reference string: **7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1**
- 3 frames (3 pages can be in memory at a time per process)



  - Here, we get 9 page faults.

- The first three references cause faults that fill the three empty frames.

- The reference to page 2 replaces page 7, because 7 will not be used until reference 18, whereas page 0 will be used at 5, and page 1 at 14.
- The reference to page 3 replaces page 1, as page 1 will be the last of the three pages in memory to be referenced again.
- With only nine page faults, optimal replacement is much better than a FIFO algorithm, which resulted in fifteen faults.
- If we ignore the first three, which all algorithms must suffer, then optimal replacement is twice as good as FIFO replacement.
- Unfortunately, the optimal page-replacement algorithm is difficult to implement, because it requires future knowledge of the reference string (similar situation with the SJF CPU scheduling algorithm).

LRU Page Replacement :

- In least recently used algorithm , we will **replace the page that has not been used for the longest period of time.** By the principle of locality, this should be the page least likely to be referenced in the near future.
- Least Recently Used page replacement algorithm keeps track of page usage over a short period of time. It works on the idea that the pages that have been most heavily used in the past are most likely to be used heavily in the future too.
- In LRU, whenever page replacement happens, the page which has not been used for the longest amount of time is replaced.
- The key distinction between the FIFO and OPT algorithms (other than looking backward versus forward in time) is that --in the FIFO algorithm uses the time when a page was brought into memory, whereas the OPT algorithm uses the time when a page is to be used.
- If we use the recent past as an approximation of the near future, then we can replace the page that has not been used for the longest period of time

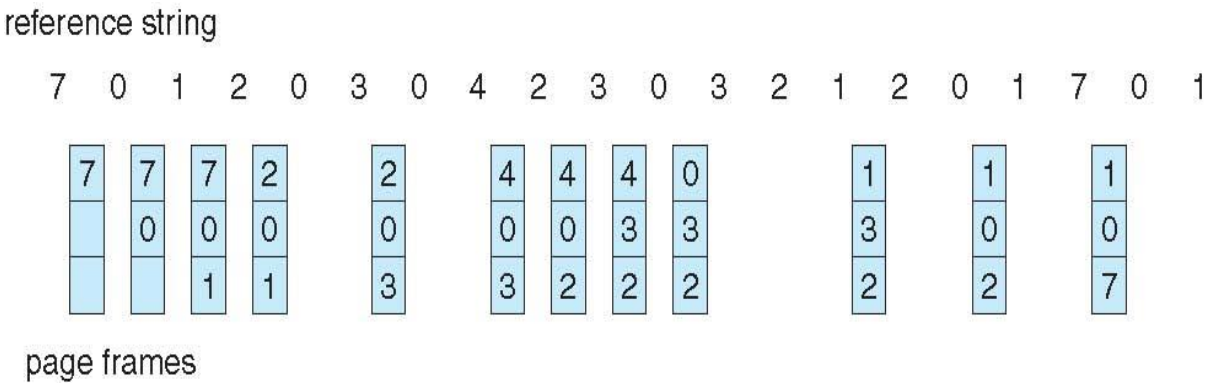
Example:

Take reference string, and take three frames which are initially empty. Apply LRU algorithm:

- Reference string: **7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1**
- Take 3 frames (3 pages can be in memory at a time per process)

In LRU,

- Use past knowledge rather than future
- Replace page that has not been used in the most amount of time
- Associate time of last use with each page



Here we get 12 page faults.

This approach is the least-recently-used (LRU) algorithm. The LRU algorithm produces 12 faults.

- Notice that the first 5 faults are the same as those for optimal replacement.
- When the reference to page 4 occurs, however, LRU replacement sees that, of the three frames in memory, page 2 was used least recently.
- Thus, the LRU algorithm replaces page 2, not knowing that page 2 is about to be used.
- When it then faults for page 2, the LRU algorithm replaces page 3, since it is now the least recently used of the three pages in memory.
- Despite these problems, LRU replacement with 12 faults is much better than FIFO replacement with 15.

Implementation of LRU :

An LRU page-replacement algorithm may require substantial hardware assistance. The problem is to determine an order for the frames defined by the time of last use. Two implementations are feasible:

Counter implementation

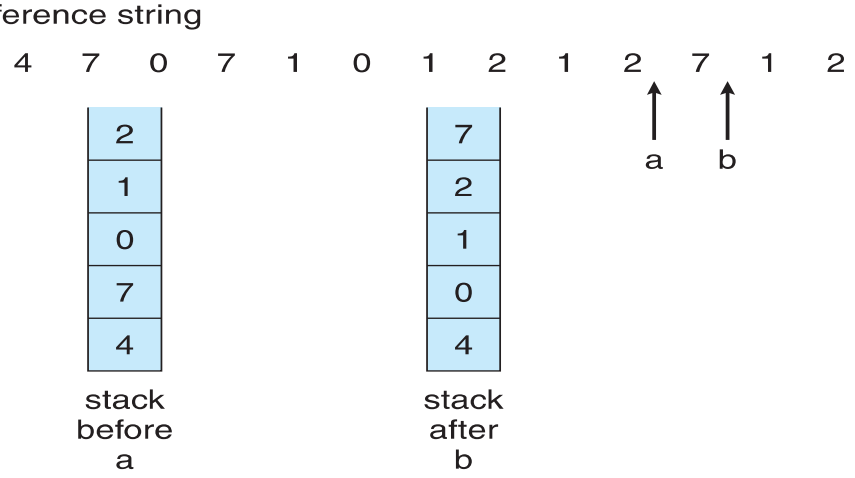
- We associate with each page-table entry a time-of-use field and add to the CPU a logical clock or counter.
- Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter
- The clock is incremented for every memory reference. Whenever a reference to a page is made, the contents of the clock register are copied to the time-of-use field in the page-table entry for that page.
- We replace the page with the smallest time value.
- When a page needs to be changed, look at the counters to find smallest value
    - Search through table needed

Stack implementation:

- Another approach to implementing LRU replacement is to keep a stack of page numbers.
- Keep a stack of page numbers in a double link form.
- 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 of the stack and the least recently used page is always at the bottom.
  - Page referenced:
    - move it to the top
    - requires 6 pointers to be changed
  - But each update more expensive
  - No search for replacement

- LRU and OPT are cases of **stack algorithms** that don't have Belady's Anomaly

Use Of A Stack to Record Most Recent Page References



Counting-Based Page Replacement:

We can keep a counter of the number of references that have been made to each page and develop the following two schemes.

- **The Least frequently used(LFU**)page-replacement algorithm requires that the page with the smallest count be replaced.
- **The Most frequently used (MFU)** page-replacement algorithm is based on the argument that the page with the smallest count was probably just brought in and has yet to be used.

Page-Buffering Algorithms

- Other procedures are often used in addition to a specific page-replacement algorithm. For example, systems commonly keep a pool of free frames.
- When a page fault occurs, a victim frame is chosen as before. However, the desired page is read into a free frame from the pool before the victim is written out.
- This procedure allows the process to restart as soon as possible, without waiting for the victim page to be written out. When the victim is later written out, its frame is added to the free-frame pool.

- An expansion of this idea is to maintain a list of modified pages.
- Whenever the paging device is idle, a modified page is selected and is written to the disk.
- Its modify bit is then reset. This scheme increases the probability that a page will be clean when it is selected for replacement and will not need to be written out.
- Another modification is to keep a pool of free frames but to remember which page was in each frame.
- Since the frame contents are not modified when a frame is written to the disk, the old page can be reused directly from the free-frame pool if it is needed before that frame is reused.
- This technique is used in the VAX/VMS system along with a FIFO replacement algorithm.

Allocation of Frames (or) Frame allocation:

Another important factor in the way frames are allocated to the various processes is page replacement.

With multiple processes competing for frames, we can classify page-replacement algorithms into two broad categories:

Equal allocation:

In a system with x frames and y processes, each process gets equal number of frames, i.e. x/y. For instance, if the system has 48 frames and 9 processes, each process will get 5 frames. The three frames which are not allocated to any process can be used as a free-frame buffer pool.

Disadvantage: In systems with processes of varying sizes, it does not make much sense to give each process equal frames. Allocation of a large number of frames to a small process will eventually lead to the wastage of a large number of allocated unused frames.

Proportional allocation:

Frames are allocated to each process according to the process size.

Advantage: All the processes share the available frames according to their needs, rather than equally.

Global versus Local Allocation

The number of frames allocated to a process can also dynamically change depending on whether you have used global replacement or local replacement for replacing pages in case of a page fault.

1. Global replacement

- Global replacement allows a process to select a replacement frame from the set all frames, even if that frame is currently allocated to some other process; that is, one process can take a frame from another.

2. Local replacement.

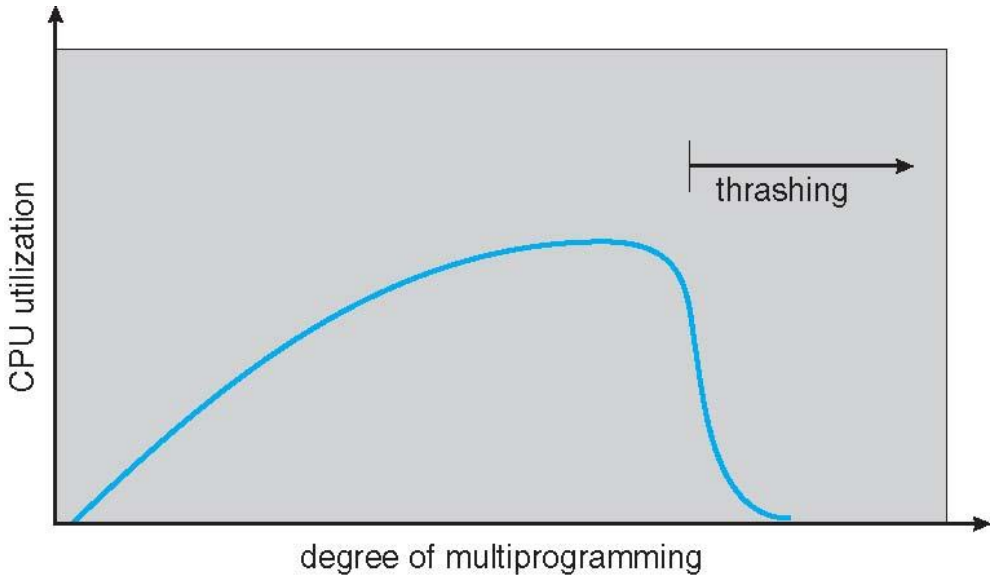
- Local replacement requires that each process select from only its own set of allocated frames. With a local replacement strategy, the number of frames allocated toa process does not change.
- With global replacement, a process may happen to select only frames allocated to other processes, thus increasing the number of frames allocated to it
- Global replacement generally results in greater system throughput and is therefore the more common method.

Thrashing in Operating System:

- In case, if the page fault and swapping happens very frequently at a higher rate, then the operating system has to spend more time swapping these pages. This state in the operating system is termed thrashing. Because of thrashing the CPU utilization is going to be reduced.
- During thrashing, the CPU spends less time on some actual productive work spend more time swapping.
- Thrashing is a condition or a situation when the system is spending a major portion of its time in servicing the page faults, but the actual processing done is very negligible.
- When many processes are executing in CPU, they completed execution and need to bring next pages. Then all the processes want page replacement. So paging has to be done.
- Thrashing occurs when a **Process** spends more time in paging or **Swapping** activities rather than its execution.
- **Definition:** This high paging activity is called thrashing. A process is thrashing if it is spending more time paging than executing
- In Thrashing, the state CPU is so much busy swapping that it cannot respond to the user program as much as it required.
- In case, if the page fault and swapping happens very frequently at a higher rate, then the operating system has to spend more time swapping these pages. This state in the operating system is termed thrashing.
- Because of thrashing the CPU utilization is going to be reduced.

Cause of Thrashing

- Thrashing results in severe performance problems.
- The OS monitors CPU utilization. If CPU utilization is too low, we increase the degree of multiprogramming by introducing a new process to the system.



- Thrashing affects the performance of execution in the Operating system. Also, thrashing results in severe performance problems in the Operating system.
- When the utilization of CPU is low, then the process scheduling mechanism tries to load many processes into the memory at the same time due to which degree of Multiprogramming can be increased. Now in this situation, there are more processes in the memory as compared to the available number of frames in the memory. Allocation of the limited amount of frames to each process.
- Whenever any process with high priority arrives in the memory and if the frame is not freely available at that time then the other process that has occupied the frame is residing in the frame will move to secondary storage and after that this free frame will be allocated to higher priority process.
- We can also say that as soon as the memory fills up, the process starts spending a lot of time for the required pages to be swapped in. Again the utilization of the CPU becomes low because most of the processes are waiting for pages.
- Thus a high degree of multiprogramming and lack of frames are two main causes of thrashing in the Operating system.
- No work is getting done, because the processes are spending all their time paging.
- As the degree of multiprogramming increases, CPU utilization also increases, although more slowly, until a maximum is reached.
- If the degree of multiprogramming is increased even further, thrashing sets in, and CPU utilization drops sharply.
- At this point, to increase CPU utilization and stop thrashing, we must decrease the degree of multiprogramming.

Preventing Thrashing

- We can limit the effects of thrashing by using a local replacement algorithm.
- With local replacement, if one process starts thrashing, it cannot steal frames from another process and cause the latter to thrash as well. However, the problem is not entirely solved.
- To prevent thrashing, we must provide a process with as many frames as it needs.

- The working-set strategy starts by looking at how many frames a process is actually using. This approach defines the locality model of process execution.
- The locality model states that, as a process executes, it moves from locality to locality. A locality is a set of pages that are actively used together. A program is generally composed of several different localities, which may overlap.

Page-Fault Frequency

- Thrashing has a high page-fault rate. Thus, we want to control the page-fault rate.
- When it is too high,we know that the process needs more frames. Conversely, if the page-fault rate is too low, then the process may have too many frames.
- We can establish upper and lower bounds on the desired page-fault rate.
- If the actual page-fault rate exceeds the upper limit, we allocate the process another from the process.
- Thus, we can directly measure and control the page-fault rate to prevent thrashing.

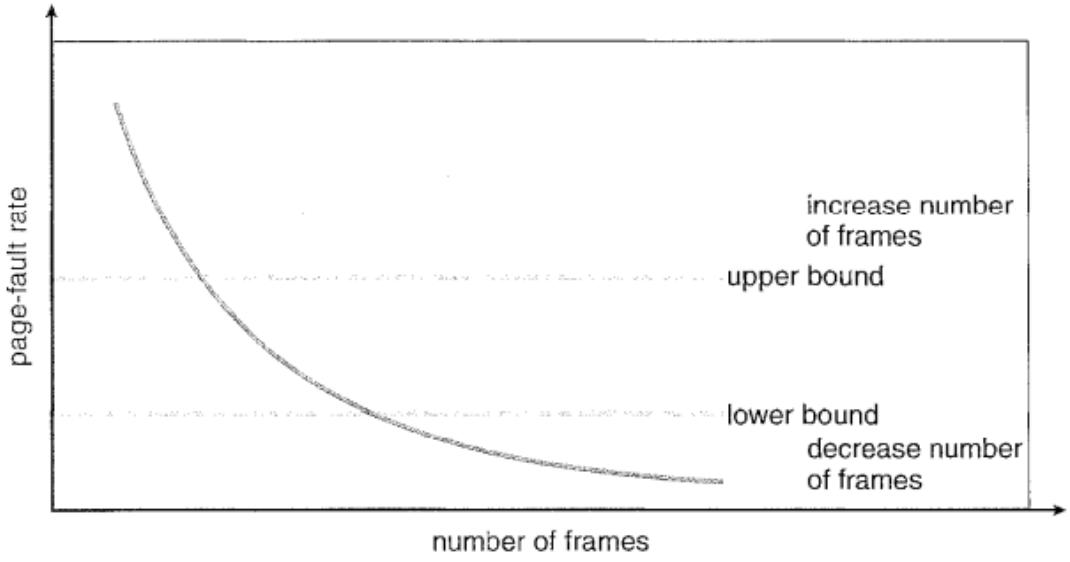


Figure 9.21 Page-fault frequency.

Memory-mapped files:

Consider a sequential read of a file on disk using the standard system calls open (),read (), and write (). Each file access requires a system call and disk access. Alternatively, we can use the virtual memory techniques discussed so far to treat file I/0 as routine memory accesses. This approach, known as memory mapping a file, allows a part of the virtual address space to be logically associated with the file. As we shall see, this can lead to significant performance increases when performing I/0.

Basic Mechanism:

Memory mapping a file is accomplished by mapping a disk block to a page (or pages) in memory. Initial access to the file proceeds through ordinary demand paging, resulting in a page fault. However, a page-sized portion of the file is read from the file system into a

physical page (some systems may opt to read in more than a page-sized chunk of memory at a time). Subsequent reads and writes to the file are handled as routine memory accesses, thereby simplifying file access and usage by allowing the system to manipulate files through memory rather than incurring the overhead of using the read () and write() system calls. Similarly, as file l/0 is done in memory- as opposed to using system calls that involve disk I/0 - file access is much faster as well.

File Systems:

File:

- A file is a collection of related information . Or in other words an entry in a directory is the file. The file may have attributes like name, creator, date, type, permissions etc.
- A file is a Contiguous logical address space
- A file can be "free formed", indexed or structured collection of related bytes having meaning only to the one who created it. Or in other words an entry in a directory is the file. The file may have attributes like name, creator, date, type, permissions etc.
- File Structure : A file has various kinds of structure. Some of them can be :
  - **Simple Record Structure** with lines of fixed or variable lengths.
  - **Complex Structures** like formatted document or reloadable load files.
  - **No Definite Structure** like sequence of words and bytes etc.

- **File system** is the part of the operating system which is responsible for file management. It provides a mechanism to store the data and access to the file contents including data and programs. Some Operating systems treats everything as a file
- Types:
  - Data
    - numeric
    - character
    - binary
  - Program
- Contents defined by file's creator
  - Many types
    - Consider **text file, source file, executable file**

Attributes of a File :

Following are some of the attributes of a file

- **Name :**It is the only information which is in human-readable form.
- **Identifier:** The file is identified by a unique tag(number) within file system.
- **Type:** It is needed for systems that support different types of files.
- **Location:** Pointer to file location on device.
- **Size:** The current size of the file.
- **Protection:** This controls and assigns the power of reading, writing, executing.
- **Time, date, and user identification:** This is the data for protection, security, and usage monitoring.

- Information about files are kept in the directory structure, which is maintained on the disk
- Many variations, including extended file attributes such as file checksum
- Information kept in the directory structure

File Systems:

File system is the part of the operating system which is responsible for file management. It provides a mechanism to store the data and access to the file contents including data and programs. Some Operating systems treats everything as a file

The File system takes care of the following issues

File Structure :

We have seen various data structures in which the file can be stored. The task of the file system is to maintain an optimal file structure.

Recovering Free space:

Whenever a file gets deleted from the hard disk, there is a free space created in the disk. There can be many such spaces which need to be recovered in order to reallocate them to other files.

disk space assignment to the files :

The major concern about the file is deciding where to store the files on the hard disk

tracking data location :

A File may or may not be stored within only one block. It can be stored in the non contiguous blocks on the disk. We need to keep track of all the blocks on which the part of the files reside.

File Operations:

A file is an abstract data type. The operations that can be performed on files are:

Creating a file.

- Two steps are necessary to create a file. First, space in the file system must be found for the file.

Writing a file.

- To write a file, we make a system call specifying both the name of the file and the information to be written to the file.
- Given the name of the file, the system searches the directory to find the file's location
- The system must keep a write pointer to the location in the file where the next write is to take place.

The write pointer must be updated whenever a write occurs

Reading a file.

- To read from a file, we use a system call that specifies the name of the file and where (in memory) the next block of the file should be put.
- Again, the directory is searched for the associated entry, and the system needs to keep a read pointer to the location in the file where the next read is to take place.
- Once the read has taken place, the read pointer is updated. •
- Because a process is usually either reading from or writing to a file, the current operation location can be kept as a per-process current file-position pointer.
- Both the read and write operations use this same pointer, saving space and reducing system complexity.

Repositioning within a file:

- The directory is searched for the appropriate entry, and the current-file-position pointer is repositioned to a given value.
- Repositioning within a file need not involve any actual I / O . This file operation is also known as a file seek.

Deleting a file.

- To delete a file, we search the directory for the named file. Having found the associated directory entry, we release all file space, so that it can be reused by other files, and erase the directory entry.

Truncating a file.

- The user may want to erase the contents of a file but keep its attributes. Rather than forcing the user to delete the file and then recreate it, this function allows all attributes to remain unchanged—except for file length—but lets the file be reset to length zero and its file space released

- The operating system can provide system calls to create, write, read, reposition, delete, and truncate files.
- File locks provide functionality similar to reader–writer locks.
- A shared lock is akin to a reader lock in that several processes can acquire the lock concurrently.
- An exclusive lock behaves like a writer lock; only one process at a time can acquire such a lock.

Open Files:

- Several pieces of data are needed to manage open files:
  - **Open-file table**: tracks open files

  - File pointer: pointer to last read/write location, per process that has the file open
  - **File-open count**: counter of number of times a file is open – to allow removal of data from open-file table when last processes closes it
  - Disk location of the file: cache of data access information
  - Access rights: per-process access mode information

Open File Locking:

- Provided by some operating systems and file systems
  - Similar to reader-writer locks
  - **Shared lock** similar to reader lock – several processes can acquire concurrently
  - **Exclusive lock** similar to writer lock
- Mandatory or advisory:
  - **Mandatory** – access is denied depending on locks held and requested
  - **Advisory** – processes can find status of locks and decide what to do

File Types – Name, Extension:

File Types

- 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.
- Example: resume.docx, server. c, and ReaderThread. cpp.
- The system uses the extension to indicate the type of the file and the type of operations that can be done on that file.
- Only a file with a.com, .exe, or.sh extension can be executed.

There are different types of files given below:

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

File Structure :

A file has various kinds of structure. Some of them can be :

- No Definite Structure - A file is a sequence of words, bytes
- Simple record structure
  - Lines
  - Fixed length
  - Variable length
- Complex Structures
  - Formatted document
  - Relocatable load file
- Can simulate last two with first method by inserting appropriate control characters
- Who decides:
  - Operating system
  - Program

File Access Mechanisms

File access mechanism refers to the manner in which the records of a file may be accessed. There are several ways to access files −

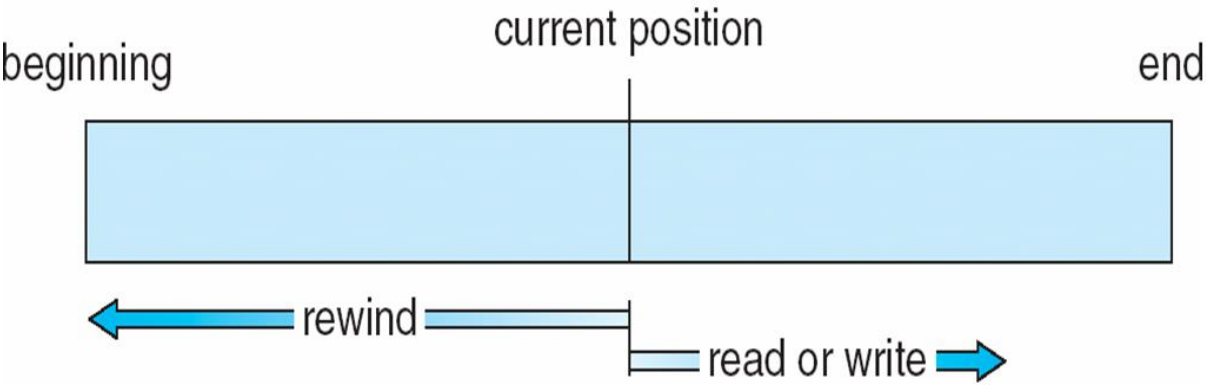
- Sequential access
- Direct/Random access
- Indexed sequential access

Sequential Access:

A sequential access is that in which the records are accessed in some sequence, i.e., the information in the file is processed in order, one record after the other. This access method is the most primitive one. Example: Compilers usually access files in this fashion.

Sequential Access:

read next
write next
reset
no read after last write
(rewrite)



Direct access

- Direct access file organization provides, accessing the records directly.
- Each record has its own address on the file with by the help of which it can be directly accessed for reading or writing.
- The records need not be in any sequence within the file and they need not be in adjacent locations on the storage medium.

- **Direct Access –** file is fixed length logical records

read *n*
write *n*
position to *n*
read next
write next
rewrite *n*

n = relative block number

- Relative block numbers allow OS to decide where file should be placed

Simulation of Sequential Access on Direct-access File:

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

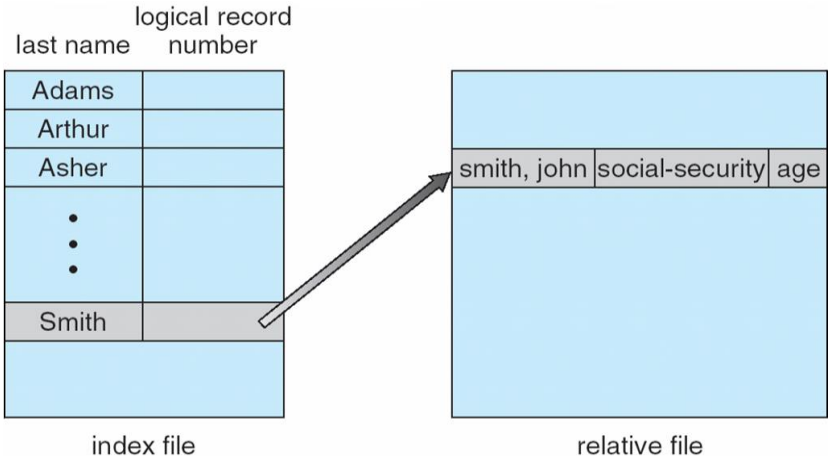
Other Access Methods:

- Can be built on top of base methods

Index method: General involve creation of an index for the file

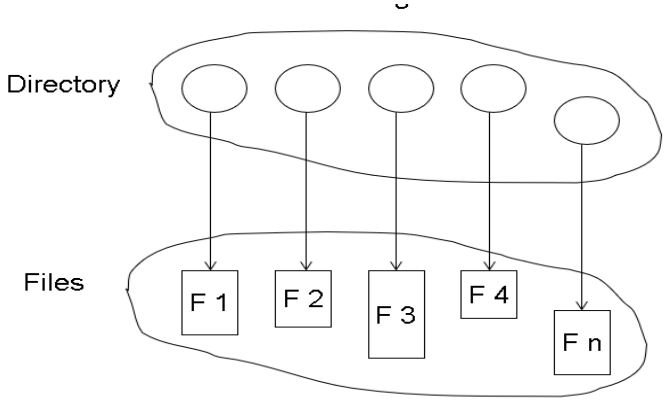
- Keep index in memory for fast determination of location of data to be operated on (consider UPC code plus record of data about that item)
- If too large, index (in memory) of the index (on disk)
- This mechanism is built up on base of sequential access.
- An index is created for each file which contains pointers to various blocks.
- Index is searched sequentially and its pointer is used to access the file directly.
- IBM indexed sequential-access method (ISAM)
  - Small master index, points to disk blocks of secondary index
  - File kept sorted on a defined key
  - All done by the OS

Example of Index and Relative Files:



Directory

- A directory is a set of files.
- A collection of nodes containing information about all files

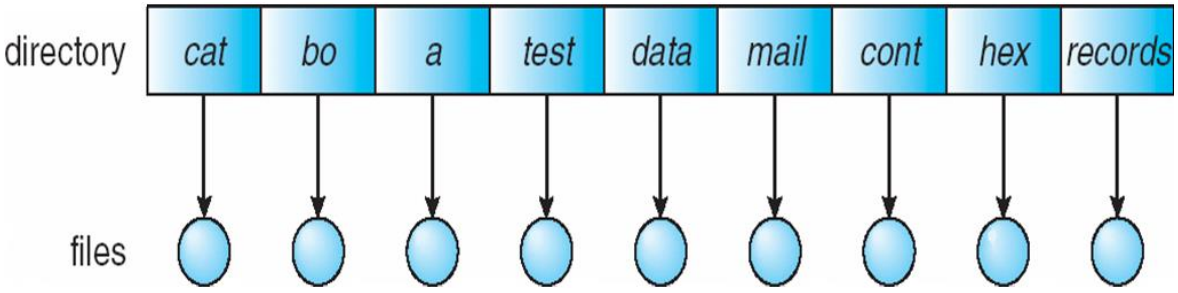


Both the directory structure and the files reside on disk.

Directory Structure:

Single-Level Directory

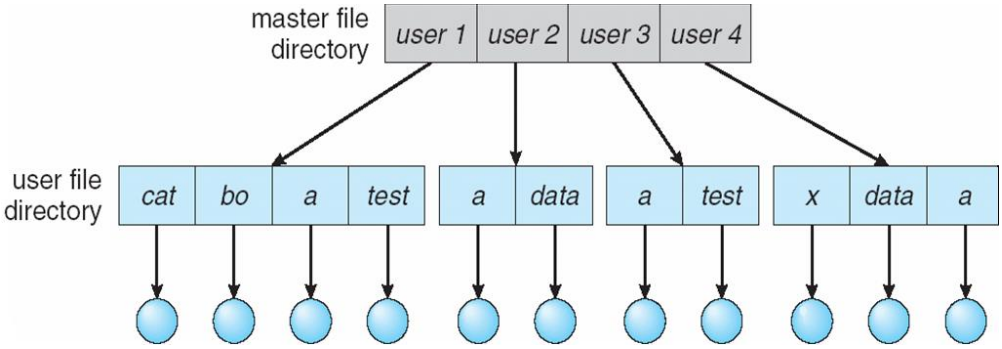
- Simplest.
- A single directory for all users
- Has significant limitations - when the number of files increases or when the system has more than one user.
- Since all files are in the same directory, they must have unique names. If two users call their data filet e s t . t x t , then the unique-name rule is violated.
- Even a single user on a single-level directory may find it difficult to remember the names of all the files as the number of files increases.
- It is not uncommon for a user to have hundreds of files on one computer system and an equal number of additional files on another system.
- Keeping track of so many files is a daunting task.



- Naming problem
- Grouping problem

Two-Level Directory:

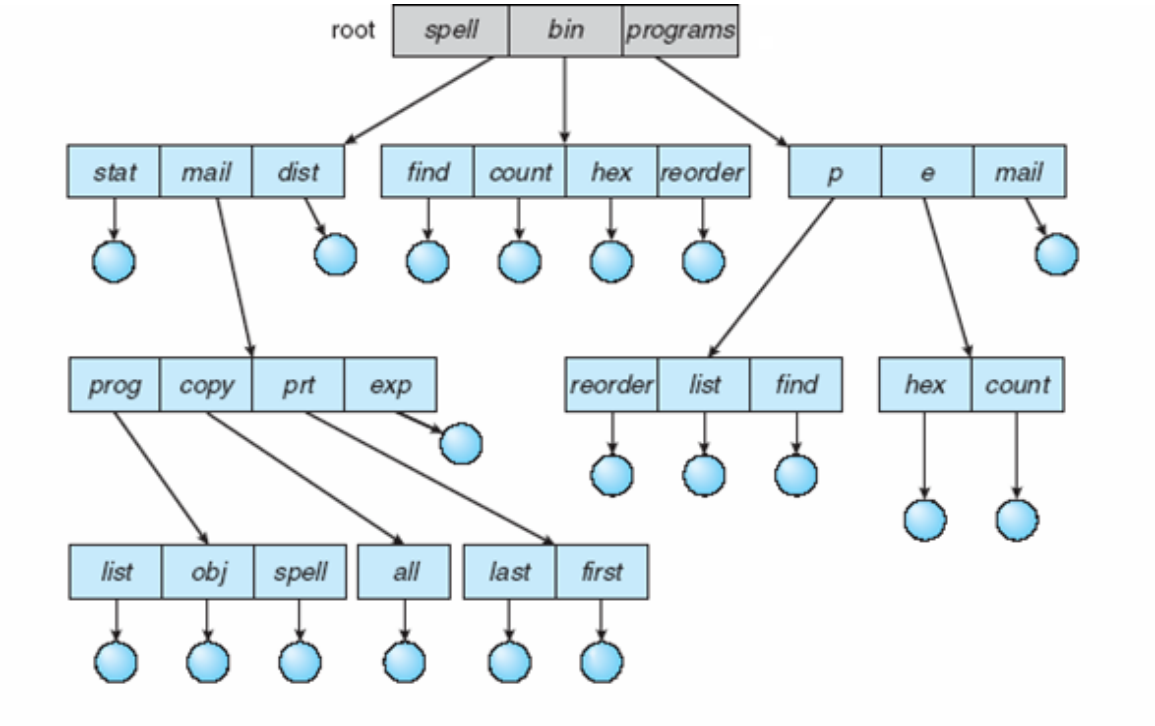
- Separate directory for each user

- 

- As, a single-level directory often leads to confusion of file names among different users. • The standard solution is to create a separate directory for each user.
- In the two-level directory structure, each user has own user file directory (UFD).
- The U F D s have similar structures, but each lists only the files of a single user.
- When a user job starts or a user logs in, the system's master file directory (M F D) is searched.
- When a user refers to a particular file, only his own U F D is searched.
- Thus, different users may have files with the same name, as long as all the file names within each U F D are unique.
- Thus, a user name and a file name define a path name
- Every file in the system has a path name. To name a file uniquely, a user must know the path name of the file desired.
- If the file is found, it is used.
- If it is not found, the system automatically searches the special user directory that contains the system files.
- Can have the same file name for different user
- Efficient searching
- No grouping capability

Tree – Structured Directories

- A tree is the most common directory structure.
- The tree has a root directory, and every file in the system has a unique path name.
- A directory (or subdirectory) contains a set of files or subdirectories. •
- ll directories have the same internal format. One bit in each directory entry defines the entry as a file (0) or as a subdirectory (1).



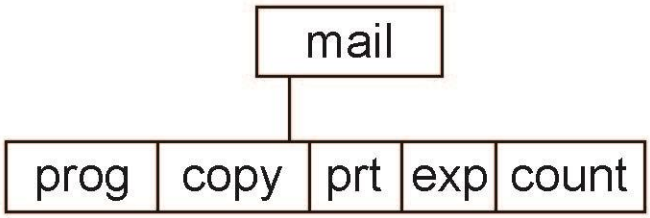
- Special system calls are used to create and delete directories.
- The current directory should contain most of the files that are of current interest to the process.
- Efficient searching
- Grouping Capability
- Current directory (working directory)
  - **cd /spell/mail/prog**
  - **type list**

• Path names can be of two types: absolute and relative.

-An absolute path name begins at the root and follows a path down to the specified file, giving the directory names on the path.

- A relative path name defines a path from the current directory.

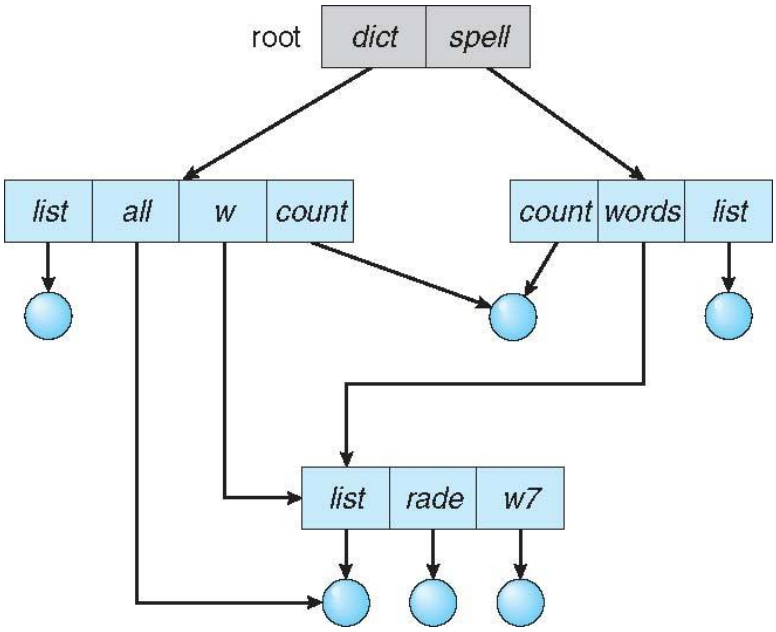
- Creating and deleting a new file is done in current directory in following way
- Delete a file
- **rm <file-name>**
- Creating a new subdirectory is done in current directory
- **mkdir <dir-name>**
- Example: if in current directory **/mail**
  mkdir count



Deleting "mail" ⇒ deleting the entire subtree rooted by "mail"

Acyclic-Graph Directories:

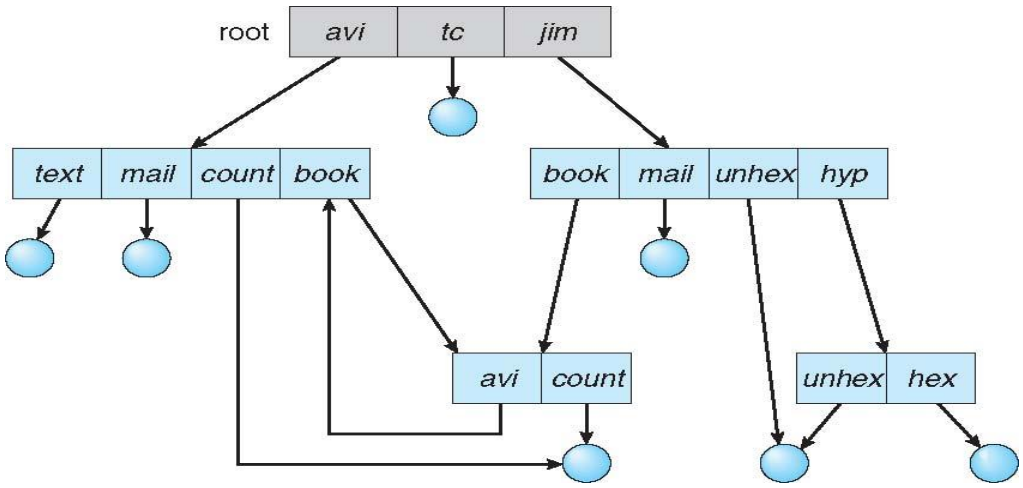
- An acyclic graph —that is, a graph with no cycles—allows directories to share subdirectories and files.
- The same file or subdirectory may be in two different directories. The acyclic graph is a natural generalization of the tree-structured directory scheme.
- It is important to note that a shared file (or directory) is not the same as two copies of the file.
- With two copies, each programmer can view the copy rather than the original, but if one programmer changes the file, the changes will not appear in the other's copy.
- With a shared file, only one actual file exists, so any changes made by one person are immediately visible to the other.



General Graph Directory:

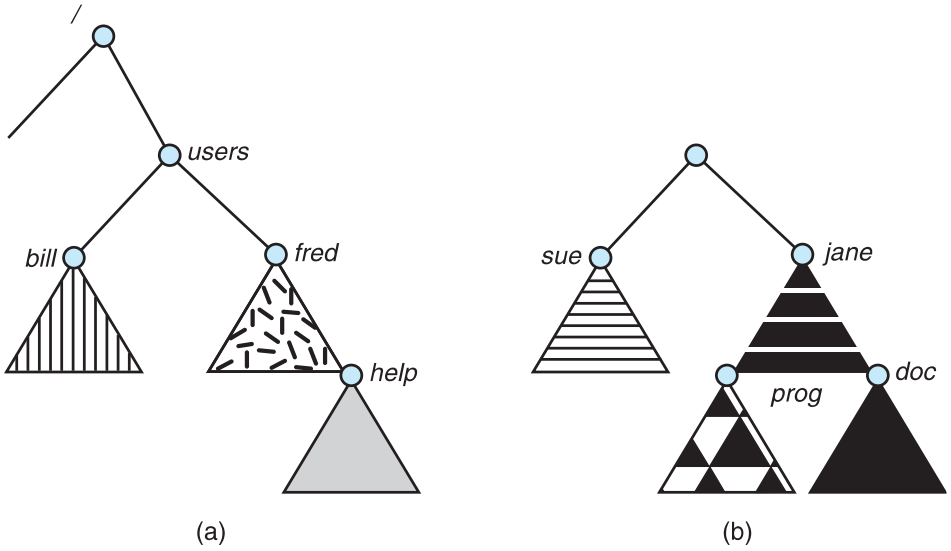
- A serious problem with using an acyclic-graph structure is ensuring that there are no cycles.
- If we start with a two-level directory and allow users to create subdirectories, a treestructured directory results.
- It should be fairly easy to see that simply adding new files and subdirectories to an existing tree-structured directory preserves the tree-structured nature

- However, when we add links, the tree structure is destroyed, resulting in a simple graph structure.

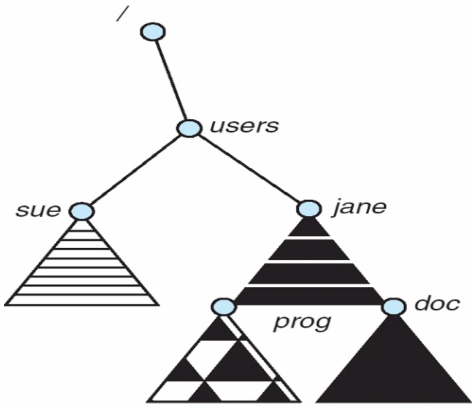


File System Mounting :

- A file system must be mounted before it can be accessed
- A unmounted file system (i.e. Fig.) is mounted at a mount point



(a) (b)



This is Mount point.

File Sharing

File sharing is very desirable for users who want to collaborate and to reduce the effort required to achieve a computing goal.

User-oriented operating systems must accommodate the need to share files in spite of the inherent difficulties.

- File sharing is very desirable for users who want to collaborate and to reduce the effort
- **Network File System (NFS)** is a common distributed file-sharing method

Multiple Users

- When an operating system accommodates multiple users, the issues of file sharing, file naming, and file protection become preeminent.
- Given a directory structure that allows files to be shared by users, the system must mediate the file sharing.
- The system can either allow a user to access the files of other users by default or require that a user specifically grant access to the files.
- To implement sharing and protection, the system must maintain more file and directory attributes than are needed on a single-user system.
- Most systems have evolved to use the concepts of file (or directory) owner (or user) and group.
- The owner is the user who can change attributes and grant access and who has most control over the file.
- The group attribute defines a subset of users who can share access to the file.
- The owner and group ID s of a given file (or directory) are stored with the other file attributes.
- When a user requests an operation on a file, the user ID can be compared with the owner attribute to determine if the requesting user is the owner of the file.

Remote File Systems

- Networking allows the sharing of resources spread across a campus or even around the world.
- One obvious resource to share is data in the form of files.
- Through the evolution of network and file technology, remote file-sharing methods have changed.
- The first implemented method involves manually transferring files between machines via programs like FTP.
- The second major method uses a distributed file system (D FS) in which remote directories are visible from a local machine.
- In some ways, the third method, the World Wide Web, is a reversion to the first.
- A browser is needed to gain access to the remote files, and separate operations (essentially a wrapper for ftp) are used to transfer files.
- ftp is used for both anonymous and authenticated access.

- Anonymous access allows a user to transfer files without having an account on the remote system.

The Client–Server Model:

- Remote file systems allow a computer to mount one or more file systems from one or more remote machines.
- In this case, the machine containing the files is the server, and the machine seeking access to the files is the client.
- The client–server relationship is common with networked machines.
- Generally, the server declares that a resource is available to clients and specifies exactly which resource (in this case, which files) and exactly which clients.
- A server can serve multiple clients, and a client can use multiple servers, depending on the implementation details of a given client–server facility. The server usually specifies the available files on a volume or directory level.
- Client identification is more difficult. A client can be specified by a network name or other identifier, such as an IP address, but these can be spoofed, or imitated.
- As a result of spoofing, an unauthorized client could be allowed access to the server. More secure solutions include secure authentication of the client via encrypted keys.

File Sharing – Failure Modes:

- All file systems have failure modes
  - For example corruption of directory structures or other non-user data, called **metadata**
- Remote file systems add new failure modes, due to network failure, server failure
- Local file systems can fail for a variety of reasons, including failure of the disk containing the file system, corruption of the directory structure or other disk-management information (collectively called **metadata**), disk-controller failure, cable failure, and host-adapter failure.
- User or system-administrator failure can also cause files to be lost or entire directories or volumes to be deleted.
- Many of these failures will cause a host to crash and an error condition to be displayed, and human intervention will be required to repair the damage.
- Remote file systems have even more failure modes.
- Because of the complexity of network systems and the required interactions between remote machines, many more problems can interfere with the proper operation of remote file systems.
- To implement this kind of recovery from failure, some kind of **state information** may be maintained on both the client and the server.
- If both server and client maintain knowledge of their current activities and open files, then they can seamlessly recover from a failure.

Consistency semantics

- Consistency semantics represent an important criterion for evaluating any file system that supports file sharing.
- In particular, they specify when modifications of data by one user will be observable by other users. Example: performing an atomic transaction to a remote disk could involve several network communications, several disk reads and writes, or both.
- Systems that attempt such a full set of functionalities tend to perform poorly.
- A successful implementation of complex sharing semantics can be found in the Andrew file system.
- For the following discussion, we assume that a series of file accesses (that is, reads and writes) attempted by a user to the same file is always enclosed between the open ( ) and close() operations. The series of accesses between the open() and close() operations.

Protection:

- File systems can be damaged by hardware problems (such as errors in reading or writing), power surges or failures, head crashes, dirt, temperature extremes, and vandalism.
- Files may be deleted accidentally.
- Protection can be provided in many ways. For a single-user laptop system, we might provide protection by locking the computer in a desk drawer or file cabinet.
- In a larger multiuser system, however, other mechanisms are needed

Types of Access

Protection mechanisms provide controlled access by limiting the types of file access that can be made.

Access is permitted or denied depending on several factors, one of which is the type of access requested. Several different types of operations may be controlled:

- **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.

Access Control

The most general scheme to implement identity-dependent access is to associate with each file and directory an access-control list (A C L) specifying user names and the types of access allowed for each user.

- When a user requests access to a particular file, the operating system checks the access list associated with that file.

- If that user is listed for the requested access, the access is allowed. Otherwise, a protection violation occurs, and the user job is denied access to the file

Access Lists and Groups:

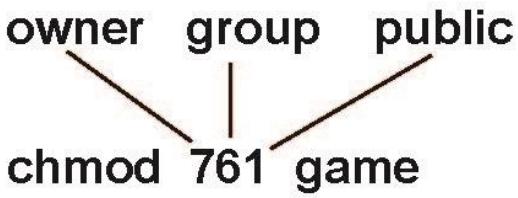
- Mode of access: read, write, execute
- Three classes of users on Unix / Linux

	RWX		
a) **owner access**	7	⇒	1 1 1
	RWX		
b) **group access**	6	⇒	1 1 0
	RWX		
c) **public access**	1	⇒	0 0 1

To condense the length of the access-control list, many systems recognize three classifications of users in connection with each file:

- **Owner**. The user who created the file is the owner.
- **Group**. A set of users who are sharing the file and need similar access is a group, or work group.
- **Public or Universe.** All other users in the system constitute the universe.

- Ask manager to create a group (unique name), say G, and add some users to the group.
- For a particular file (say *game*) or subdirectory, define an appropriate access.



Other Protection Approaches

Another approach to the protection problem is to associate a password with each file. If the passwords are chosen randomly and changed often, this scheme may be effective in limiting access to a file. The use of passwords has a few disadvantages.

- First, the number of passwords that a user needs to remember may become large, making the scheme impractical.
- Second, if only one password is used for all the files, then once it is discovered, all files are accessible.

File System Structure:

File structure:

- File structure
  - Logical storage unit

  - Collection of related information
- **File system** resides on secondary storage (disks)
  - Provided user interface to storage, mapping logical to physical
  - Provides efficient and convenient access to disk by allowing data to be stored, located retrieved easily
- Disk provides in-place rewrite and random access
  - I/O transfers performed in **blocks** of **sectors** (usually 512 bytes)
- **File control block** – storage structure consisting of information about a file
- **Device driver** controls the physical device
- File system organized into layers

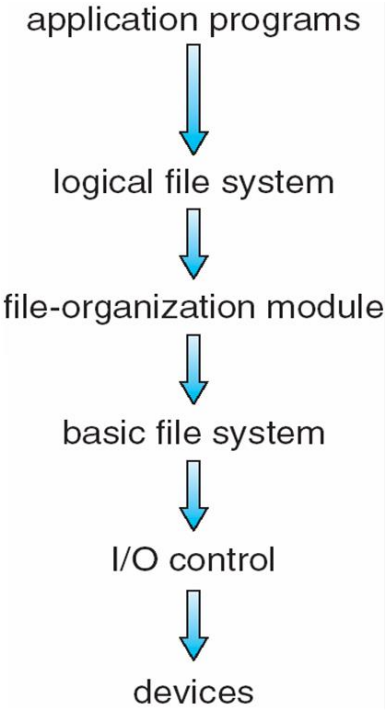
File System Structure

File System provide efficient access to the disk by allowing data to be stored, located and retrieved in a convenient way. A file System must be able to store the file, locate the file and retrieve the file.

Most of the Operating Systems use layering approach for every task including file systems. Every layer of the file system is responsible for some activities.

The image shown below, elaborates how the file system is divided in different layers, and also the functionality of each layer.

Layered File System:



- When an application program asks for a file, the first request is directed to the logical file system. The logical file system contains the Meta data of the file and directory structure. If the application program doesn't have the required permissions of the file then this layer will throw an error. Logical file systems also verify the path to the file.

- Generally, files are divided into various logical blocks. Files are to be stored in the hard disk and to be retrieved from the hard disk. Hard disk is divided into various tracks and sectors. Therefore, in order to store and retrieve the files, the logical blocks need to be mapped to physical blocks. This mapping is done by File organization module. It is also responsible for free space management.
- Once File organization module decided which physical block the application program needs, it passes this information to basic file system. The basic file system is responsible for issuing the commands to I/O control in order to fetch those blocks.
- I/O controls contain the codes by using which it can access hard disk. These codes are known as device drivers. I/O controls are also responsible for handling interrupts.

File System Layers:

- **Device drivers** manage I/O devices at the I/O control layer
  - Given commands like "read drive1, cylinder 72, track 2, sector 10, into memory location 1060" outputs low-level hardware specific commands to hardware controller
- **Basic file system** given command like "retrieve block 123" translates to device driver
- Also manages memory buffers and caches (allocation, freeing, replacement)
  - Buffers hold data in transit
  - Caches hold frequently used data
- **File organization module** understands files, logical address, and physical blocks
  - Translates logical block # to physical block #
  - Manages free space, disk allocation

- **Logical file system** manages metadata information
  - Translates file name into file number, file handle, location by maintaining file control blocks
- Layering useful for reducing complexity and redundancy, but adds overhead and can decrease performance
- Logical layers can be implemented by any coding method according to OS designer

File-System Implementation:

- We have using two types data structures we can implement
  1)On-disk Structures and
  2)In-memory structures

On-disk Structures –

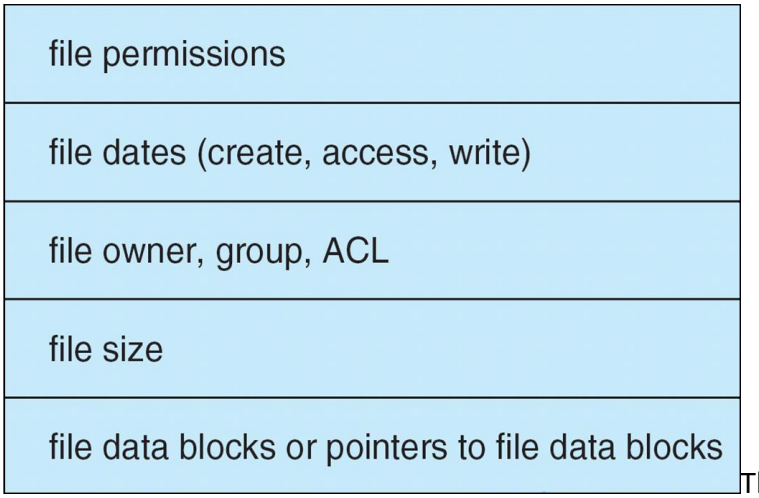
Generally they contain information about total number of disk blocks, free disk blocks, location of them and etc. Below given are different on-disk structures :

1. **Boot Control Block –**
   It is usually the first block of volume and it contains information needed to

boot an operating system. In UNIX it is called boot block and in NTFS it is called as partition boot sector.

2. **Volume Control Block –**
   It has information about a particular partition. Ex:- free block count, block size and block pointers, total number of blocks, # of free blocks, block size, free block pointers or array etc. In UNIX it is called super block and in NTFS it is stored in master file table.
3. **Directory Structure –**
   They store file names and associated inode numbers, master file table . In UNIX, includes file names and associated file names and in NTFS, it is stored in master file table.
4. **Per-File FCB –**
   It contains details about files and it has a unique identifier number to allow association with directory entry. In NTFS it is stored in master file table.

- Per-file **File Control Block (FCB)** contains many details about the file
  - inode number, permissions, size, dates



This is FCB

2. In-Memory Structure :

They are maintained in main-memory and these are helpful for file system management for caching. Several in-memory structures given below :

Mount Table –

It contains information about each mounted volume. Mount table stores file system mounts, mount points, file system types

Directory-Structure cache –

This cache holds the directory information of recently accessed directories.

System wide open-file table –

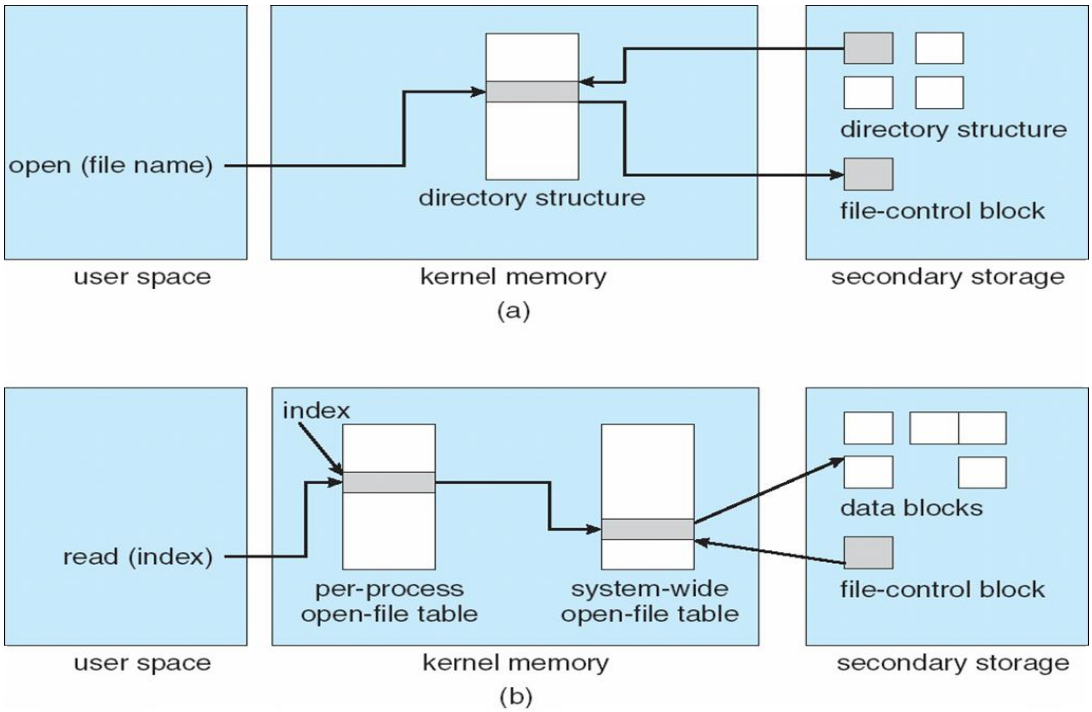
It contains the copy of FCB of each open file.

Per-process open-file table –

It contains information opened by that particular process and it maps with appropriate system wide open-file.

- Plus buffers hold data blocks from secondary storage

- Open returns a file handle for subsequent use
- Data from read eventually copied to specified user process memory address
- The following figure illustrates the necessary file system structures provided by the operating systems
- Figure (a) refers to opening a file
- Figure (b) refers to reading a file



Directory Implementation:

a) Linear List

- The simplest method of implementing a directory is to use a linear list of file names with pointers to the data blocks. This method is simple to program but time-consuming to execute.
- To create a new file, we must first search the directory to be sure that no existing file has the same name. Then, we add a new entry at the end of the directory.
- To delete a file, we search the directory for the named file and then release the space allocated to it.
- To reuse the directory entry, we can do one of several things. We can mark the entry as unused (by assigning it a special name, such as an all-blank name, or by including a used– unused bit in each entry), or we can attach it to a list of free directory entries.
- A third alternative is to copy the last entry in the directory into the freed location and to decrease the length of the directory.
- Linear list is Simple to program . But it is Time-consuming to execute

- A linked list can also be used to decrease the time required to delete a file. The real disadvantage of a linear list of directory entries is that finding a file requires a linear search.
- A sorted list allows a binary search and decreases the average search time.
- An advantage of the sorted list is that a sorted directory listing can be produced without a separate sort step.

b) Hash Table –

- The hash table takes a value computed from the file name and returns a pointer to the file name in the linear list. Therefore, it can greatly decrease the directory search time.
- Insertion and deletion are also fairly straightforward, although some provision must be made for collisions—situations in which two file names hash to the same location.
- The major difficulties with a hash table are its generally fixed size and the dependence of the hash function on that size.
- Decreases directory search time
- **Collisions** – situations where two file names hash to the same location
- Only good if entries are fixed size, or use chained-overflow method

Allocation Methods:

- The allocation methods define how the files are stored in the disk blocks. There are three main disk space or file allocation methods.
    1. Contiguous or Sequenced Allocation
    2. Linked Allocation
    3. Indexed Allocation

The main idea behind these methods is to provide:

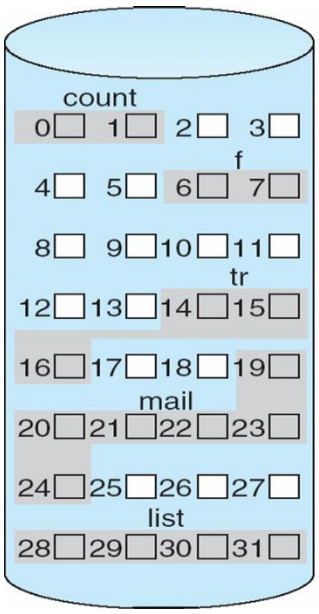
- Efficient disk space utilization.
- Fast access to the file blocks.
- All the three methods have their own advantages and disadvantages as discussed below:

1. Contiguous Allocation or Sequenced Allocation:

- In this scheme, each file occupies a contiguous set of blocks on the disk.
- Each file is stored in sequential memory blocks &that are next to each other.
    - Best performance in most cases
    - It is Simple – only starting location (block no) and length (number of blocks) are required
- For example, if a file requires “n” blocks and is given a block “b” as the starting location, then the blocks assigned to the file will be: b, b+1, b+2,……b+n-1.
- The directory entry for a file with contiguous allocation contains
    - Address of starting block
    - Length of the allocated portion.

- Easy to implement.
- External fragmentation is a major issue with this type of allocation technique.

- When head movement is needed (from the last sector of one cylinder to the first sector of the next cylinder), the head need only move from one track to the next. Thus, the number of disk seeks required for accessing contiguously allocated files is minimal, as is seek time when a seek is finally needed.
- Contiguous allocation has some problems, however. One difficulty is finding space for a new file



directory

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

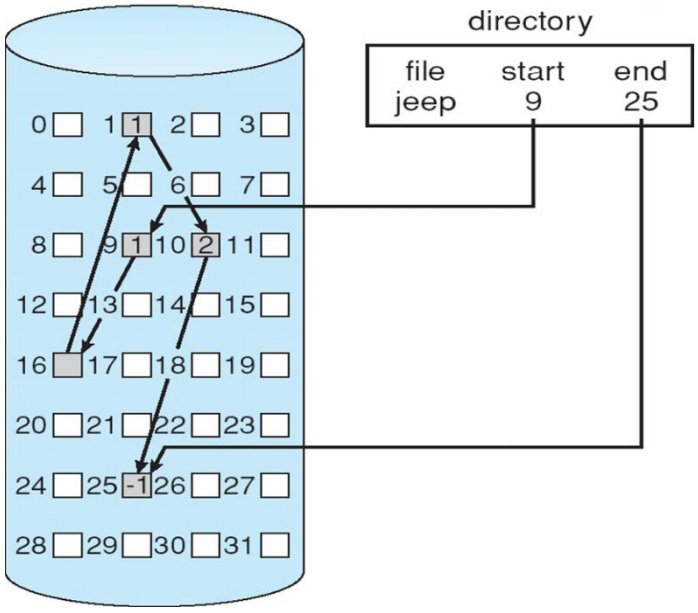
The *file 'mail'* in the following figure starts from the block 19 with length = 6 blocks. Therefore, it occupies *19, 20, 21, 22, 23, 24* blocks.

- The contiguous-allocation problem can be seen as a particular application of the general dynamic storage-allocation which involves how to satisfy a request of size n from a list of free holes.
- First fit and best fit are the most common strategies used to select a free hole from the set of available holes. All these algorithms suffer from the problem of external fragmentation. As files are allocated and deleted, the free disk space is broken into little pieces.

2) Linked allocation –

- Linked allocation solves all problems of contiguous allocation.
- With linked allocation, each file is a linked list of disk blocks; the disk blocks may be scattered anywhere on the disk.
- In this scheme, each file is a linked list of disk blocks which **need not be** contiguous. The disk blocks can be scattered anywhere on the disk
- The directory entry contains a pointer to the first and the last blocks of each file.
- Each block contains a pointer to the next block occupied by the file.

- The directory contains a pointer to the first and last blocks of the each file
  - File ends at null pointer
  - No external fragmentation
  - Each block contains pointer to next block
  - No compaction, external fragmentation
  - Free space management system called when new block needed
- Example: A file of five blocks might start at block 9 and continue at block 16, then block 1, then block 10, and finally block 25



The file 'jeep' in following image shows how the blocks are randomly distributed. The last block (25) contains -1 indicating a null pointer and does not point to any other block.

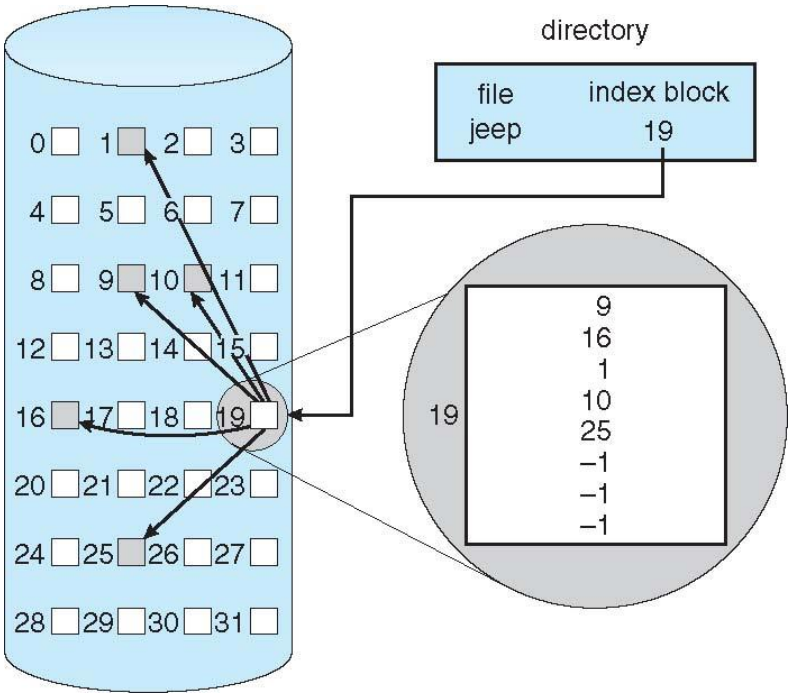
Advantages of Linked Allocation

- This is very flexible in terms of file size. File size can be increased easily since the system does not have to look for a contiguous chunk of memory.
- This method does not suffer from external fragmentation. This makes it relatively better in terms of memory utilization.

3) Indexed allocation

  - Each file has its own **index block**(s) of pointers to its data blocks
- In this scheme, a special block known as the **Index block** contains the pointers to all the blocks occupied by a file. Each file has its own index block. The "i" th entry in the index block contains the disk address of the "i" th file block. The directory entry contains the address of the index block as shown in the image:
- Linked allocation solves the external-fragmentation and size-declaration problems of contiguous allocation
- Linked allocation cannot support efficient direct access, since the pointers to the blocks are scattered with the blocks themselves all over the disk and must be retrieved in order.

- Indexed allocation solves this problem 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 ith entry in the index block points to the ith block of the file.
- The directory contains the address of the index block .To find and read the ith block, we use the pointer in the ith index-block entry.
- When the file is created, all pointers in the index block are set to null. When the ith block is first written, a block is obtained from the free-space manager, and its address is put in the ith index block entry.



- Indexed Allocation provides solutions to problems of contiguous and linked allocation.
- A index block is created having all pointers to files.
- Each file has its own index block which stores the addresses of disk space occupied by the file.
- Directory contains the addresses of index blocks of files
- Need index table
- Random access
- Dynamic access without external fragmentation, but have overhead of index block

Advantages for Indexed allocation

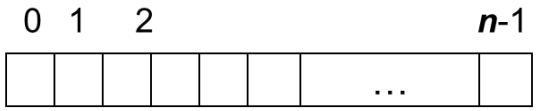
- This supports direct access to the blocks occupied by the file and therefore provides fast access to the file blocks.
- It overcomes the problem of external fragmentation.

Free-Space Management:

- To keep track of free disk space, the system maintains a free-space list.
- The free-space list records all free disk blocks—those not allocated to some file or directory. To create a file, we search the free-space list for the required amount of space and allocate that space to the new file.
- This space is then removed from the free-space list. When a file is deleted, its disk space is added to the free-space list.

a) Bit Vector

- Frequently, 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.
- Example: Consider a disk where blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26, and 27 are free and the rest of the blocks are allocated. The free-space bit map would be 00111100111111000110000001110000 0...
- The main advantage of this approach is its relative simplicity and its efficiency in finding the first free block or n consecutive free blocks on the disk
- The first non-0 word is scanned for the first 1 bit, which is the location of the first free block.
- The calculation of the block number is (number of bits per word) × (number of 0-value words) + offset of first 1 bit.
- **Bit vector** or **bit map** (***n*** blocks)

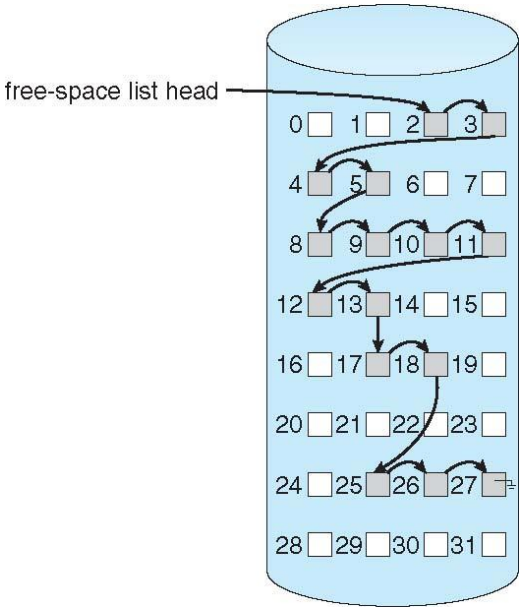


$$bit[i] = \begin{cases} 1 \Rightarrow block[i] \text{ free} \\ 0 \Rightarrow block[i] \text{ occupied} \end{cases}$$

b) Linked Free Space List on Disk:

- In a Linked List keeping a pointer to the first free block in a special location on the disk and caching it in memory. This first block contains a pointer to the next free disk block, and so on.
- 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26, and 27 were free and the rest of the blocks were allocated. In this situation, we would keep a pointer to block 2 as the first free block. Block 2 would contain a pointer to block 3, which would point to block 4, which would point to block 5, which would point to block 8, and so on.
- Linked list (free list)
- Cannot get contiguous space easily
- No waste of space
- No need to traverse the entire list (if # free blocks recorded)

Diagram for Linked Free Space List on Disk:



Grouping:

- A modification of the free-list approach stores the addresses of n free blocks in the first free block. The first n−1 of these blocks are actually free.
- The last block contains the addresses of another n free blocks, and so on.
- The addresses of a large number of free blocks can now be found quickly, unlike the situation when the standard linked-list approach is used.
- Modify linked list to store address of next *n-1* free blocks in first free block, plus a pointer to next block that contains free-block-pointers (like this one)

Counting:

- Another approach takes advantage of the fact that, generally, several contiguous blocks may be allocated or freed simultaneously, particularly when space is allocated with the contiguous allocation algorithm or through clustering.
- Thus, rather than keeping a list of n free disk addresses, we can keep the address of the first free block and the number (n) of free contiguous blocks that follow the first block. Each entry in the free-space list then consists of a disk address and a count.
- Because space is frequently contiguously used and freed, with contiguous-allocation allocation, extents, or clustering
    - Keep address of first free block and count of following free blocks
    - Free space list then has entries containing addresses and counts

Efficiency and Performance:

- Efficiency dependent on:
    - Disk allocation and directory algorithms

  - Types of data kept in file's directory entry
  - Pre-allocation or as-needed allocation of metadata structures
  - Fixed-size or varying-size data structures
- Performance
  - Keeping data and metadata close together
  - **Buffer cache** – separate section of main memory for frequently used blocks
  - **Synchronous** writes sometimes requested by apps or needed by OS
    - No buffering / caching – writes must hit disk before acknowledgement
    - **Asynchronous** writes more common, buffer-able, faster
  - **Free-behind** and **read-ahead** – techniques to optimize sequential access
  - Reads frequently slower than writes