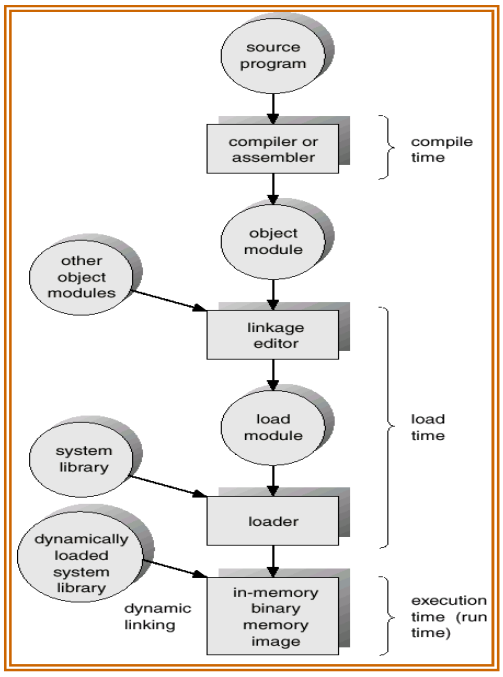
Memory Management

- Memory consists of a large array of words or bytes, each with its own address. The CPU fetches instructions from memory according to the value of the program counter. These instructions may cause additional loading from and storing to specific memory addresses.
- Memory unit sees only a stream of memory addresses. It does not know how they are generated.
- Program must be brought into memory and placed within a process for it to be run.

- Input queue – collection of processes on the disk that are waiting to be brought into memory for execution. User programs go through several steps before being run.



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

- **Compile time**: If memory location known a priori, absolute code can be generated; must recompile code if starting location changes.

  Example: .COM-format programs in MS-DOS.
- **Load time**: Must generate relocatable code if memory location is not known at compile time.
- **Execution time**: Binding delayed until run time if the process can be moved during its execution from one memory segment to another. Need hardware support for address maps (e.g., relocation registers).

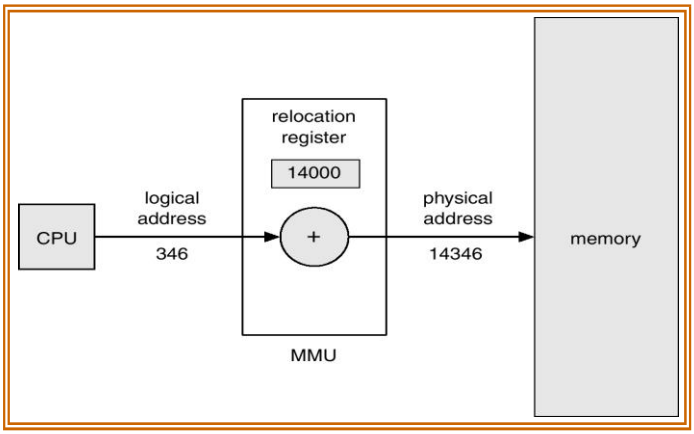
Logical Versus Physical Address Space

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

- Logical address – address generated by the CPU; also referred to as virtual address.
- Physical address – address seen by the memory unit.

- The set of all logical addresses generated by a program is a logical address space; the set of all physical addresses corresponding to these logical addresses are a physical address space.

- Logical and physical addresses are the same in compile-time and load-time address-binding schemes; logical (virtual) and physical addresses differ in execution-time address-binding scheme.
- The run-time mapping from virtual to physical addresses is done by a hardware device called the memory management unit (MMU).



- This method requires hardware support slightly different from the hardware configuration. The base register is now called a relocation register. The value in the relocation register is added to every address generated by a user process at the time it is sent to memory.
- The user program never sees the real physical addresses. The program can create a pointer to location 346, store it in memory, manipulate it and compare it to other addresses. The user program deals with logical addresses. The memory mapping hardware converts logical addresses into physical addresses. The final location of a referenced memory address is not determined until the reference is made.

Dynamic Loading

- Routine is not loaded until it is called.
- 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 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.

- Better memory-space utilization; unused routine is never loaded.

- Useful when large amounts of code are needed to handle infrequently occurring cases.
- No special support from the operating system is required.
- Implemented through program design.

Dynamic Linking

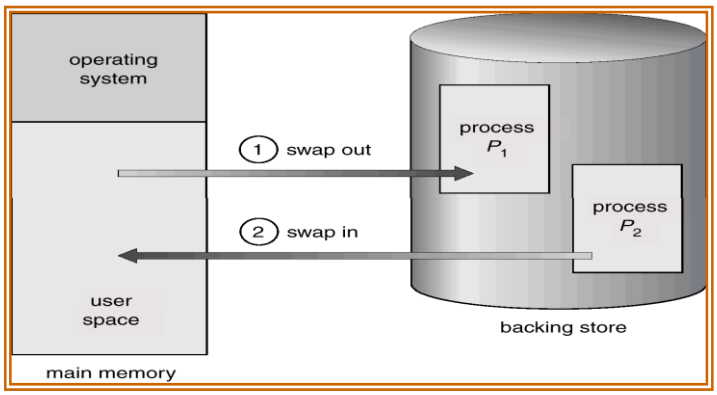
- Linking is postponed until execution time.
- Small piece of code, stub, is used to locate the appropriate memory-resident library routine, or to load the library if the routine is not already present.
- When this stub is executed, it checks to see whether the needed routine is already in memory. If not, the program loads the routine into memory.
- Stub replaces itself with the address of the routine, and executes the routine.
- Thus the next time that code segment is reached, the library routine is executed directly, incurring no cost for dynamic linking.
- Operating system is needed to check if routine is in processes' memory address.
- Dynamic linking is particularly useful for libraries.

Swapping

- A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution. For example, assume a multiprogramming environment with a round robin CPU scheduling algorithm. When a quantum expires, the memory manager will start to swap out the process that just finished, and to swap in another process to the memory space that has been freed. In the mean time, the CPU scheduler will allocate a time slice to some other process in memory. When each process finished its quantum, it will be swapped with another process. Ideally, the memory manager can swap processes fast enough that some processes will be in memory, ready to execute, when the CPU scheduler wants to reschedule the CPU. The quantum must also be sufficiently large that reasonable amounts of computing are done between swaps.
- Roll out, roll in – swapping variant used for priority-based scheduling algorithms. If a higher priority process arrives and wants service, the memory manager can swap out the lower priority process so that it can load and execute lower priority process can be swapped back in and continued. This variant is some times called roll out, roll in. Normally a process that is swapped out will be swapped back into the same memory space that it occupied previously. This restriction is dictated by the process cannot be moved to different locations. If execution time

binding is being used, then a process can be swapped into a different memory space, because the physical addresses are computed during execution time.

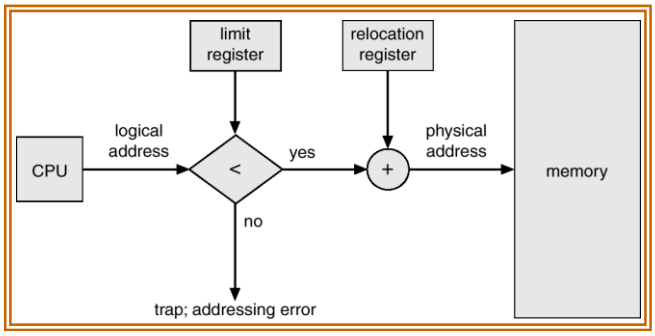
- Backing store – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images. It must be large enough to accommodate copies of all memory images for all users, and it must provide direct access to these memory images. The system maintains a ready queue consisting of all processes whose memory images are scheduler decides to execute a process it calls the dispatcher. The dispatcher checks to see whether the next process in the queue is in memory. If not, and there is no free memory region, the dispatcher swaps out a process currently in memory and swaps in the desired process. It then reloads registers as normal and transfers control to the selected process.
- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped.
- Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows).



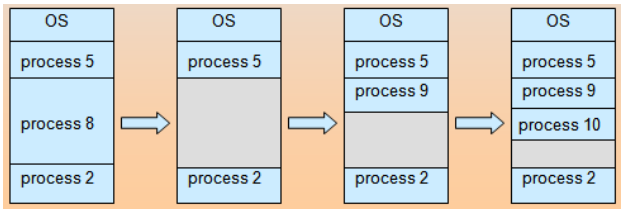
Contiguous Memory Allocation

- Main memory is usually divided into two partitions:
  - Resident operating system, usually held in low memory with interrupt vector.
  - User processes, held in high memory.
- In contiguous memory allocation, each process is contained in a single contiguous section of memory.
- Single-partition allocation
  - Relocation-register scheme used to protect user processes from each other, and from changing operating-system code and data.

- Relocation register contains value of smallest physical address; limit register contains range of logical addresses – each logical address must be less than the limit register.



- Multiple-partition allocation
  - 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.
  - Operating system maintains information about:
    a) allocated partitions b) free partitions (hole)
  - A set of holes of various sizes, is scattered throughout memory at any given time. When a process arrives and needs memory, the system searches this set for a hole that is large enough for this process. If the hole is too large, it is split into two: one part is allocated to the arriving process; the other is returned to the set of holes. When a process terminates, it releases its block of memory, which is then placed back in the set of holes. If the new hold 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 is how to satisfy a request of size n from a list of free holes. There are many solutions to this problem. The set of holes is searched to determine which hole is best to allocate. The first-fit, best-fit and worst-fit strategies are the most common ones used to select a free hole from the set of available holes.



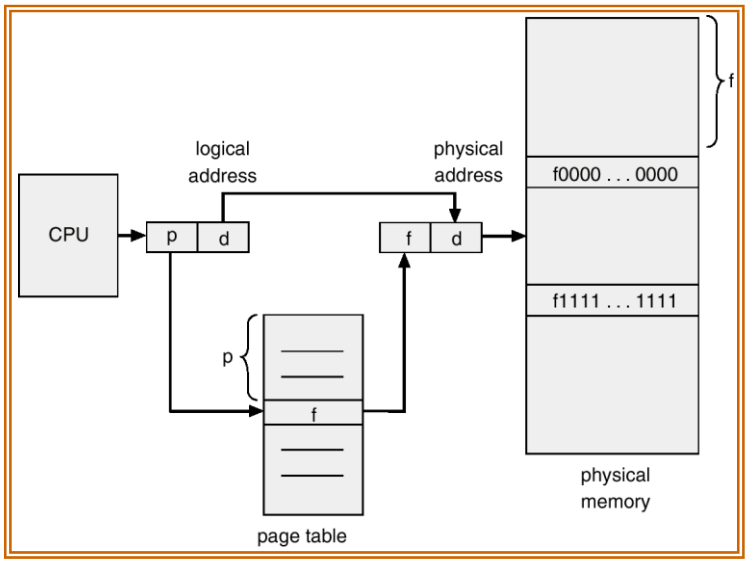
- **First-fit:** Allocate the first hole that is big enough.
- **Best-fit:** Allocate the smallest hole that is big enough; must search entire list, unless ordered by size.
- **Worst-fit:** Allocate the largest hole; must also search entire list.

Fragmentation

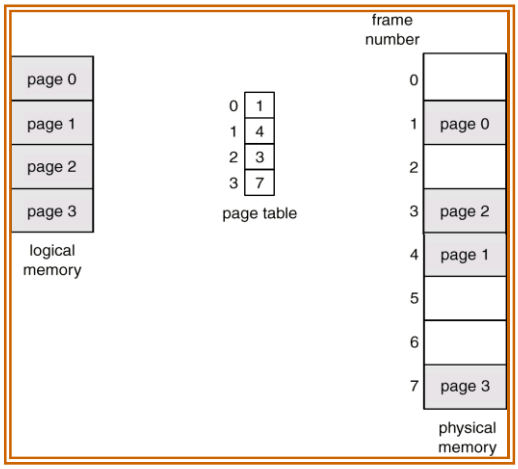
- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous.
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used.
- Reduce external fragmentation by compaction
  - Shuffle memory contents to place all free memory together in one large block.
  - Compaction is possible only if relocation is dynamic, and is done at execution time.

Paging

- Paging is a memory management scheme that permits the physical address space of a process to be non contiguous.
- Divide physical memory into fixed-sized blocks called **frames** (size is power of 2, for example 512 bytes).
- Divide logical memory into blocks of same size called **pages**. When a process is to be executed, its pages are loaded into any available memory frames from the backing store. The backing store is divided into fixed sized blocks that are of the same size as the memory frames.
- The hardware support for paging is illustrated in below figure.
- Every address generated by the CPU is divided into two parts: a page number (p) and a page offset (d). The page number is used as an index into a page table. The page table contains the base address of each page in physical memory. This base address is combined with the page offset to define the physical memory address that is sent to the memory unit.



- The paging model of memory is shown in below figure. The page size is defined by the hardware. The size of a page is typically of a power of 2, varying between 512 bytes and 16 MB per page, depending on the computer architecture. The selection of a power of 2 as a page size makes the translation of a logical address into a page number and page offset particularly easy. If the size of logical address is 2^m, and a page size is 2^n addressing units, then the high order m-n bits of a logical address designate the page number, and the n low order bits designate the page offset.



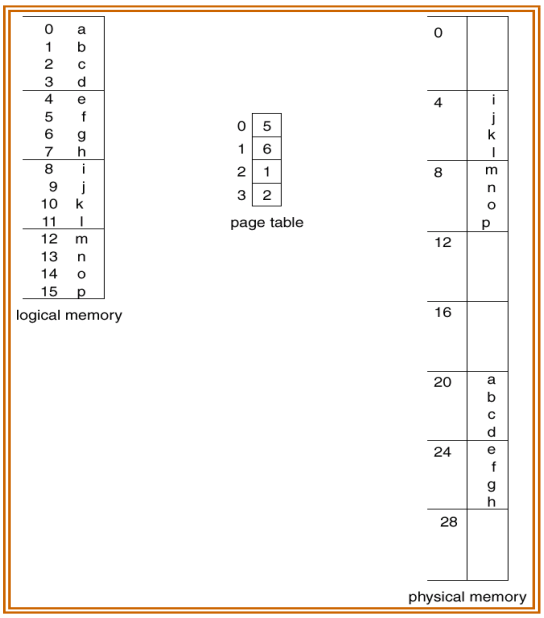
- Keep track of all free frames.
- To run a program of size n pages, need to find n free frames and load program.
- Set up a page table to translate logical to physical addresses.
- Internal fragmentation may occur.

Let us take an example. Suppose a program needs 32 KB memory for allocation. The whole program is divided into smaller units assuming 4 KB and is assigned some address. The address consists of two parts such as:

- A large number in higher order positions and
- Displacement or offset in the lower order bits.

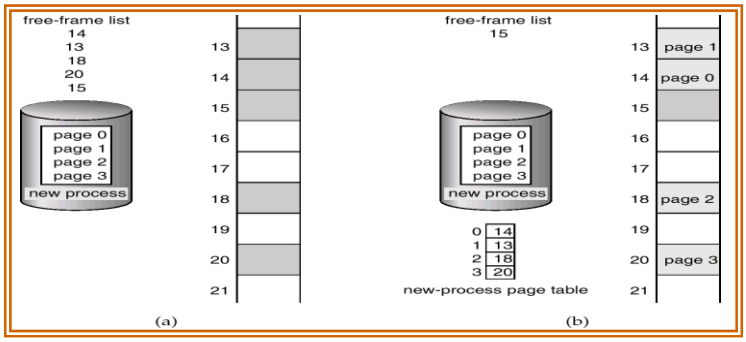
The numbers allocated to pages are typically in power of 2 to simplify extraction of page numbers and offsets. To access a piece of data at a given address, the system first extracts the page number and the offset. Then it translates the page number to physical page frame and access data at offset in physical page frame. At this moment, the translation of the address by the OS is done using a page table. Page table is a linear array indexed by virtual page number which provides the physical page frame that contains the particular page. It employs a lookup process that extracts the page number and the offset. The system in addition checks that the page number is within the address space of process and retrieves the page number in the page table. Physical address will calculated by using the formula.

Physical address = page size of logical memory X frame number + offset



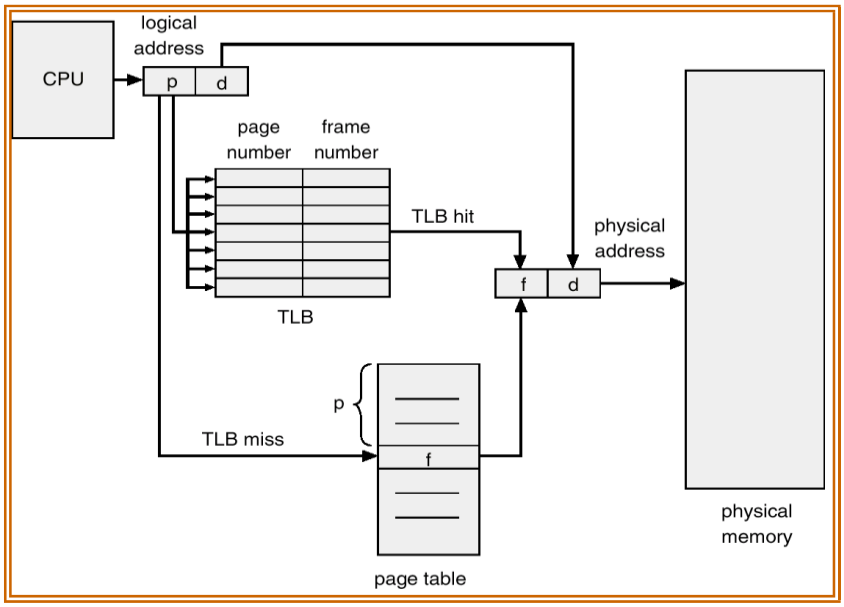
When a process arrives in the system to be executed, its size expressed in pages is examined. Each page of the process needs one frame. Thus if the process requires n pages, at least n frames must be

available in memory. If n frames are available, they are allocated to this arriving process. The first page of the process is loaded into one of the allocated frames, and the frame number is put in the page table for this process. The next page is loaded into another frame, and its frame number is put into the page table and so on as in below figure. 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 that memory as one single contiguous space, containing only this one program. In fact, the user program is scattered throughout physical memory, which also holds other programs. The difference between the user's view of memory and the actual physical memory is reconciled by the address-translation hardware. The logical addresses are translated into physical addresses. This mapping is hidden from the user and is controlled by the operating system.



Implementation of Page Table

- Page table is kept in main memory.
- Page-tablebase register (PTBR) points to the page table.
- In this scheme every data/instruction-byte access requires two memory accesses. One for the page-table entry and one for the byte.
- The two memory access problem can be solved by the use of a special fast-lookup hardware cache called associative registers or associative memory or translation look-aside buffers(TLBs).
- Typically, the number of entries in a TLB is between 32 and 1024.



- 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 found, its frame number is immediately available and is used to access memory. The whole task may take less than 10 percent longer than it would if an unmapped memory reference were used.
- If the page number is not in the TLB (known as a TLB miss), a memory reference to the page table must be made. When the frame number is obtained, we can use it to access memory.

Hit Ratio

- Hit Ratio: the percentage of times that a page number is found in the associative registers.
- For example, if it takes 20 nanoseconds to search the associative memory and 100 nanoseconds to access memory; for a 98-percent hit ratio, we have

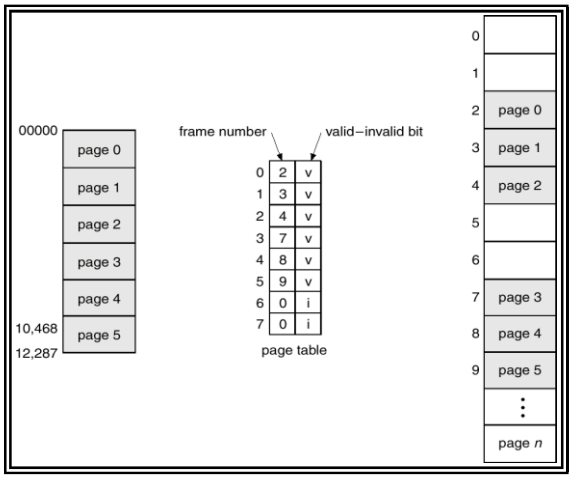
  Effective memory-access time = 0.98 x 120 + 0.02 x 220

  = 122 nanoseconds.
- The Intel 80486 CPU has 32 associative registers, and claims a 98-percent hit ratio.

Valid or invalid bit in a page table

- Memory protection implemented by associating protection bit with each frame.
- Valid-invalid bit attached to each entry in the page table:
  - "Valid" indicates that the associated page is in the process' logical address space, and is thus a legal page.
  - "Invalid" indicates that the page is not in the process' logical address space.

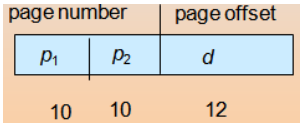
- Pay attention to the following figure. The program extends to only address 10,468, any reference beyond that address is illegal. However, references to page 5 are classified as valid, so accesses to addresses up to 12,287 are valid. This reflects the internal fragmentation of paging.



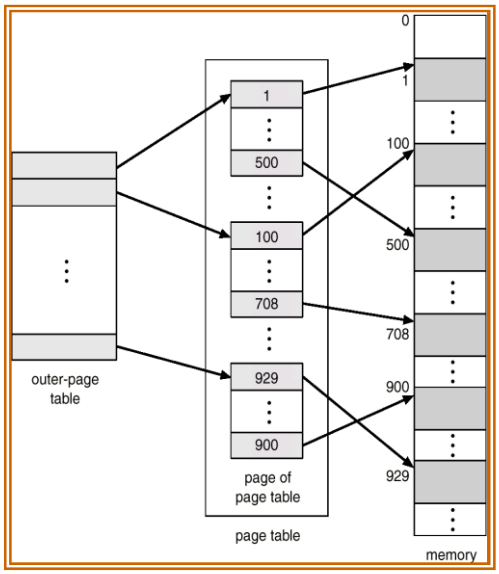
Structure of the Page Table

Hierarchical Paging:

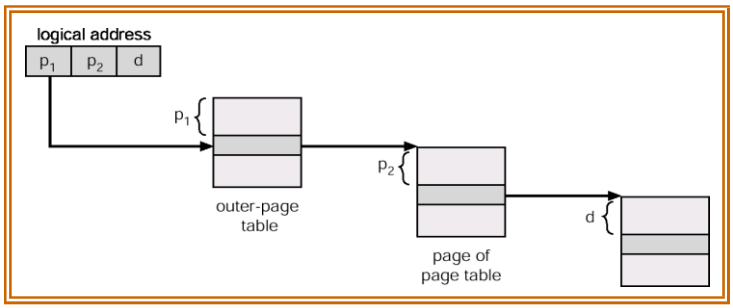
- A logical address (on 32-bit machine with 4K page size) is divided into:
  - A page number consisting of 20 bits.
  - A page offset consisting of 12 bits.
- Since the page table is paged, the page number is further divided into:
  - A 10-bit page number.
  - A 10-bit page offset.
- Thus, a logical 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 outer page table.The below figure shows a two level page table scheme.

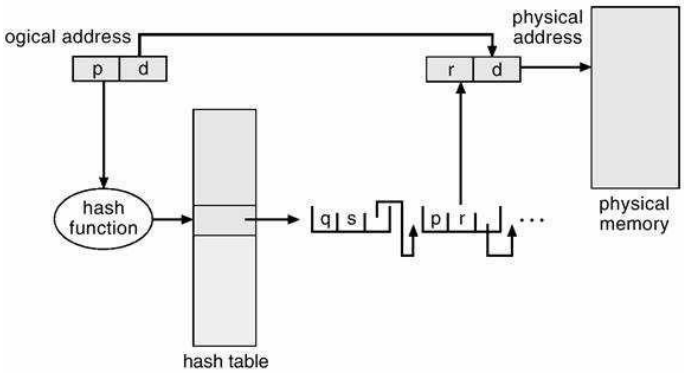


Address-translation scheme for a two-level 32-bit paging architecture is shown in below figure.



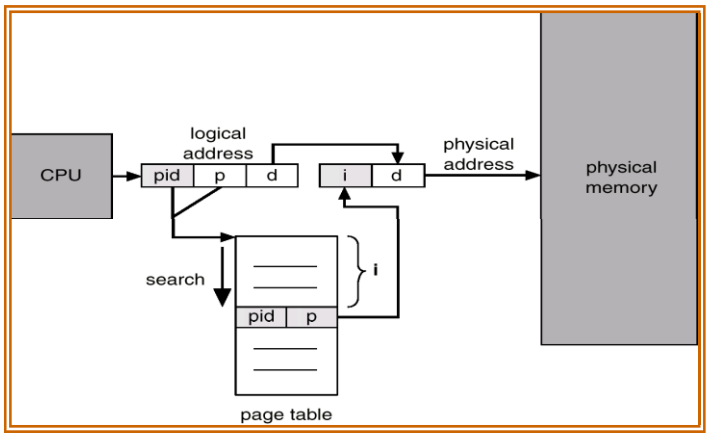
Hashed Page Table:

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 has to the same location. Each element consists of three fields: (a) the virtual page number, (b) the value of the mapped page frame, and (c) a pointer to the next element in the linked list. The algorithm works as follows: The virtual page number in the virtual address is hashed into the hash table. The virtual page number is compared to field (a) in the first element in the linked list. If there is a match, the corresponding page frame (field (b)) is used to form the desired physical address. If there is no match, subsequent entries in the linked list are searched for a matching virtual page number. The scheme is shown in below figure.



Inverted Page Table:

- One entry for each real page (frame) 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.
- There is only one page table in the system. Not per process.
- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs.
- Use hash table to limit the search to one — or at most a few — page-table entries.



Each virtual address in the system consists of a triple <process-id, page-number, offset>. Each inverted page table entry is a pair <process-id, page-number> where the process-id assumes the role of the address space identifier. When a memory reference occurs, part of the virtual address, consisting of <process-id, page-number>, is presented to the memory subsystem. The inverted page table is then searched for a match. If a match is found say at entry i, then the physical address <i, offset> is generated. If no match is found, then an illegal address access has been attempted.

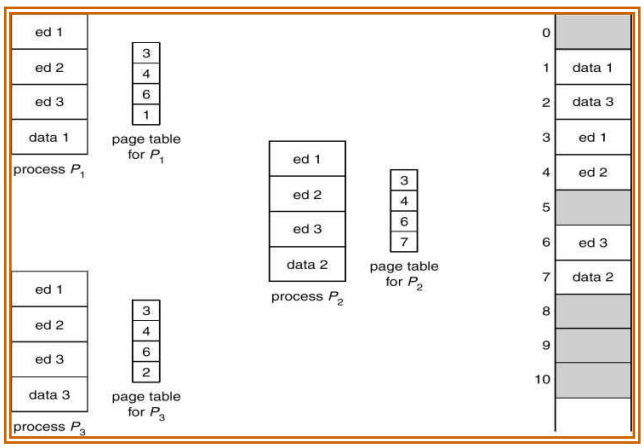
Shared Page:

- Shared code

- One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems).
- Shared code must appear in same location in the logical address space of all processes.

- Private code and data
  - Each process keeps a separate copy of the code and data.
  - The pages for the private cod**e** and data can appear anywhere in the logical address space.

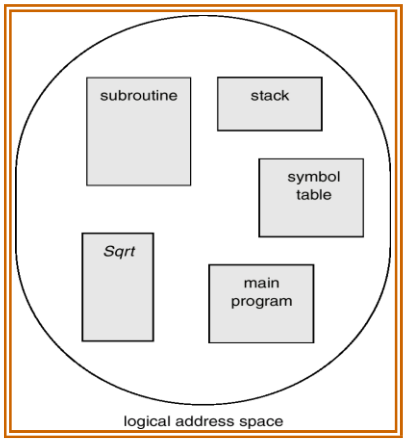
Reentrant code or pure code is non self modifying code. If the code is reentrant, then it never changes during execution. Thus, two or more processes can execute the same code at the same time. Each process has its own copy of registers and data storage to hold the data for the process' execution. The data for two different processes will of course vary for each process.



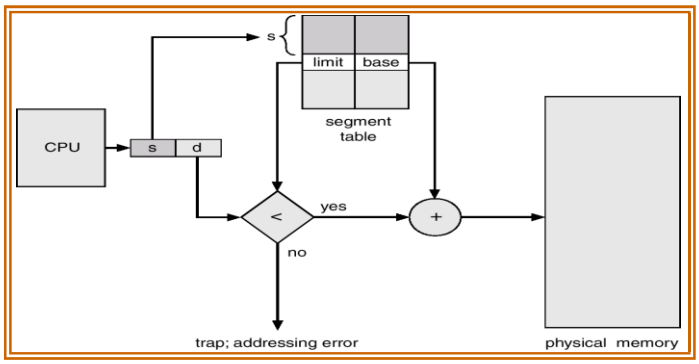
Segmentation

- Memory-management scheme that supports user view of memory.
- A program is a collection of segments. A segment is a logical unit such as:
  - Main program,
  - Procedure,
  - Function,
  - Method,
  - Object,
  - Local variables, global variables,
  - Common block,
  - Stack,

- Symbol table, arrays



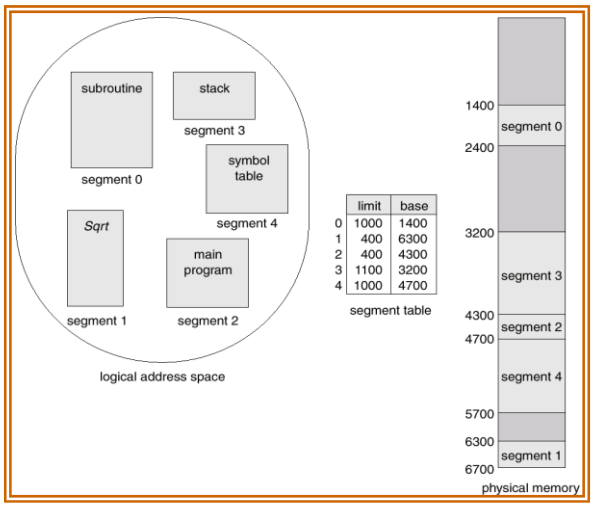
- Segmentation is a memory management scheme that supports this user view of memory.
- A logical address space is a collection of segments. Each segment has a name and a length.
- The addresses specify both the segment name and the offset within the segment.
- The user therefore specifies each address by two quantities such as segment name and an offset. For simplicity of implementation, segments are numbered and are referred to by a segment number, rather than by a segment name.
- Logical address consists of a two tuples:
    - <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.



- When the user program is compiled by the compiler it constructs the segments.
- The loader takes all the segments and assigned the segment numbers.
- The mapping between the logical and physical address using the segmentation technique is shown in above figure.
- Each entry in the segment table as limit and base address.
- The base address contains the starting physical address of a segment where the limit address specifies the length of the segment.
- The logical address consists of 2 parts such as segment number and offset.

The segment number **Segmentation with Paging**

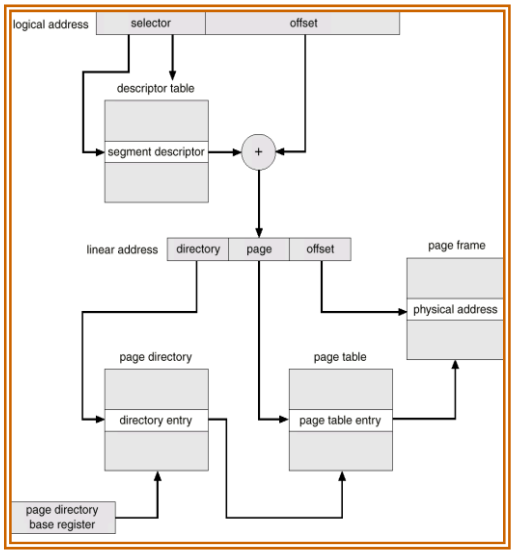
- Both paging and segmentation have advantages and disadvantages, that's why we can combine



these two methods to improve this technique for memory allocation.

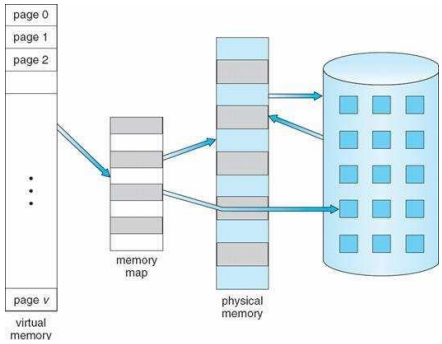
- These combinations are best illustrated by architecture of Intel-386.
- The IBM OS/2 is an operating system of the Intel-386 architecture. In this technique both segment table and page table is required.
- The program consists of various segments given by the segment table where the segment table contains different entries one for each segment.
- Then each segment is divided into a number of pages of equal size whose information is maintained in a separate page table.

- If a process has four segments that is 0 to 3 then there will be 4 page tables for that process, one for each segment.
- The size fixed in segmentation table (SMT) gives the total number of pages and therefore maximum page number in that segment with starting from 0.
- If the page table or page map table for a segment has entries for page 0 to 5.
- The address of the entry in the PMT for the desired page p in a given segment s can be obtained by B + P where B can be obtained from the entry in the segmentation table.
- Using the address (B +P) as an index in page map table (page table), the page frame (f) can be obtained and physical address can be obtained by adding offset to page frame.



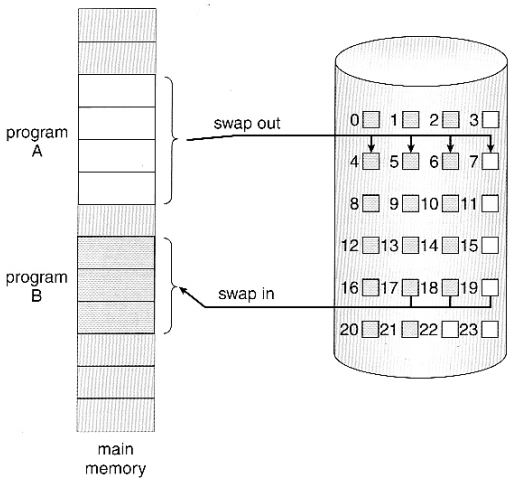
Virtual Memory

- It is a technique which allows execution of process that may not be compiled within the primary memory.
- It separates the user logical memory from the physical memory. This separation allows an extremely large memory to be provided for program when only a small physical memory is available.
- Virtual memory makes the task of programming much easier because the programmer no longer needs to working about the amount of the physical memory is available or not.
- The virtual memory allows files and memory to be shared by different processes by page sharing.
- It is most commonly implemented by demand paging.

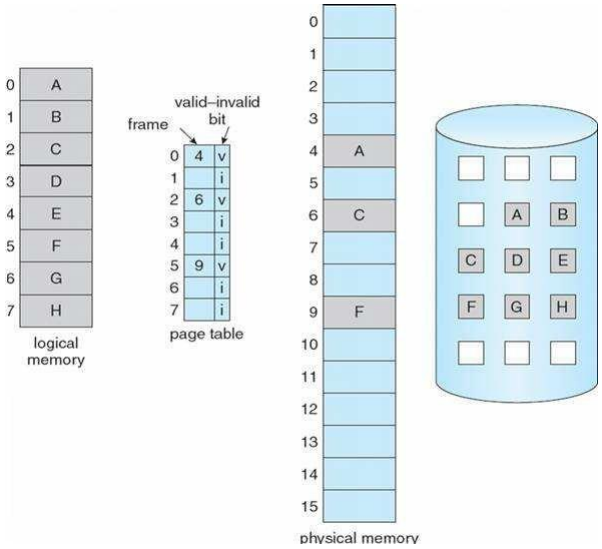


Demand Paging

A demand paging system is similar to the paging system with swapping feature. When we want to execute a process we swap it into the memory. A swapper manipulates entire process where as a pager is concerned with the individual pages of a process. The demand paging concept is using pager rather than swapper. When a process is to be swapped in, the pager guesses which pages will be used before the process is swapped out again. Instead of swapping in a whole process, the pager brings only those necessary pages into memory. The transfer of a paged memory to contiguous disk space is shown in below figure.



Thus it avoids reading into memory pages that will not used any way decreasing the swap time and the amount of physical memory needed. In this technique we need some hardware support to distinct between the pages that are in memory and those that are on the disk. A valid and invalid bit is used for this purpose. When this bit is set to valid it indicates that the associate page is in memory. If the bit is set to invalid it indicates that the page is either not valid or is valid but currently not in the disk.



Marking a page invalid will have no effect if the process never attempts to access that page. So while a process executes and access pages that are memory resident, execution proceeds normally. Access to a page marked invalid causes a page fault trap. It is the result of the OS's failure to bring the desired page into memory.

Procedure to handle page fault

If a process refers to a page that is not in physical memory then

- We check an internal table (page table) for this process to determine whether the reference was valid or invalid.
- If the reference was invalid, we terminate the process, if it was valid but not yet brought in, we have to bring that from main memory.
- Now we find a free frame in memory.
- Then we read the desired page into the newly allocated frame.
- When the disk read is complete, we modify the internal table to indicate that the page is now in memory.
- We restart the instruction that was interrupted by the illegal address trap. Now the process can access the page as if it had always been in memory.

Page Replacement

- Each process is allocated frames (memory) which hold the process's pages (data)
- Frames are filled with pages as needed – this is called demand paging

- Over-allocation of memory is prevented by modifying the page-fault service routine to replace pages
- The job of the page replacement algorithm is to decide which page gets victimized to make room for a new page
- Page replacement completes separation of logical and physical memory

Page Replacement Algorithm

Optimal algorithm

- Ideally we want to select an algorithm with the lowest page-fault rate
- Such an algorithm exists, and is called (unsurprisingly) the optimal algorithm:
- Procedure: replace the page that will not be used for the longest time (or at all) – i.e. replace the page with the greatest forward distance in the reference string
- Example using 4 frames:

Reference #	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
Page referenced	**1**	**2**	**3**	**4**	**1**	**2**	**5**	**1**	**2**	**3**	**4**	**5**
Frames	1	1	1	1	1	1	1	1	1	1	4	4
_ = faulting page		2	2	2	2	2	2	2	2	2	2	2
			3	3	3	3	3	3	3	3	3	3
				4	4	4	5	5	5	5	5	5

- Analysis: 12 page references, 6 page faults, 2 page replacements. Page faults per number of frames = 6/4 = 1.5
- Unfortunately, the optimal algorithm requires special hardware (crystal ball, magic mirror, etc.) not typically found on today's computers
- Optimal algorithm is still used as a metric for judging other page replacement algorithms

FIFO algorithm

- Replaces pages based on their order of arrival: oldest page is replaced
- Example using 4 frames:

Reference #	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
Page referenced	**1**	**2**	**3**	**4**	**1**	**2**	**5**	**1**	**2**	**3**	**4**	**5**
Frames	1	1	1	1	1	1	5	5	5	5	4	4
_ = faulting page		2	2	2	2	2	2	1	1	1	1	5
			3	3	3	3	3	3	2	2	2	2
				4	4	4	4	4	4	3	3	3

- Analysis: 12 page references, 10 page faults, 6 page replacements. Page faults per number of frames = 10/4 = 2.5

LFU algorithm (page-based)

- procedure: replace the page which has been referenced least often
- For each page in the reference string, we need to keep a reference count. All reference counts start at 0 and are incremented every time a page is referenced.
- example using 4 frames:

Reference #	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
Page referenced	**1**	**2**	**3**	**4**	**1**	**2**	**5**	**1**	**2**	**3**	**4**	**5**
Frames	^{1}1	^{1}1	^{1}1	^{1}1	^{2}1	^{2}1	^{2}1	^{3}1	^{3}1	^{3}1	^{3}1	^{3}1
_ = faulting page		^{1}2	^{1}2	^{1}2	^{1}2	^{2}2	^{2}2	^{2}2	^{3}2	^{3}2	^{3}2	^{3}2
^{n} = reference count			^{1}3	^{1}3	^{1}3	^{1}3	^{1}5	^{1}5	^{1}5	^{2}3	^{2}3	^{2}5
				^{1}4	^{1}4	^{1}4	^{1}4	^{1}4	^{1}4	^{1}4	^{2}4	^{2}4

- At the 7th page in the reference string, we need to select a page to be victimized. Either 3 or 4 will do since they have the same reference count (1). Let's pick 3.
- Likewise at the 10th page reference; pages 4 and 5 have been referenced once each. Let's pick page 4 to victimize. Page 3 is brought in, and its reference count (which was 1 before we paged it out a while ago) is updated to 2.
- Analysis: 12 page references, 7 page faults, 3 page replacements. Page faults per number of frames = 7/4 = 1.75

LFU algorithm (frame-based)

- Procedure: replace the page in the frame which has been referenced least often
- Need to keep a reference count for each frame which is initialized to 1 when the page is paged in, incremented every time the page in the frame is referenced, and reset every time the page in the frame is replaced
- Example using 4 frames:

Reference #	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
Page referenced	**1**	**2**	**3**	**4**	**1**	**2**	**5**	**1**	**2**	**3**	**4**	**5**
Frames	$^{1}\underline{1}$	$^{1}1$	$^{1}1$	$^{1}1$	$^{2}1$	$^{2}1$	$^{2}1$	$^{3}1$	$^{3}1$	$^{3}1$	$^{3}1$	$^{3}1$
_ = faulting page		$^{1}\underline{2}$	$^{1}2$	$^{1}2$	$^{1}2$	$^{2}2$	$^{2}2$	$^{2}2$	$^{3}2$	$^{3}2$	$^{3}2$	$^{3}2$
^{n} = reference count			$^{1}\underline{3}$	$^{1}3$	$^{1}3$	$^{1}3$	$^{1}\underline{5}$	$^{1}5$	$^{1}5$	$^{1}\underline{3}$	$^{1}3$	$^{1}\underline{5}$
				$^{1}\underline{4}$	$^{1}4$	$^{1}4$	$^{1}4$	$^{1}4$	$^{1}4$	$^{1}4$	$^{2}4$	$^{2}4$

- At the 7th reference, we victimize the page in the frame which has been referenced least often -- in this case, pages 3 and 4 (contained within frames 3 and 4) are candidates, each with a reference count of 1. Let's pick the page in frame 3. Page 5 is paged in and frame 3's reference count is reset to 1.
- At the 10th reference, we again have a page fault. Pages 5 and 4 (contained within frames 3 and 4) are candidates, each with a count of 1. Let's pick page 4. Page 3 is paged into frame 3, and frame 3's reference count is reset to 1.
- Analysis: 12 page references, 7 page faults, 3 page replacements. Page faults per number of frames = 7/4 = 1.75

LRU algorithm

- Replaces pages based on their most recent reference – replace the page with the greatest backward distance in the reference string
- Example using 4 frames:

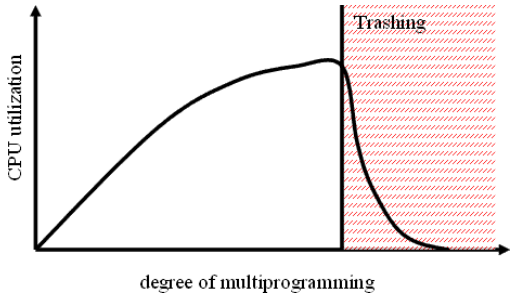
Reference #	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**

Page referenced	**1**	**2**	**3**	**4**	**1**	**2**	**5**	**1**	**2**	**3**	**4**	**5**
Frames	1	1	1	1	1	1	1	1	1	1	1	5
_ = faulting page		2	2	2	2	2	2	2	2	2	2	2
			3	3	3	3	5	5	5	5	4	4
				4	4	4	4	4	4	3	3	3

- Analysis: 12 page references, 8 page faults, 4 page replacements. Page faults per number of frames = 8/4 = 2
- One possible implementation (not necessarily the best):
    - Every frame has a time field; every time a page is referenced, copy the current time into its frame's time field
    - When a page needs to be replaced, look at the time stamps to find the oldest

Thrashing

- If a process does not have "enough" pages, the page-fault rate is very high
    - low CPU utilization
    - OS thinks it needs increased multiprogramming
    - adds another process to system
- Thrashing is when a process is busy swapping pages in and out
- Thrashing results in severe performance problems. Consider the following scenario, which is based on the actual behaviour of early paging systems. The operating system monitors CPU utilization. If CPU utilization is too low, we increase the degree of multiprogramming by introducing a new process to the system. A global page replacement algorithm is used; it replaces pages with no regard to the process to which they belong. Now suppose that a process enters a new phase in its execution and needs more frames.



FILE SYSTEM

File concept:

A file is a collection of related information that is stored on secondary storage. Information stored in files must be persistent i.e. not affected by power failures & system reboots. Files may be of free from such as text files or may be formatted rigidly. Files represent both programs as well as data. Part of the OS dealing with the files is known as file system. The important file concepts include:

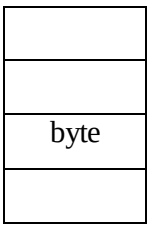
1. **File attributes:** A file has certain attributes which vary from one operating system to another.
   - **Name:** Every file has a name by which it is referred.
   - **Identifier:** It is unique number that identifies the file within the file system.
   - **Type:** This information is needed for those systems that support different types of files.
   - **Location:** It is a pointer to a device & to the location of the file on that device
   - **Size:** It is the current size of a file in bytes, words or blocks.
   - **Protection:** It is the access control information that determines who can read, write & execute a file.
   - **Time, date & user identification:** It gives information about time of creation or last modification & last use.
2. **File operations:** The operating system can provide system calls to create, read, write, reposition, delete and truncate files.
   - **Creating files:** Two steps are necessary to create a file. First, space must be found for the file in the file system. Secondly, an entry must be made in the directory for the new file.
   - **Reading a file:** Data & read from the file at the current position. The system must keep a read pointer to know the location in the file from where the next read is to take place. Once the read has been taken place, the read pointer is updated.

- **Writing a file:** Data are written to the file at the current position. The system must keep a write pointer to know the location in the file where the next write is to take place. The write pointer must be updated whenever a write occurs.
- **Repositioning within a file (seek):** The directory is searched for the appropriate entry & the current file position is set to a given value. After repositioning data can be read from or written into that position.
- **Deleting a file:** To delete a file, we search the directory for the required file. After deletion, the space is releasedso that it can be reused by other files.
- **Truncating a file:** The user may erase the contents of a file but allows all attributes to remain unchanged expect the file length which is rest to 'O' & the space is released.

3. **File types:** The file name is spilt into 2 parts, Name & extension. Usually these two parts are separated by a period. The user & the OS can know the type of the file from the extension itself. Listed below are some file types along with their extension:

File Type	Extension
Executable File	exe, bin, com
Object File	obj, o (compiled)
Source Code file	C, C++, Java, pas
Batch File	bat, sh (commands to command the interpreter)
Text File	txt, doc (textual data documents)
Archieve File	arc, zip, tar (related files grouped together into file compressed for storage)
Multimedia File	mpeg (Binary file containing audio or A/V information)

4. **File structure:** Files can be structured in several ways. Three common possible are:
   - **Byte sequence:**The figure shows an unstructured sequence of bytes. The OS doesn't care about the content of file. It only sees the bytes. This structure provides maximum flexibility. Users can write anything into their files & name them according to their convenience. Both UNIX & windows use this approach.



- **Record sequence:** In this structure, a file is a sequence of fixed length records. Here the read operation returns one records & the write operation overwrites or append or record.

- **Tree:**In this organization, a file consists of a tree of records of varying lengths. Each record consists of a key field. The tree is stored on the key field to allow first searching for a particular key.

Access methods: Basically, access method is divided into 2 types:

- **Sequential access:** It is the simplest access method. Information in the file is processed in order i.e. one record after another. A process can read all the data in a file in order starting from beginning but can't skip & read arbitrarily from any location. Sequential files can be rewound. It is convenient when storage medium was magnetic tape rather than disk.
- **Direct access:** A file is made up of fixed length-logical records that allow programs to read & write records rapidly in no particular O order. This method can be used when disk are used for storing files. This method is used in many applications e.g. database systems. If an airline customer wants to reserve a seat on a particular flight, the reservation program must be able to access the record for that flight directly without reading the records before it. In a direct access file, there is no restriction in the order of reading or writing. For example, we can read block 14, then read block 50 & then write block 7 etc. Direct access files are very useful for immediate access to large amount of information.

Directory structure: The file system of computers can be extensive. Some systems store thousands of file on disk. To manage all these data, we need to organize them. The organization is done in 2 steps. The file system is broken into partitions. Each partition contains information about file within it.

Operation on a directory:

- **Search for a file:** We need to be able to search a directory for a particular file.
- **Create a file:** New files are created & added to the directory.
- **Delete a file:** When a file is no longer needed, we may remove it from the directory.
- **List a directory:** We should be able to list the files of the directory.
- **Rename a file:** The name of a file is changed when the contents of the file changes.
- **Traverse the file system:** It is useful to be able to access every directory & every file within a directory.

Structure of a directory: The most common schemes for defining the structure of the directory are:

1. **Single level directory:** It is the simplest directory structure. All files are present in the same directory. So it is easy to manage & understand.

   Limitation: A single level directory is difficult to manage when the no. of files increases or when there is more than one user. Since all files are in same directory, they must have unique names. So, there is confusion of file names between different users.
2. **Two level directories:** The solution to the name collision problem in single level directory is to create a separate directory for each user. In a two level directory structure, each user has its own user file directory. When a user logs in, then master file directory is searched. It is indexed by user name & each entry points to the UFD of that user.

   Limitation: It solves name collision problem. But it isolates one user from another. It is an advantage when users are completely independent. But it is a disadvantage when the users need to access each other's files & co-operate among themselves on a particular task.
3. **Tree structured directories:** It is the most common directory structure. A two level directory is a two level tree. So, the generalization is to extend the directory structure to a tree of arbitrary height. It allows users to create their own subdirectories & organize their files. Every file in the system has a unique path name. It is the path from the root through all the sub-directories to a specified file. A directory is simply another file but it is treated in a special way. One bit in each

directory entry defines the entry as a file (O) or as sub- directories. Each user has a current directory. It contains most of the files that are of current interest to the user. Path names can be of two types: An absolute path name begins from the root directory & follows the path down to the specified files. A relative path name defines the path from the current directory. E.g. If the current directory is root/spell/mail, then the relative path name is prt/first & the absolute path name is root/ spell/ mail/ prt/ first. Here users can access the files of other users also by specifying their path names.

4. **A cyclic graph directory:**It is a generalization of tree structured directory scheme. An a cyclic graph allows directories to have shared sub-directories & files. A shared directory or file is not the same as two copies of a file. Here a programmer can view the copy but the changes made in the file by one programmer are not reflected in the other's copy. But in a shared file, there is only one actual file. So many changes made by a person would be immediately visible to others. This scheme is useful in a situation where several people are working as a team. So, here all the files that are to be shared are put together in one directory. Shared files and sub-directories can be implemented in several ways. A common way used in UNIX systems is to create a new directory entry called link. It is a pointer to another file or sub-directory. The other approach is to duplicate all information in both sharing directories. A cyclic graph structure is more flexible then a tree structure but it is also more complex.

   Limitation: Now a file may have multiple absolute path names. So, distinct file names may refer to the same file. Another problem occurs during deletion of a shared file. When a file is removed by any one user. It may leave dangling pointer to the non existing file. One serious problem in a cyclic graph structure is ensuring that there are no cycles. To avoid these problems, some systems do not allow shared directories or files. E.g. MS-DOS uses a tree structure rather than a cyclic to avoid the problems associated with deletion. One approach for deletion is to preserve the file until all references to it are deleted. To implement this approach, we must have some mechanism for determining the last reference to the file. For this we have to keep a list of reference to a file. But due to the large size of the no. of references. When the count is zero, the file can be deleted.

5. **General graph directory:** When links are added to an existing tree structured directory, the tree structure is destroyed, resulting in a simple graph structure. Linking is a technique that allows a file to appear in more than one directory. The advantage is the simplicity of algorithm to transverse the graph & determines when there are no more references to a file. But a similar

problem exists when we are trying to determine when a file can be deleted. Here also a value zero in the reference count means that there are no more references to the file or directory & the file can be deleted. But when cycle exists, the reference count may be non-zero even when there are no references to the directory or file. This occurs due to the possibility of self referencing (cycle) in the structure. So, here we have to use garbage collection scheme to determine when the last references to a file has been deleted & the space can be reallocated. It involves two steps:

- Transverse the entire file system & mark everything that can be accessed.
- Everything that isn't marked is added to the list of free space.

But this process is extremely time consuming. It is only necessary due to presence of cycles in the graph. So, a cyclic graph structure is easier to work than this.

Protection

When information is kept in a computer system, a major concern is its protection from physical damage (reliability) as well as improper access.

Types of access: In case of systems that don't permit access to the files of other users. Protection is not needed. So, one extreme is to provide protection by prohibiting access. The other extreme is to provide free access with no protection. Both these approaches are too extreme for general use. So, we need controlled access. It is provided by limiting the types of file access. Access is permitted depending on several factors. One major factor is type of access requested. The different type of operations that can be controlled are:

- **Read**
- **Write**
- **Execute**
- **Append**
- **Delete**
- **List**

Access lists and groups:

Various users may need different types of access to a file or directory. So, we can associate an access lists with each file and directory to implement identity dependent access. When a user access requests access to a particular file, the OS checks the access list associated with that file. If that user is granted the requested access, then the access is allowed. Otherwise, a protection violation occurs & the user is denied access to the file. But the main problem with access lists is their length. It is

very tedious to construct such a list. So, we use a condensed version of the access list by classifying the users into 3 categories:

- **Owners:** The user who created the file.
- **Group:** A set of users who are sharing the files.
- **Others:** All other users in the system.

Here only 3 fields are required to define protection. Each field is a collection of bits each of which either allows or prevents the access. E.g. The UNIX file system defines 3 fields of 3 bits each: rwx

- r(read access)
- w(write access)
- x(execute access)

Separate fields are kept for file owners, group & other users. So, a bit is needed to record protection information for each file.

Allocation methods

There are 3 methods of allocating disk space widely used.

1. **Contiguous allocation:**
   a. It requires each file to occupy a set of contiguous blocks on the disk.
   b. Number of disk seeks required for accessing contiguously allocated file is minimum.
   c. The IBM VM/CMS OS uses contiguous allocation. Contiguous allocation of a file is defined by the disk address and length (in terms of block units).
   d. If the file is 'n' blocks long and starts all location 'b', then it occupies blocks b, b+1, b+2,---- - -b+ n-1.
   e. The directory for each file indicates the address of the starting block and the length of the area allocated for each file.
   f. Contiguous allocation supports both sequential and direct access. For sequential access, the file system remembers the disk address of the last block referenced and reads the next block when necessary.
   g. For direct access to block i of a file that starts at block b we can immediately access block b + i.
   h. **Problems:** One difficulty with contiguous allocation is finding space for a new file. It also suffers from the problem of external fragmentation. As files are deleted and allocated, the free disk space is broken into small pieces. A major problem in contiguous allocation is how

much space is needed for a file. When a file is created, the total amount of space it will need must be found and allocated. Even if the total amount of space needed for a file is known in advances, pre-allocation is inefficient. Because a file that grows very slowly must be allocated enough space for its final size even though most of that space is left unused for a long period time. Therefore, the file has a large amount of internal fragmentation.

2. **Linked Allocation:**
    a. Linked allocation solves all problems of contiguous allocation.
    b. In linked allocation, each file is linked list of disk blocks, which are scattered throughout the disk.
    c. The directory contains a pointer to the first and last blocks of the file.
    d. Each block contains a pointer to the next block.
    e. These pointers are not accessible to the user. To create a new file, we simply create a new entry in the directory.
    f. For writing to the file, a free block is found by the free space management system and this new block is written to & linked to the end of the file.
    g. To read a file, we read blocks by following the pointers from block to block.
    h. There is no external fragmentation with linked allocation & any free block can be used to satisfy a request.
    i. Also there is no need to declare the size of a file when that file is created. A file can continue to grow as long as there are free blocks.
    j. **Limitations:** It can be used effectively only for sequential access files. To find the ‘ i ' th block of the file, we must start at the beginning of that file and follow the pointers until we get the ith block. So it is inefficient to support direct access files. Due to the presence of pointers each file requires slightly more space than before. Another problem is reliability. Since the files are linked together by pointers scattered throughout the disk. What would happen if a pointer were lost or damaged.
3. **Indexed Allocation:**
    a. Indexed allocation solves the problem of linked allocation by bringing all the pointers together to one location known as the index block.
    b. 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.

c. The directory contains the address of the index block. The read the ith block, we use the pointer in the ith index block entry and read the desired block.
d. To write into the ith block, a free block is obtained from the free space manager and its address is put in the ith index block entry.
e. Indexed allocation supports direct access without suffering external fragmentation.
f. **Limitations:** The pointer overhead of index block is greater than the pointer overhead of linked allocation. So here more space is wasted than linked allocation. In indexed allocation, an entire index block must be allocated, even if most of the pointers are nil.

Free Space Management

Since there is only a limited amount of disk space, it is necessary to reuse the space from the deleted files. To keep track of free disk space, the system maintains a free space list. It records all the disk blocks that are free i.e. not allocated to some file or dictionary. To create a file, we search the free space list for the required amount of space and allocate it 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.

Implementation:

There are 4 ways to implement the free space list such as:

- **Bit Vector:** The free space list is implemented as a bit map or bit vector. Each block is represented as 1 bit. If the block is free, the bit is 1 and if it is allocated then the bit is 0. For example, consider a disk where blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26 & 27 are free and rest of the blocks are allocated. The free space bit map would be 00111100111111000110000001111……..

  The main advantage of this approach is that it is simple and efficient to find the first free block or n consecutive free blocks on the disk. But bit vectors are inefficient unless the entire vector is kept in main memory. It is possible for smaller disks but not for larger ones.
- **Linked List:** Another approach is to link together all the free disk blocks and keep a pointer to the first free block. The first free block contains a pointer to the next free block and so on. For example, we keep a pointer to block 2 as the free block. Block 2 contains a pointer to block which points to block 4 which then points to block 5 and so on. But this scheme is not efficient. To traverse the list, we must read each block which require a lot of I/O time.

- **Grouping:** In this approach, we store the address of n free blocks in the first free block. The first n-1 of these blocks is actually free. The last block contains the address of another n free blocks and so on. Here the addresses of a large number of free blocks can be found out quickly.
- **Counting:** Rather than keeping a list of n free disk block addresses, we can keep the address of the first free block and the number of free contiguous blocks. So here each entry in the free space list consists of a disk address and a count.