Unit – IV
MEMORY MANAGEMENT STRATEGIES

STORAGE MANAGEMENT

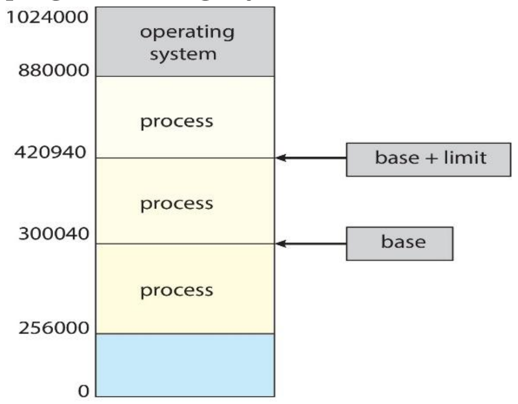
MEMORY MANAGEMENT STRATEGIES

1. Introduction

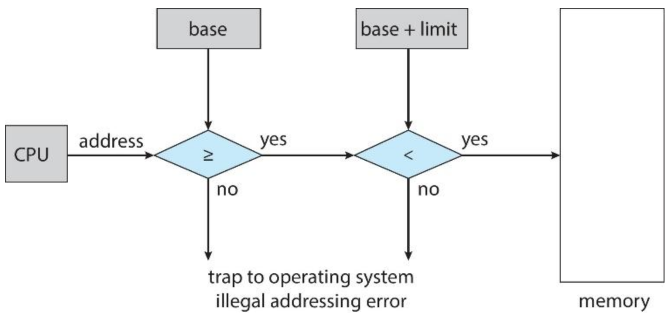
- A typical instruction-execution cycle, for example, first fetches an instruction from memory. The instruction is then decoded and may cause operands to be fetched from memory. After the instruction has been executed on the operands, results may be stored back in memory.
- The memory unit sees only a stream of memory addresses; it does not know how they are generated (by the instruction counter, indexing, indirection, literal addresses, and so on) or what they are for (instructions or data). Accordingly, we can ignore how a program generates a memory address.

1.1. Basic Hardware

- For proper system operation we must protect the operating system from access by user processes.
- We first need to make sure that each process has a separate memory space. To separate memory spaces, we need the ability to determine the range of legal addresses that the process may access and to ensure that the process can access only these legal addresses.
- We can provide this protection by using two registers, usually a base and 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 the limit register is 120900, then the program can legally access all addresses from 300040 through 420939.



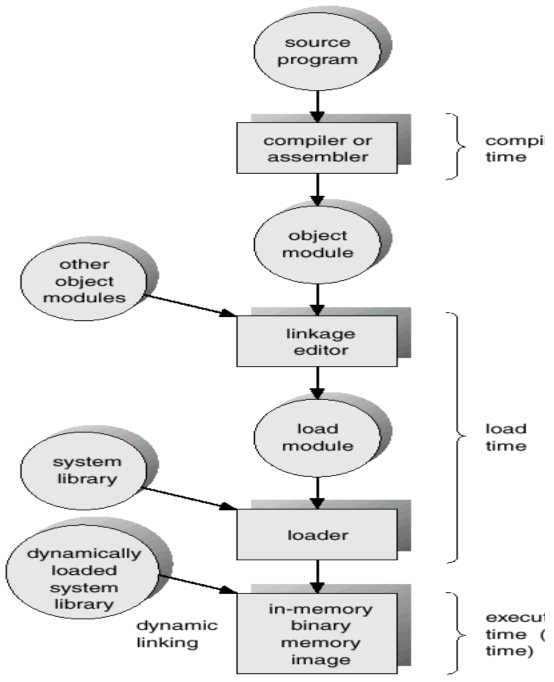
- CPU must check every memory access generated in user mode to be sure it is between base and limit for that user



- Any attempt by a program executing in user mode to access operating-system memory or other users memory results in a trap to the operating system, which treats the attempt as a fatal error.
- The base and limit registers can be loaded only by the operating system, which uses a special privileged instruction.

1.2. Address Binding

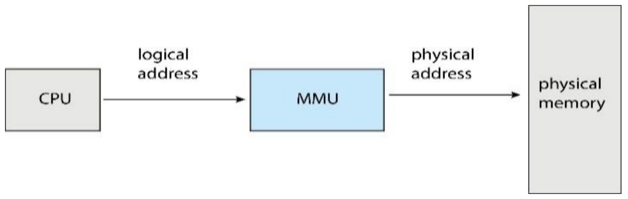
- In most cases, a user program goes through several steps—some of which may be optional—before being executed.
- Addresses may be represented in different ways during these steps. Addresses in the source program are generally symbolic (such as the variable count).
- A compiler typically binds these symbolic addresses to relocatable addresses (such as "14 bytes from the beginning of this module").
- The linker or loader in turn binds the relocatable addresses to absolute addresses (such as 74014). Each binding is a mapping from one address space to another.
- 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).

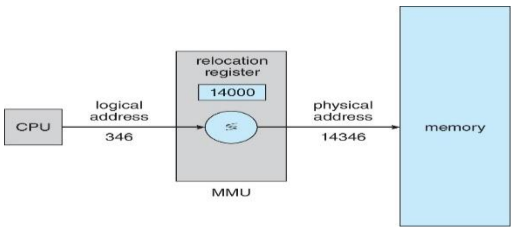
1.3. Logical versus Physical Address Space

- An address generated by the CPU is commonly referred to as a ***logical address***, whereas an address seen by the memory unit—that is, the one loaded into the memory-address register of the memory—is commonly referred to as a ***physical address.***
- 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
- 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



- The run-time mapping from virtual to physical addresses is done by a hardware device called the memory-management unit (MMU).
- We can choose from many different methods to accomplish such mapping.

- Consider simple scheme, which is a generalization of the base-register scheme.
- The 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 it is sent to memory.
- For example, if the base is at 14000, then an attempt an access to location 346 is mapped to location 14346.
- The user program never accesses the real physical addresses. The program can create a pointer to location 346, store it in memory, manipulate it, and compare it with other addresses—all as the number 346. Only when it is used as a memory address is it relocated relative to the base register.



- 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
- The final location of a referenced memory address is not determined until the reference is made.

1.4. Dynamic Loading

- The entire program does not need to be in memory to execute.
- With dynamic loading, a 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 routine has been loaded. If it has 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.
- Advantage of dynamic loading
  - A routine is loaded only when it is needed. This method is particularly useful when large amounts of code are needed to handle infrequently occurring cases.
  - It does not require special support from the operating system. It is the responsibility of the users to design their programs to take advantage of such a method.

1.5. Dynamic Linking and Shared Libraries

- Dynamically linked libraries (DLLs) are system libraries that are linked to user programs when the programs are run.
- **Static linking** – system libraries and program code combined by the loader into the binary program image
- **Dynamic linking** –linking postponed until execution time.
- This feature is usually used with system libraries, such as the standard C language library. Without this facility, each program on a system must include a copy of its language library in the executable image. This requirement not only increases the size of an executable image but also may waste main memory.
- A second advantage of DLLs is that these libraries can be shared among multiple processes, so that only one instance of the DLL in main memory.
- For this reason, DLLs are also known as shared libraries, and are used extensively in

Windows and Linux systems.

2. Swapping

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

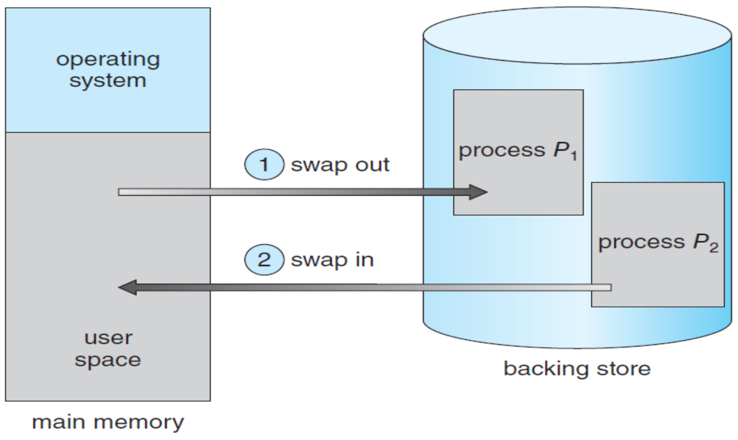


Figure 9.19 Standard swapping of two processes using a disk as a backing store.

2.1. Standard Swapping

- Standard swapping involves moving processes between main memory and a backing store.
- **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.
- **Roll out, roll in** – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed.
- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped and Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)
- System maintains a ready queue of ready-to-run processes which have memory images on disk

2.2. Swapping on Mobile Systems

- Mobile systems typically do not support swapping in any form.
- Mobile devices generally use flash memory rather than more spacious hard disks as their persistent storage. The resulting space constraint is one reason why mobile operating-system designers avoid swapping.
- Instead of using swapping, when free memory falls below a certain threshold, Apple's iOS asks applications to voluntarily relinquish allocated memory.
- Android does not support swapping and adopts a strategy similar to that used by iOS. It may terminate a process if insufficient free memory is available.
- Because of these restrictions, developers for mobile systems must carefully allocate and release memory to ensure that their applications do not use too much memory or suffer

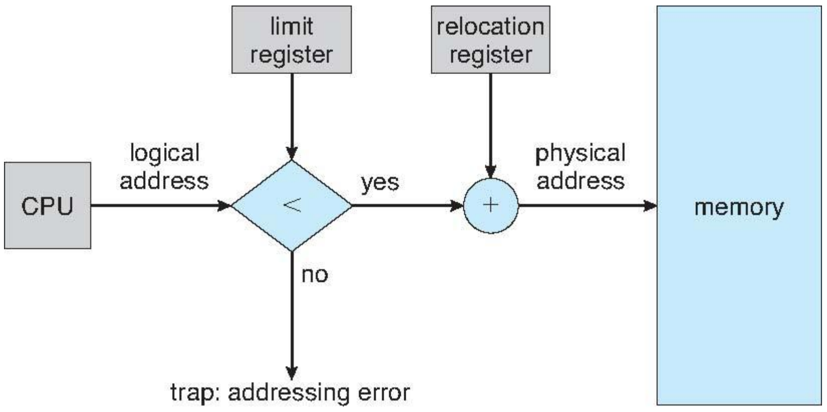
from memory leaks.

3. Contiguous memory allocation

- Main memory must accommodate both the operating system and the various user processes.
- In Contiguous memory allocation, main memory is usually divided into two partitions
  1) one for the resident operating system usually held in low memory addresses.
  2) one for the user processes usually held in high memory addresses.
- In contiguous memory allocation, each process is contained in a single section of memory that is contiguous to the section containing the next process.

3.1. Memory Mapping and Protection

- We can prevent a process from accessing memory that it does not own. If we have a system with a relocation register together with a limit register we accomplish our goal.
- The relocation register contains the value of the smallest physical address; the limit register contains the range of logical addresses (for ex: relocation = 100040 and limit = 74600).
- Each logical address must fall within the range specified by 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.

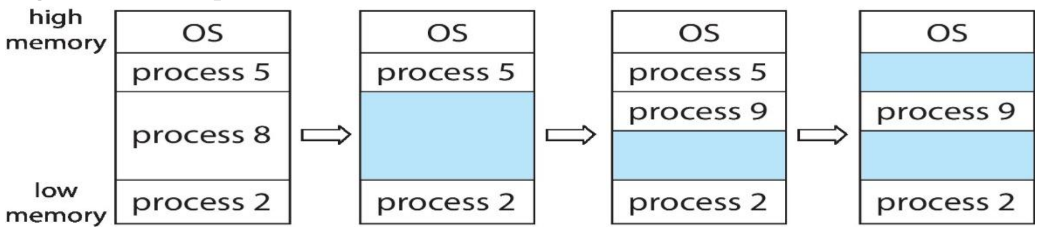


- When the CPU scheduler selects a process for execution, the dispatcher loads the relocation and limit registers with the correct values as part of the context switch.
- Because every address generated by a CPU is checked against these registers, we can protect both the operating system and the other users' programs and data from being modified by this running process.
- The relocation-register scheme provides an effective way to allow the operating system's size to change dynamically.
- For example, the operating system contains code and buffer space for device drivers. If a device driver is not currently in use, it makes little sense to keep it in memory; instead, it can be loaded into memory only when it is needed. Likewise, when the device driver is no longer needed, it can be removed and its memory allocated for other needs.

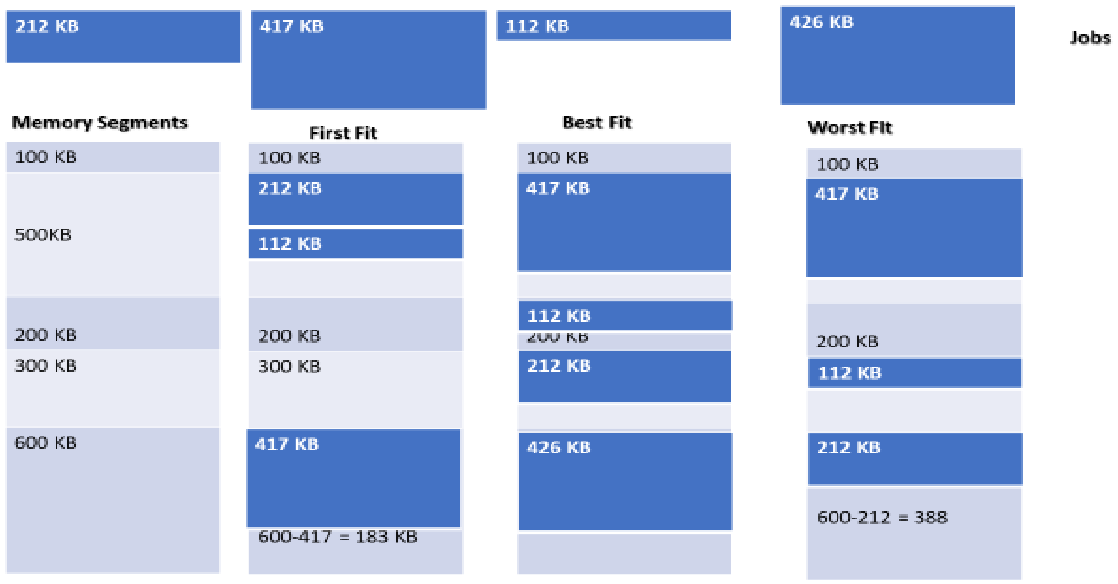
3.2. Memory Allocation

- One of the simplest methods of allocating memory is to assign processes to variably sized partitions in memory, where each partition may contain exactly one process.
- In this variable partition scheme, the operating system keeps a table indicating which parts of memory are available and which are occupied.
- Initially, all memory is available for user processes and is considered one large block of available memory, a hole. Eventually, as you will see, memory contains a set of holes of various sizes

- Figure below depicts this scheme.



- Initially, the memory is fully utilized, containing processes 5, 8, and 2. After process 8 leaves, there is one contiguous hole. Later on, process 9 arrives and is allocated memory. Then process 5 departs, resulting in two noncontiguous holes.
- When a process is allocated space, it is loaded into memory, where it can then compete for CPU time. When a process terminates, it releases its memory, which the operating system may then provide to another process.
- The memory blocks available comprise a set of holes of various sizes scattered throughout memory. When a process arrives and needs memory, the system searches the set for a hole that is large enough for this process.
- If the hole is too large, it is split into two parts. 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 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 n from list of free holes. There are many solutions to this problem.
- **First fit:** Allocate the first hole that is big 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.
- First-fit and best-fit better than worst-fit in terms of speed and storage utilization

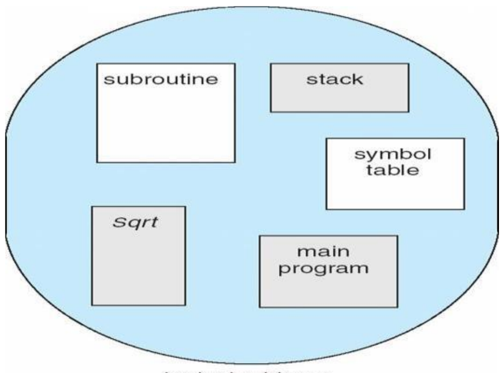


3.3. Fragmentation

- Both the first-fit and best-fit strategies for memory allocation suffer from external fragmentation. As processes are loaded and removed from memory, the free memory space is broken into little pieces.
- **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.
- This fragmentation problem can be severe. In the worst case, we could have a block of free (or wasted) memory between every two processes. If all these small pieces of memory were in one big free block instead, we might be able to run several more processes.
- Depending on the total amount of memory storage and the average process size, external fragmentation may be a minor or a major problem.
- Statistical analysis of first fit, for instance, reveals that, even with some optimization, given N allocated blocks, another 0.5 N blocks will be lost to fragmentation. That is, one-third of memory may be unusable! This property is known as the 50-percent rule.
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
- 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.
- Compaction is possible only if relocation is dynamic and is done at execution time. If addresses are relocated dynamically, relocation requires only moving the program and data and then changing the base register to reflect the new base address.
- When compaction is possible, we must determine its cost. 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 processes to be noncontiguous, thus allowing a process to be allocated physical memory wherever such memory is available.
- This is the strategy used in paging, the most common memory-management

4. Segmentation

- Most programmers view memory as a collection of variable-sized segments, with no necessary ordering among segments.
- When writing a program, a programmer thinks of it as a main program with a set of methods, procedures, or functions. It may also include various data structures: objects, arrays, stacks, variables and so on.



logical address

- Segmentation is a memory-management scheme that supports programmer view of memory.

4.1. Basic Method

- A logical address space is a collection of segments. Each segment has a name and a length.
- A logical address consists of a *two tuple*:

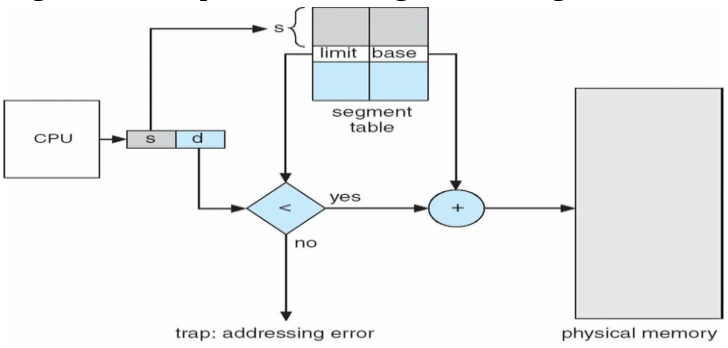
<segment-number,offset>

Normally, when a program is compiled, the compiler automatically constructs segments reflecting the input program.

- A C compiler might create a separate segments for the following:
  1. The code
  2. Global variables
  3. The head from which memory is allocated
  4. The stacks used by thread
  5. The standard C library

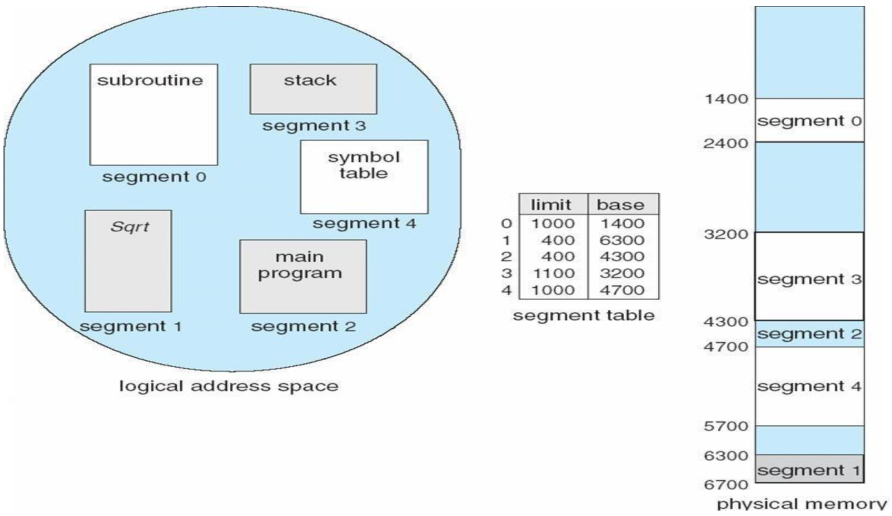
4.2. Hardware

- A ***segment table*** maps two-dimensional programmer-defined addresses into one-dimensional physical addresses.
- Each entry in the segment table has a ***segment base*** and a ***segment limit.*** The segment base contains the starting physical address where the segment resides in memory, and the segment limit specifies the length of the segment.



- The use of a segment table is illustrated in Figure above.
- The segment number ***s*** 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 operating system.
- When an offset is legal, it is added to the segment base to produce the address in physical memory of the desired byte.

Example of segmentation



	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

- Consider the situation shown in Figure above. We have five segments numbered from 0 through 4. The segments are stored in physical memory as shown.

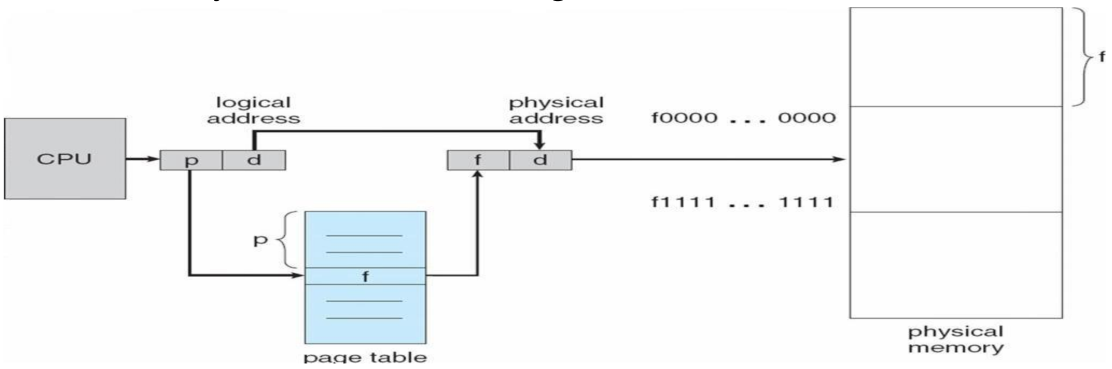
- The segment has a separate entry for each segment, giving the beginning address of the segment in physical memory and the length of that segment.
- For example, segment 2 is 400 bytes long and begins at location 4300. Thus a reference, of byte 53 of segment 2 is mapped onto location 4300+53=4533.
- A reference of segment 3, byte 852, is mapped to 3200+852=4052.

5. Paging

- A computer can address more memory than the amount physically installed on the system. This extra memory is actually called virtual memory and it is a section of a hard that's set up to emulate the computer's RAM.

5.1. Basic Method

- Paging involves breaking physical memory into fixed-size bocks called ***frames*** and breaking logical memory into blocks of the same size called ***pages.***
- Logical memory is broken into blocks of the fixed size called ***pages*** (size is power of 2, between 512 bytes and 8192 bytes).
- Main memory is divided into small fixed-sized blocks of (physical) memory called frames and the size of a frame is kept the same as that of a page to have optimum utilization of the main memory and to avoid external fragmentation.

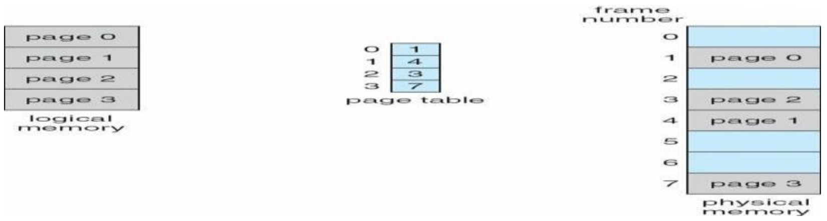


- The hardware support for paging is illustrated in Figure above.

Address Translation

- Page address is called ***logical address*** and represented by ***page number*** and the ***offset***.
  Logical address = Page number + Page offset
- Frame address is called ***physical address*** and represented by a ***frame number*** and the ***offset***.
  Physical address = Frame number + Page offset
- A data structure called ***page map table*** is used to keep track of the relation between a page of a process to a frame in physical memory.

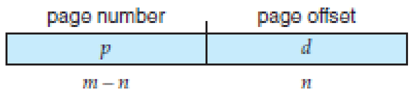
Paging model of logical and physical memory



- The page size is defined by the hardware. The size of a page is a power of 2, varying between 512 bytes and 1 GB 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

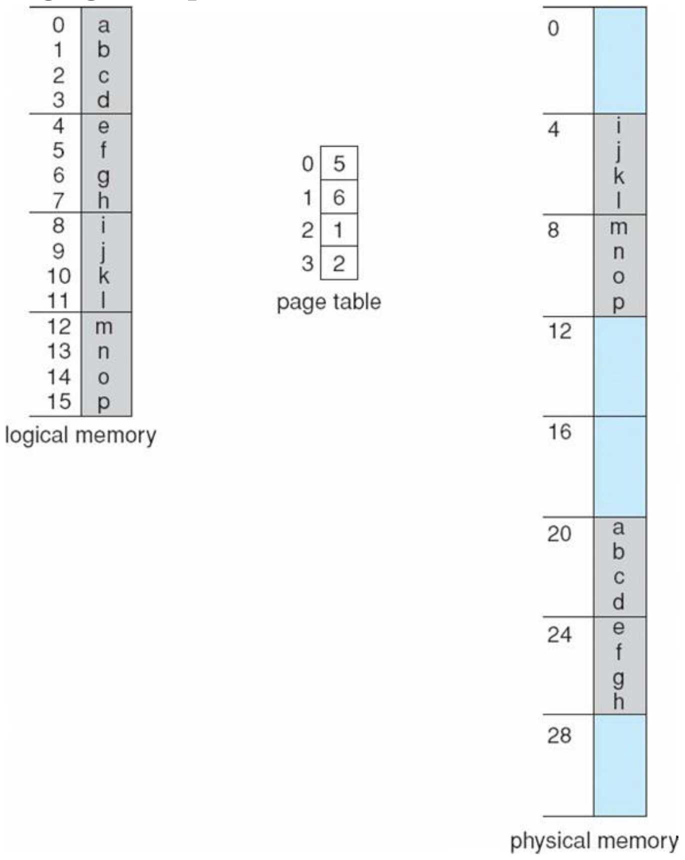
a page number and page offset.

- If the size of the logical address space is 2^m, and a page size is 2^n bytes, 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. Thus, the logical address is as follows:



Where ***p*** is an index into the page table and ***d*** is the displacement within the page

Paging Example



- Consider the memory in Figure above. Here, in the logical address, n=2 and m=4.
- Using a page size of 4 bytes and a physical memory of 32 bytes (8pages), we show how the programmer's view of memory can mapped into physical memory.
- Logical address 0 is page 0, offset 0. Indexing into the page table, we find that page 0 is in frame 5. Thus, logical address 0 maps to physical address 20 [= (5x4) + 0].
- Logical address 3 maps to physical address 23 [= (5x4) + 3].
- Logical address 4 is page 1, offset 0; according to the page table, page 1 is mapped to frame 6. Thus, logical address 4 maps to physical address 24 [= (6x4) + 0].
- Logical address 13 maps to physical address 9.

Paging – Calculating internal Fragmentation

Page size = 2,048 bytes
Process size = 72,766 bytes
Pages: Process size / Page size= 72,766 bytes / 2,048 bytes= 35.5 pages
35 pages + 1,086 bytes
Page Table Size: 32-bit address –
Number of pages * Page table entry size= 36 pages * 4 bytes= 144 bytes
Internal fragmentation of 2,048 - 1,086 = 962 bytes

Worst case fragmentation = 1 frame – 1 byte

On average fragmentation = 1 / 2 frame size

Free Frames

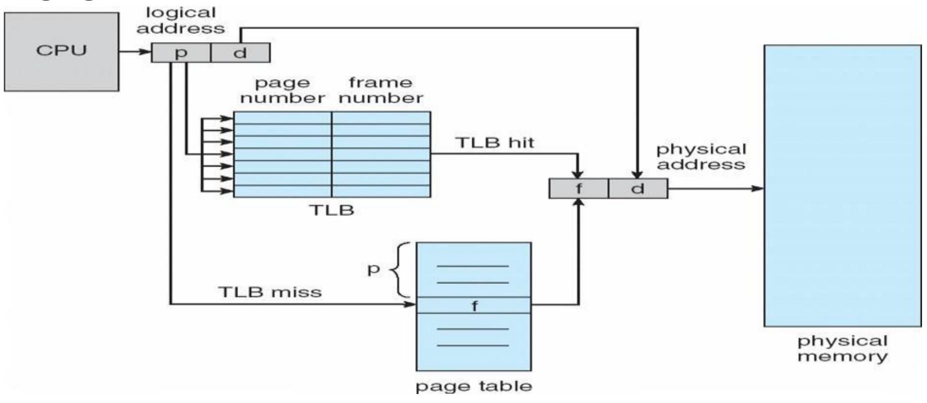
- When the system allocates a frame to any page, it translates this logical address into a physical address and create entry into the page table to be used throughout execution of the program.
- When a process is to be executed, its corresponding pages are loaded into any available memory frames. Suppose you have a program of 8Kb but your memory can accommodate only 5Kb at a given point in time, then the paging concept will come into picture.
- When a computer runs out of RAM, the operating system will move idle or unwanted pages of memory to secondary memory to free up RAM for other processes and brings them back when needed by the program.
- This process continues during the whole execution of the program where the OS keeps removing idle pages from the main memory and write them onto the secondary memory and bring them back when required by the program.

5.2. Hardware Support

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 (PRLR) 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.
- The 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)

Paging Hardware With TLB



- Memory structures for paging can get massive using straight-forward methods

  Page Size: 4 KB

  Process size: 1094 MB

  Address is 32 bit

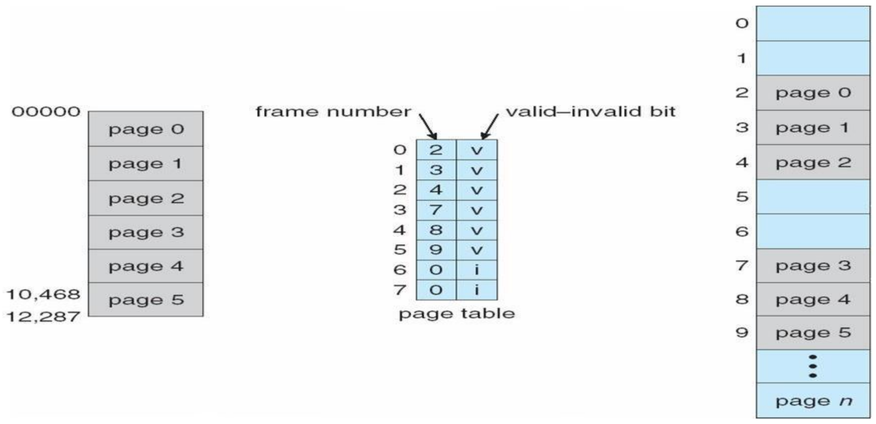
  Number of pages = Process size / Page size= 1,147,483,648 bytes / 4,096 bytes (since 4 KB = 4,096 bytes)= 280,230 pages

  Page table size = Number of pages * Page table entry size= 280,230 pages * 4 bytes= 1,120,920 bytes= 1.07 MB (approximately)

5.3. Protection

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



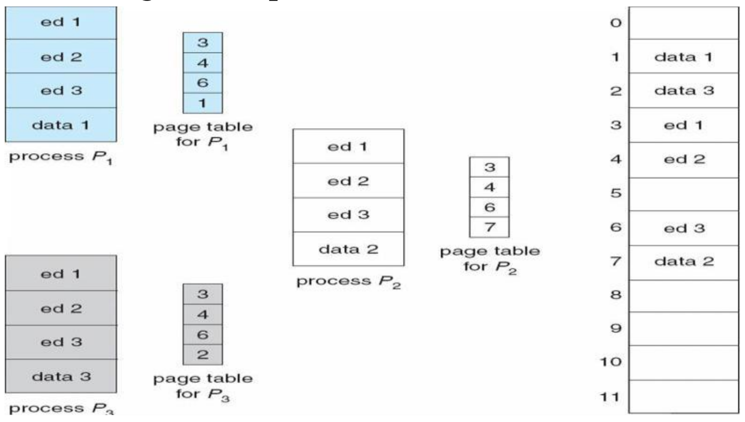
5.4. Shared Pages

- An advantage of paging is the possibility of ***sharing*** common code. This consideration is particularly important in a time-sharing environment.

  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 code and data can appear anywhere in the logical address space.

Shared Pages Example



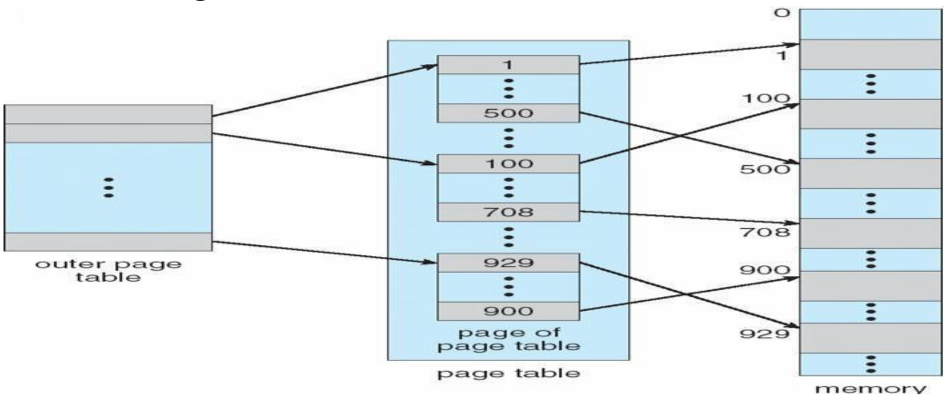
- Consider a system that supports 40 users, each of whom executes a text editor. If the text editor consists of 150 KB of code and 50 KB of data space, we need 8000 KB to support the 40 users.
- If the code is **reentrant code (or pure code),** however, it can be shared, as shown in figure below. Here, we see three processes sharing a three-page editor – each page 50 KB in size. Each process has its own data page.
- Reentrant 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.

6. Structure of the Page Table

6.1. Hierarchical Paging

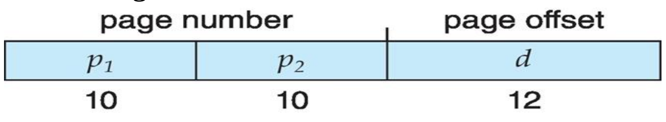
- Break up the logical address space into multiple page tables A simple technique is a two-level page table.

Two-Level Page-Table Scheme



Two-Level Paging Example

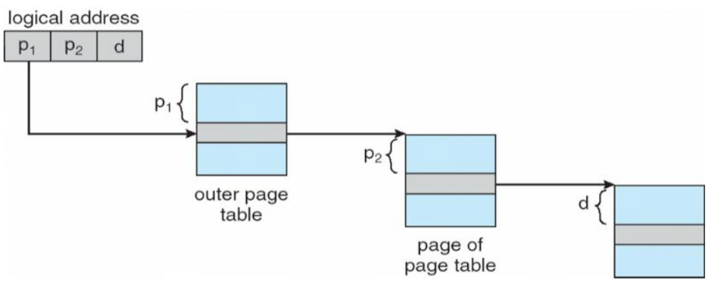
- A logical address (on 32-bit machine with 4K page size) is divided into:
  - a page number consisting of 20 bits
  - a page offset consisting of 10 bits
- Since the page table is paged, the page number is further divided into:
  - a 10-bit page number
  - a 10-bit page offset
- Thus, a logical address is as follows:



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

- This scheme is known as a **forward-mapped page table.**

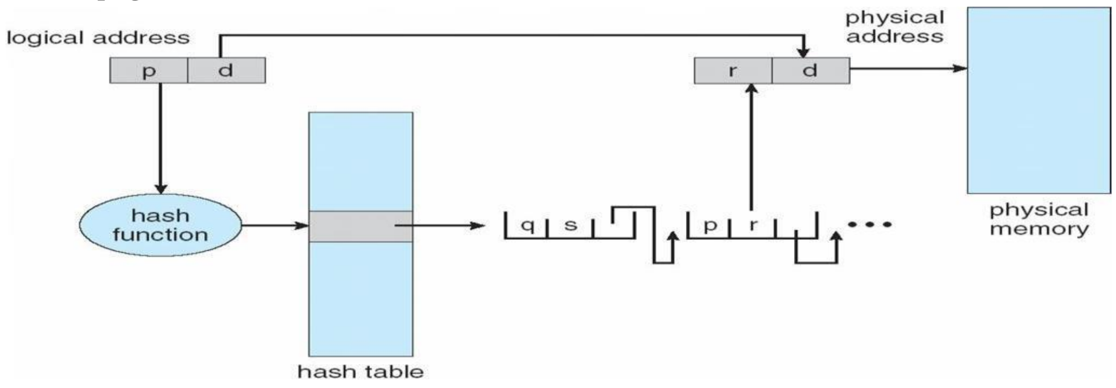
Address Translation Scheme



6.2. Hashed Page Tables

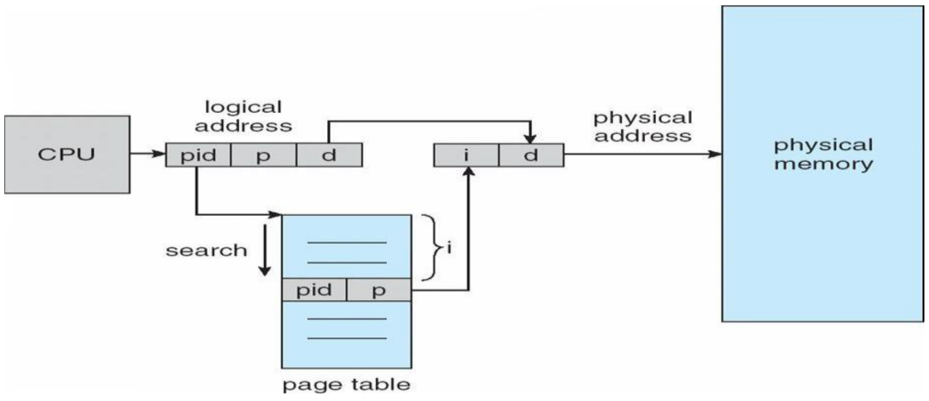
- A common approach for handling address spaces larger than 32 bits is to use a ***hashed page table***, with the hash value being the virtual page number.
- Each entry in the hash table contains a linked list of elements that hash to the same location.
- Each element consists of three fields:
  1. The virtual page number
  2. The value of the mapped page frame, and
  3. 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 with field 1 in the first element in the linked list.
- If there is a match, the corresponding page frame (field 2) 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.



6.3. Inverted Page Tables

- An inverted page table has 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
- Use hash table to limit the search to one — or at most a few — page-table entries

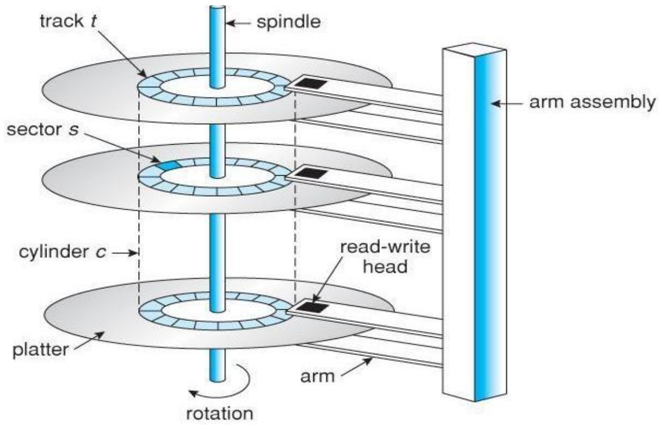


STORAGE MANAGEMENT

1. Overview of Mass Storage Structure

1.1. Magnetic Disks

- Magnetic disks provide the bulk of secondary storage for modern computer systems.
- Each disk platter has a flat circular shape, like a CD. Common platter diameters range from 1.8 to 3.5 inches.



- The two surfaces of a platter are covered with a magnetic material. We store information by recording it magnetically on the platters.
- A read-write head "flies" just above each surface of every platter. The heads are attached to a **disk arm** that moves all the heads as a unit.
- The surface of a platter is logically divided into circular **tracks,** which are subdivided into **sectors.** The set of tracks that are at one arm position makes up a **cylinder.**
- There may be thousands of concentric cylinders in a disk drive, and each track may contains hundreds of sectors.
- When the disk is in use, a drive motor spins it at high speed. Most drives rotate 60 to 250 times per second, specified in terms of **rotations per minute (RPM).**
- Disk speed has two parts:
  - The **transfer rate** is the rate at which data flow between the drive and computer.
  - The **positioning time,** or **random-access time**, consists of two parts:
    - ✓ the time necessary to move the disk arm to the desired cylinder, called the **seek time,** and
    - ✓ the time necessary for the desired sector to rotate to the disk head, called the **rotational latency.**
- A disk drive is attached to a computer by a set of wires called an I/O bus. The data transfers on a bus are carried out by special electronic processors called **controllers.** The **host controller** is the controlled at the computer end of the bus.

- A **disk controller** is built into each disk drive. To perform a disk I/O operation, the computer places a command into the host controller, typically using memory-mapped I/O ports.
- The **host controller** then sends the command via messages to the disk controller, and the disk controller operates the disk-drive hardware to carry out the command.

1.2. Solid-State Disks

- An SSD is nonvolatile memory that is used like a hard drive.
- SSDs have the same characteristics as traditional hard disks but can be more reliable because they have no moving parts and faster because they have no seek time or latency.
- In addition, then consume less power. However, they are more expensive per megabyte than traditional hard disks, have less capacity than the larger hard disks, and may have shorter life spans than hard disks.
- One use for SSDs is in storage arrays, where they hold file-system metadata that require high performance.

1.3. Magnetic Tapes

- Magnetic tape was used as an early secondary-storage medium. It is permanent and can hold large quantities of data. Its access time is slow compared with that of main memory and magnetic disks.
- Tapes are mainly used for backup, for storage of infrequently used information and as a medium for transferring information from one system to another.
- A tape is kept in a spool and is wound or rewound past a read – write head. Moving to the correct spot on a tape can take minutes, but once positioned, tape drives can write data at speeds comparable to disk drives.
- Tapes and their drives are usually categorized by width, including 4, 8 and 19 millimeters and ¼ and ½ inch.

2. Disk Scheduling

- Disk scheduling algorithms are used to allocate the services to the I/O requests on the disk.
- If desired disk drive and controller is available, request is served immediately. If busy, new request for service will be placed in the queue of pending requests for that drive.
- When one request is completed, the Operating System has to choose which pending request to service next.
- The Operating system relies on the type of algorithm it needs when dealing and choosing what particular disk request is to be processed next.
- The objective of using these algorithms is keeping Head movements to the amount as less possible. The less the head to move, the faster the seek time will be.

2.1. FCFS Scheduling

- The simplest form of disk scheduling algorithm is the first-come, first-served (FCFS) algorithm.

- The I/O requests are served or processes according to their arrival. The request arrives first will be accessed and served first.
- Since it follows the order of arrival, it causes the wild swings from the innermost to the outermost tracks of the disk and vice versa. The farther the location of the request being serviced by the read/write head from its current location, the higher the seek time will be.

Example

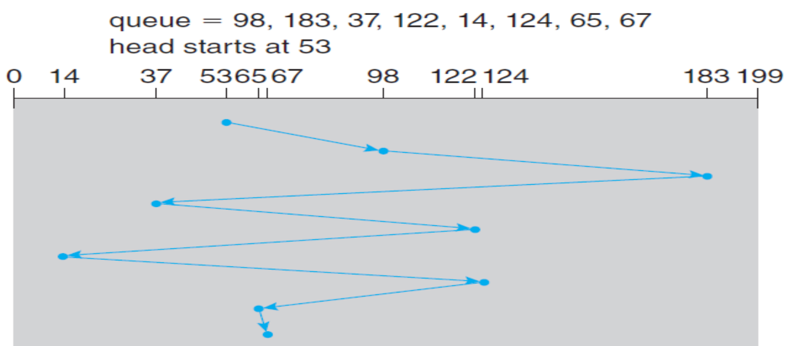


Figure 11.6 FCFS disk scheduling.

- Consider, for example, a disk queue with requests for I/O blocks on cylinders
  98, 183, 37, 122, 14, 124, 65, 67 in that order
- If the disk scheduling is initially at cylinder 53, it will first move from 53 to 98, then 183, 37, 122, 14, 124, 65, 67, for a total head movement of 640 cylinders.
- Total head movement computation (TMH)= (98-53)+(183-98)+(183-37)+(122-37)+(122-14)+(124-14)+(124-65)+(67-65) = **640** cylinders
- Assuming a seek rate of 5 milliseconds is given, Seek Time = TMH * Seek Rate = 3,200ms

2.2. SSTF Scheduling

- The SSTF algorithm selects the request with the least seek time from the current head position. In other words, SSTF chooses the pending request closes to the current head position.
- For our example, request queue, the closest request to the initial head position 53 is at cylinder 65. Once we are cylinder 65, the next closest request is at cylinder 67, from there, the request at cylinder 37 is closer than the one at 98, so 37 is served next. Continuing, we service the request at cylinder 14, then 98, 122, 124 and finally 183.
- This scheduling algorithm results in a total head movement of only **236** cylinders.
- This algorithm gives a substantial improvement of performance. SSTF scheduling is essentially a form of shortest-job first (SJF) scheduling; and it may cause starvation of some requests.

2.3. SCAN Scheduling

- In the SCAN algorithm, the disk arm starts at one end of the disk and moves towards the other end, servicing requests as it reaches each cylinder, until it gets to the other end of the disk. At the other end, the direction of head movement is reversed, and servicing continues. The head movement continuously scans back and forth across the disk.

- The scan algorithm sometimes called the **elevator algorithm,** since the disk arm behaves just like an elevator in a building, first servicing all the requests going up and then reversing to service requests the other way.

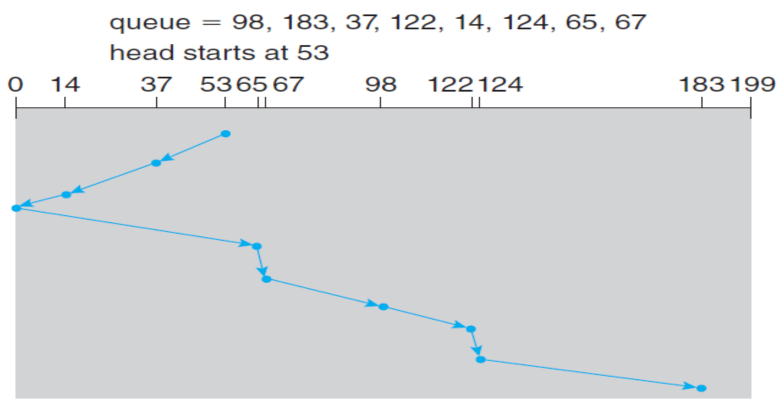


Figure 11.7 SCAN disk scheduling.

- Before applying SCAN to schedule the requests on cylinders 98, 183, 37, 122, 14, 124, 65 and 67, we need to know the direction of head movement in addition to the head's current position.
- Assuming that the disk arm is moving toward 0 and that the initial head position is at 53, the head will next service 37 and then 14.
- At cylinder 0, the arm will reverse and will move toward the other end of the disk, servicing the requests at 65, 67, 98, 122, 124 and 183.
- If a request arrives in the queue just in front of the head, it will be serviced almost immediately; a request arriving just behind the head will have to wait until the arm moves to the end of the disk, reverses direction, and comes back.

2.4. C-Scan Scheduling

- Circular SCAN Scheduling is a variant of SCAN designed to provide a more uniform wait time.
- Like SCAN, C-SCAN moves the head from one end of the disk to the other, servicing the requests along the way. When the head reaches the other end, however, it immediately returns to the beginning of the disk without servicing any requests on the return trip.
- The C-SCAN scheduling algorithm essentially treats the cylinders as a circular list that wraps around from the final cylinder to the first one.

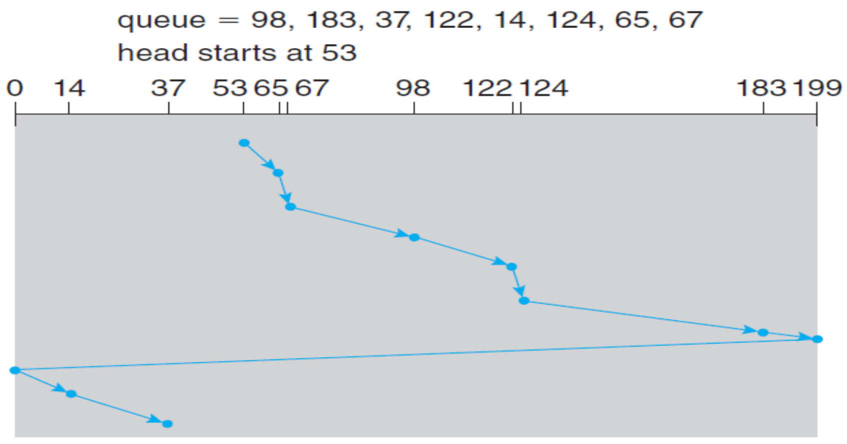


Figure 11.8 C-SCAN disk scheduling.

2.5. LOOK Scheduling

- Both SCAN and C-SCAN move the disk arm across the full width of the disk. In practice, neither algorithm is often implemented this way.
- More commonly, the arm goes only as far as the final request in each direction. Then it reverses direction immediately, without going all the way to the end of the disk.
- Versions of SCAN and C-SCAN that follow this pattern are called LOOK and C-LOOK scheduling, because they ***look*** for a request before continuing to move in a given direction.

2.6. Selection of a Disk-Scheduling Algorithm

- SSTF is common and has a natural appeal because it increases performance over FCFS.
- SCAN and C-SCAN perform better for systems that place a heavy load on the disk, because they are less likely to cause a starvation problem.
- To avoid starvation Linux implements **deadline** scheduler
  - ▪ Maintains separate read and write queues, gives read priority
    - ✓ Because processes more likely to block on read than write
  - ▪ Implements four queues: 2x read and 2x write
    - ✓ 1 read and 1 write queue sorted in LBA order, essentially implementing C-SCAN
    - ✓ 1 read and 1 write queue sorted in FCFS order
    - ✓ All I/O requests sent in batch sorted in that queue's order
    - ✓ After each batch, checks if any requests in FCFS older than configured age (default 500ms)
      - o If so, LBA queue containing that request is selected for next batch of I/O