Module IV
Virtual Memory Management

4.1 Introduction /Background :

The memory-management algorithms are necessary because of one basic requirement: The instructions being executed must be in physical memory. An examination of real programs shows us that, in many cases, the entire program is not needed. For instance, consider the following:

- Programs often have code to handle unusual error conditions. Since these errors seldom, if ever, occur in practice, this code is almost never executed.
- Arrays, lists, and tables are often allocated more memory than they actually need. An array may be declared 100 by 100 elements, even though it is seldom larger than 10 by 10 elements.
- Certain options and features of a program may be used rarely.

The ability to execute a program that is only partially in memory would confer many benefits:

- A program would no longer be constrained by the amount of physical memory that is available.
- Because each user program could take less physical memory, more programs could be run at the same time, with a corresponding increase in CPU utilization and throughput
- Less I/O would be needed to load or swap each user program into memory, so each user program would run faster.

Virtual memory involves the separation of logical memory as perceived by users from physical memory. This separation, allows an extremely large virtual memory to be provided for programmers when only a smaller physical memory is available (Figure 9.1).

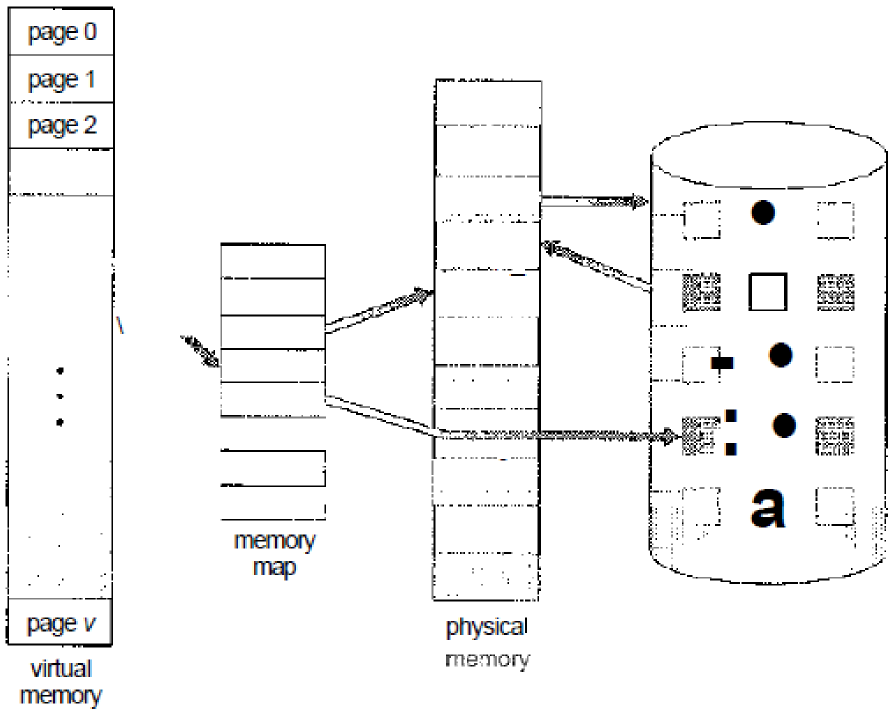


Figure 9.1 Diagram showing virtual memory that is larger than physical memory.

The **virtual address space** of a process refers to the logical (or virtual) view of how a process is stored in memory. Typically, this view is that a process begins at a certain logical address—say, addresses 0—and exists in contiguous memory, as shown in Figure 9.2. We allow for the heap to grow upward hi memory as it is used for dynamic memory allocation. Similarly, we allow for the stack to grow downward in memory through successive function calls. The large blank space (or hole) between the heap and the stack is part of the virtual address space.

Virtual memory also allows files and memory to be shared by two or more processes through page sharing. This leads to the following benefits:

- System libraries can be shared by several processes through mapping of the shared object into a virtual address space.
- Virtual memory enables processes to share memory.
- Virtual memory can allow pages to be shared during process creation with the fork( ) system call, thus speeding up process creation.

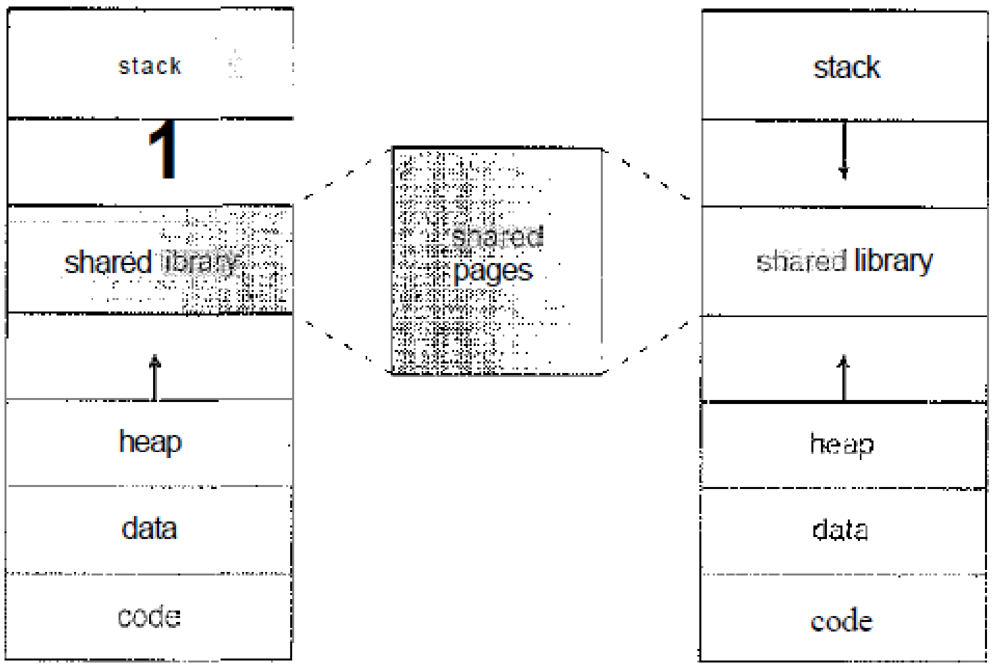


Figure 9.3 Shared library using virtual memory.

4.2 Demand Paging

- A demand-paging system is similar to a paging system with swapping where processes reside in secondary memory.
- When we want to execute a process, we swap it into memory.
- Rather than swapping the entire process into memory, however, we use a **lazy swapper.** A lazy swapper never swaps a page into memory unless that page will be needed.

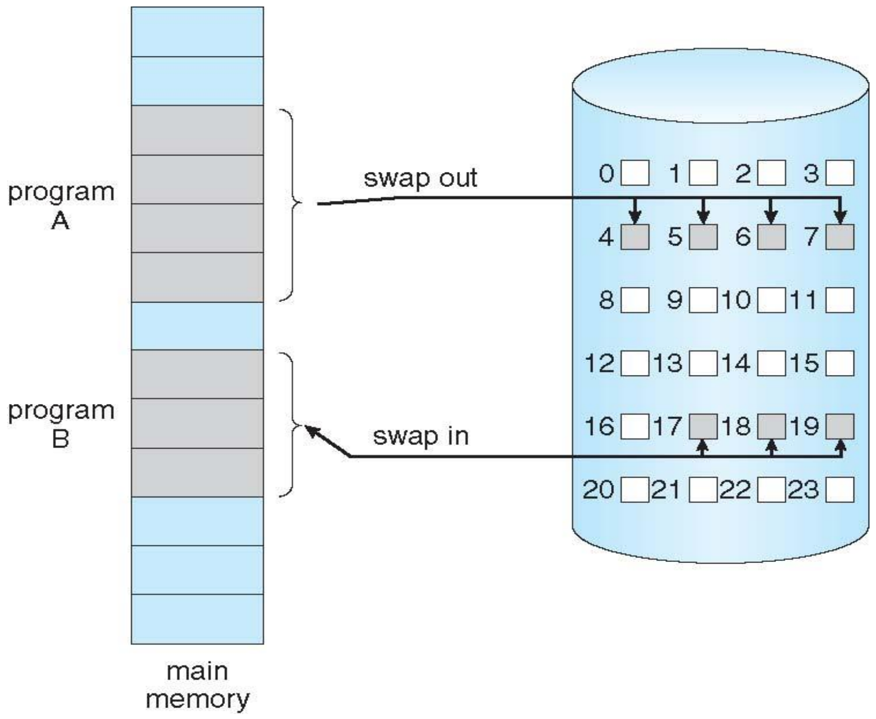


Figure 5.8 Transfer of a paged memory to contiguous disk space.

- 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.
- In page table we maintain a extra bit called valid-invalid bit to know the status of page.
- If it is set to 1 then page is valid and present in primary memory, then we calculate the physical address and continue execution.
- If it is set to 0, then page is not present in primary memory, such a state is called **page fault**. Following procedure is used to handle Page faults.
  1. We check an page table for this process to determine whether the reference was a valid or an invalid memory access.
  2. If the reference was invalid, we terminate the process. Generate trap to operating systems.
  3. OS brings page from secondary storage.
  4. Find a free frame if available, else use page replacement algorithm to remove one page from primary memory to make fare free.
  5. We modify the page table to indicate that the page is now in memory.
  6. We restart the instruction that was interrupted by the trap. The process can now access the page as though it had always been in memory.

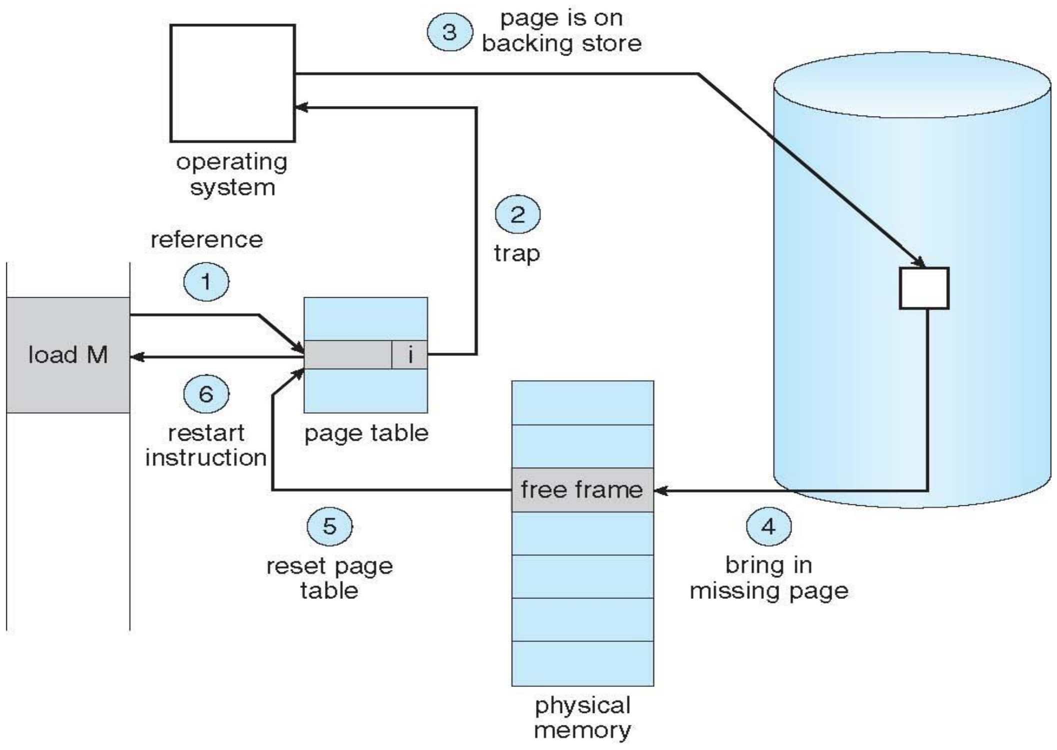


Figure 5.9 Steps in handling a page fault

Performance of Demand Paging

- Demand paging can significantly affect the performance of a computer system. To see why, let's compute the **effective access time** for a demand-paged memory. For most computer systems, the memory-access time, denoted *ma,* ranges from 10 to 200 nanoseconds. As long as we have no page faults, the effective access time is equal to the memory access time. If, however, a page fault occurs, we must first read the relevant page from disk and then access the desired word.
- Let p be the probability of a page fault ($0 <= p <= 1$). We would expect p to be close to zero—that is, we would expect to have only a few page faults.
- The effective access time is = $(1 - p) * ma + p *$ page fault time.
- If we take an average page-fault service time of 8 milliseconds and a memory-access time of 200 nanoseconds, then the effective access time in nanoseconds is

$$\text{Effective access time} = (1 - p) * (200) + p \text{ (8 milliseconds)}$$
$$= (1 - p) * 200 + p * 8.00(1000000)$$
$$= 200 + 7{,}999{,}800 * p.$$

- If one access out of 1,000 causes a page fault, the effective access time is 8.2 microseconds. That is page fault is 1/10 or 0.1%.

4.3 Copy-on-Write

- We know that the fork() system call creates a child process as a duplicate of its parent.
- Traditionally, fork() worked by creating a copy of the parent's address space for the child, duplicating the pages belonging to the parent. However, considering that many

child processes invoke the exec() system call immediately after creation, the copying of the parent's address space may be unnecessary.

- Alternatively, we can use a technique known **as copy-on-write**, which works by allowing the parent and child processes initially to share the same pages. These shared pages are marked as copy-on-write pages, meaning that if either process writes to a shared page, a copy of the shared page is created.
- Copy-on-write is illustrated in Figures 5.10 and Figure 5.11, which show the contents of the physical memory before and after process 1 modifies page C.
- For example, assume that the child process attempts to modify a page containing portions of the stack, with the pages set to be copy-on-write.
- The operating system will then create a copy of this page, mapping it to the address space of the child process.
- The child process will then modify its copied page and not the page belonging to the parent process.
- Obviously, when the copy-on-write technique is used, only the pages that are modified by either process are copied; all unmodified pages can be shared by the parent and child processes.

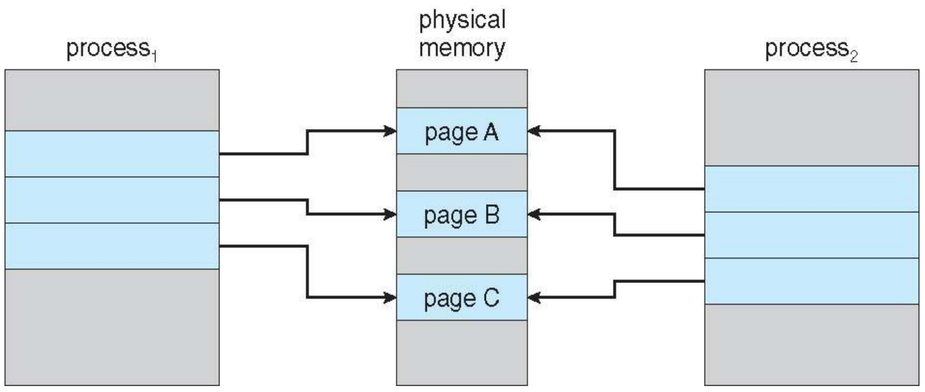


Figure 5.10 Before process 1 modifies page C.

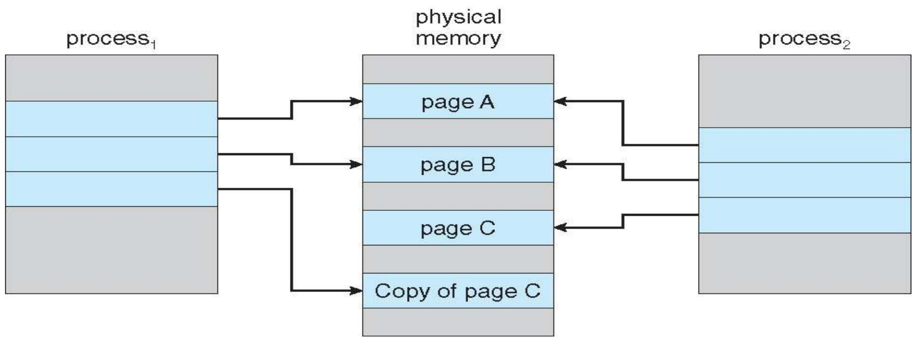


Figure 5.11 After process 1 modifies page C.

4.4 Page Replacement

- In demand paging or virtual memory management systems, we bring only those pages into primary memory which is required at any time.

- If there is no free frame in primary memory and CPU wants to execute a page which is not present in primary memory this is called page fault. Then we need to remove one page from primary memory and bring new page from secondary storage.
- Page replacement is process of selecting a page in primary memory to make a way for new page.

4.4.1 Basic Page Replacement

- Page replacement takes the following approach. If no frame is free, we find one that is not currently being used and free it.
- We can free a frame by writing its contents to swap space and changing the page table (and all other tables) to indicate that the page is no longer in memory (Figure 5.12).
- We can now use the freed frame to hold the page for which the process faulted. We modify the page-fault service routine to include page replacement.

- **Steps in Page replacement**
  1. Find the location of the desired page on the disk.
  2. Find a free frame:
     a. If there is a free frame, use it. If there is no free frame, use a page-replacement algorithm to select a victim frame.
     b. Write the victim frame to the disk; change the page and frame tables accordingly.
     c. Read the desired page into the newly freed frame; change the page and frame tables.
  3. Restart the user process.

- If no frames are free, *two* page transfers (one out and one in) are required. This situation effectively doubles the page-fault service time and increases the effective access time accordingly.
- We can reduce this overhead by using a **modify bit** (or **dirty bit).** When this scheme is used, each page or frame has a modify bit associated with it in the hardware.
- The modify bit for a page is set by the hardware whenever any word or byte in the page is written into, indicating that the page has been modified.
- When we select a page for replacement, we examine its modify bit. If the bit is set, we know that the page has been modified since it was read in from the disk. In this case, we must write that page to the disk.
- If the modify bit is not set, however, the page has *not* been modified since it was read into memory. Therefore, if the copy of the page on the disk has not been overwritten (by some other page, for example), then we need not write the memory page to the disk: It is already there.
- This technique also applies to read-only pages (for example, pages of binary code). Such pages cannot be modified; thus, they may be discarded when desired. This scheme can significantly reduce the time required to service a page fault, since it reduces I/O time by one-half *if* the page has not been modified.

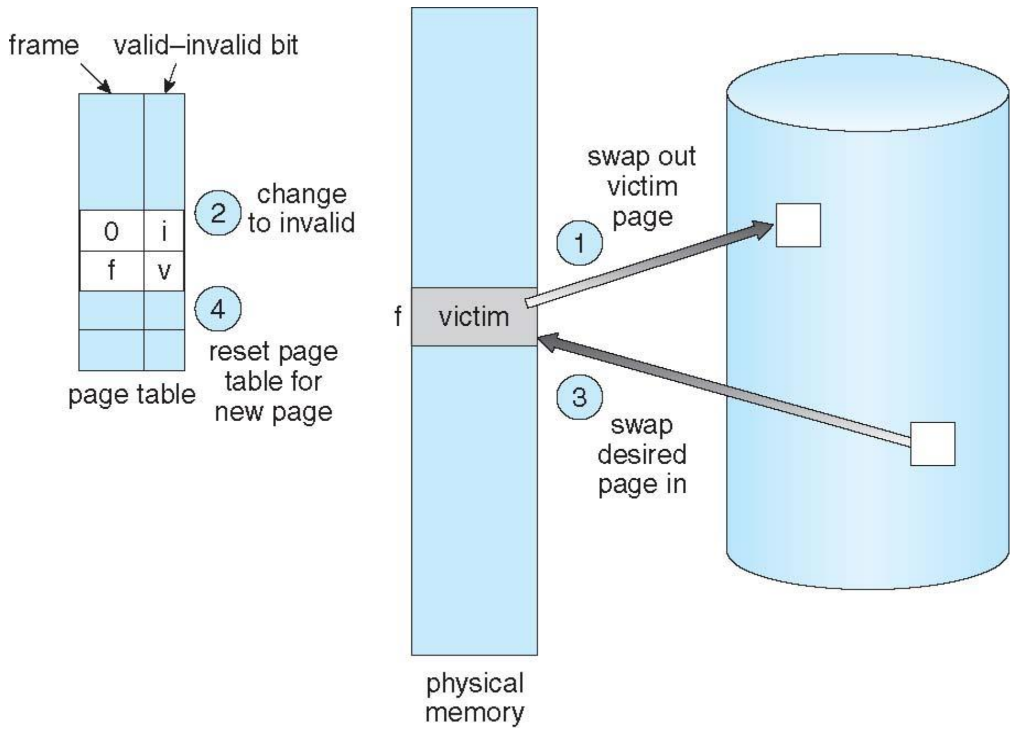


Figure 5.12 Page replacement

4.5 Page Replacement Algorithms

- We evaluate an algorithm by running it on a particular string of memory references and computing the number of page faults. The string of memory references is called a **reference string.**
- For example, if we trace a particular process, we might record the following address sequence:
- 0100, 0432, 0101,0612, 0102, 0103, 0104, 0101, 0611, 0102, 0103, 0104,0101,0610, 0102, 0103, 0104, 0101, 0609, 0102, 0105
- At 100 bytes per page, this sequence is reduced to the following reference string: 1,4,1,6,1,6,1,6,1,6,1
- For illustrating different page replacement algorithms, consider the following reference string 7, 0,1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2,1, 2, 0, 1, 7, 0,1 for a memory with three frames.

4.5.1 FIFO Page Replacement

- The simplest page-replacement algorithm is a first-in, first-out (FIFO) algorithm. A FIFO replacement algorithm associates with each page the time when that page was brought into memory. When a page must be replaced, the oldest page is chosen.
- Figure 5.13 shows the FIFO place replacement algorithm.

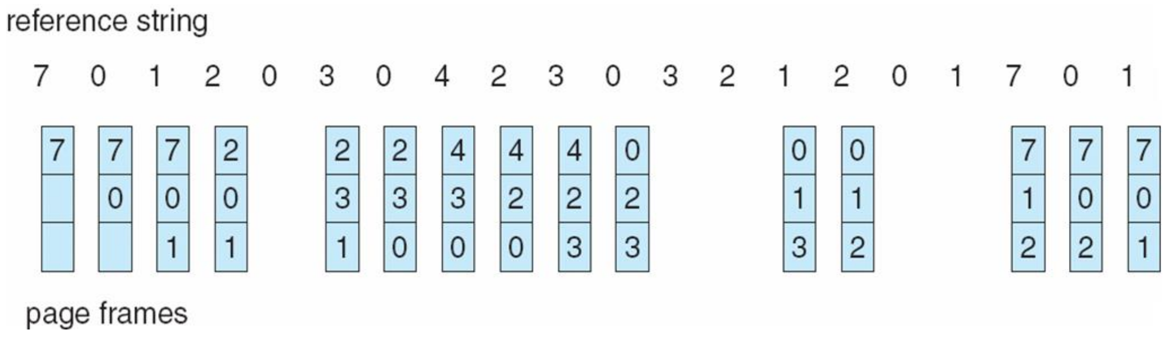


Figure 5.13 FIFO page-replacement algorithm.

- To illustrate the problems that are possible with a FIFO page-replacement algorithm., we consider the following reference string: 1,2,3,4,1,2,5,1,2,3,4,5
- Figure 5.14 shows the curve of page faults for this reference string versus the number of available frames. Notice that the number of faults for four frames (ten) is *greater* than the number of faults for three frames (nine)! This most unexpected result is known as Belady's **anomaly:** For some page-replacement algorithms, the page-fault rate may *increase* as the number of allocated frames increases.

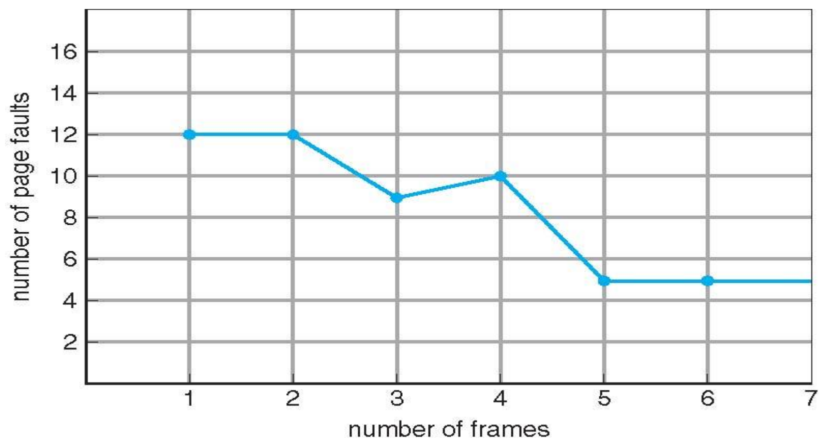


Figure 5.14 Page-fault curve for FIFO replacement on a reference string.

4.5.2 Optimal Page Replacement

- An optimal page-replacement algorithm has the lowest page-fault rate of all algorithms and will never suffer from Belady's anomaly.
- Use of this page-replacement algorithm guarantees the lowest possible page fault rate for a fixed number of frames. For example, on our sample reference string, the optimal page-replacement algorithm would yield nine page faults, as shown in Figure 5.15.
- **Replace the page that will not be used for the longest period of time.**

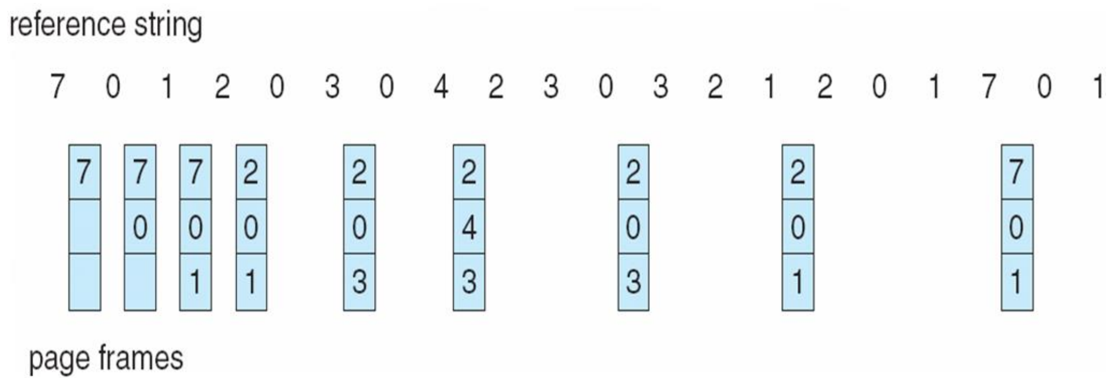


Figure 5.15 Optimal page-replacement algorithm.

4.5.3 LRU Page Replacement

- If the optimal algorithm is not feasible, perhaps an approximation of the optima] algorithm is possible.
- The key distinction between the FIFO and OPT algorithms (other than looking backward versus forward in time) is that the FIFO algorithm uses the time when a page was brought into memory, whereas the OPT algorithm uses the time when a page is to be *used.*

- If we use the recent past as an approximation of the near future, then we can replace the page that *has not been used* for the longest period of time (Figure 5.16). This approach is the **least-recently-used** (LRU) **algorithm.**
- **Replace the page that *has not been used* for the longest period of time**

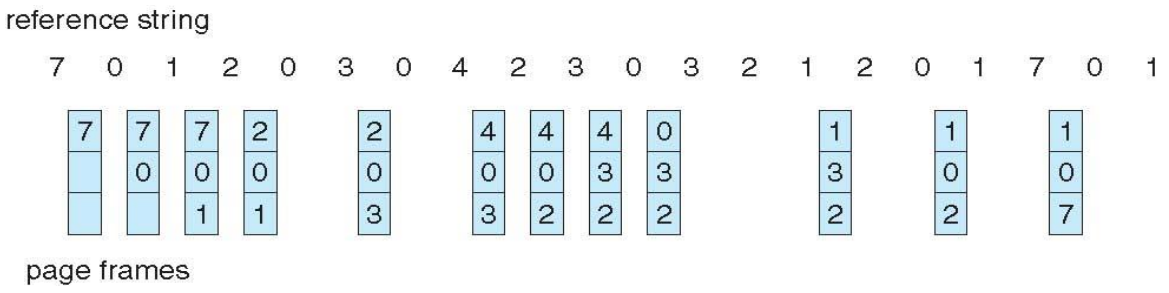


Figure 5.16 LRU page-replacement algorithm.

Counters: In the simplest case, we associate with each page-table entry a time-of-use field and add to the CPU a logical clock or counter. The clock is incremented for every memory reference. Whenever a reference to a page is made, the contents of the clock register are copied to the time-of-use field in the page-table entry for that page. In this way, we always have the "time" of the last reference to each page. We replace the page with the smallest time value.

Stack: Another approach to implementing LRU replacement is to keep a stack of page numbers. Whenever a page is referenced, it is removed from the stack and put on the top. In this way, the most recently used page is always at the top of the stack and the least recently used page is always at the bottom (Figure 9.16).

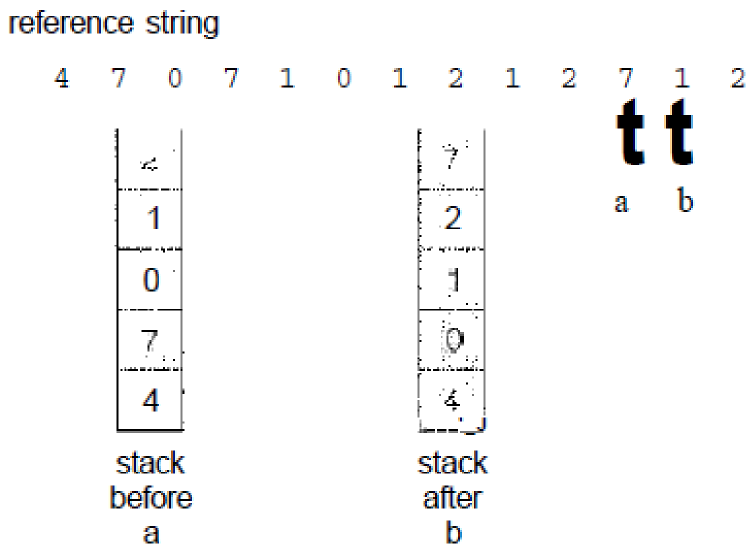


Figure 9.16 Use of a stack to record the most recent page references.

4.5.4 LRU-Approximation Page Replacement

- Few computer systems provide sufficient hardware support for true LRU page replacement. Some systems provide no hardware support.
- Many systems provide some help, however, in the form of a reference bit.
- The reference bit for a page is set by the hardware whenever that page is referenced (either a read or a write to any byte in the page).
- Reference bits are associated with each entry in the page table.
- Initially, all bits are cleared (to 0) by the operating system.
- As a user process executes, the bit associated with each page referenced is set (to 1) by the hardware.
- After some time, we can determine which pages have been used and which have not been used by examining the reference bits, although we do not know the *order* of use.

4.5.4.1 Additional-Reference-Bits Algorithm

- We can gain additional ordering information by recording the reference bits at regular intervals.
- We can keep an 8-bit byte for each page in a table in memory.
- At regular intervals (say, every 100 milliseconds), a timer interrupt transfers control to the operating system.
- The operating system shifts the reference bit for each page into the high-order bit of its 8-bit byte, shifting the other bits right by 1 bit and discarding the low-order bit.
- These 8-bit shift registers contain the history of page use for the last eight time periods.

4.5.4.2 Second-Chance Algorithm

- The basic algorithm of second-chance replacement is a FIFO replacement algorithm.
- When a page has been selected, however, we inspect its reference bit.
- If the value is 0, we proceed to replace this page; but if the reference bit is set to 1, we give the page a second chance and move on to select the next FIFO page.
- When a page gets a second chance, its reference bit is cleared, and its arrival time is reset to the current time.
- Thus, a page that is given a second chance will not be replaced until all other pages have been replaced.
- One way to implement the second-chance algorithm (sometimes referred to as the *dock* algorithm) is as a circular queue.

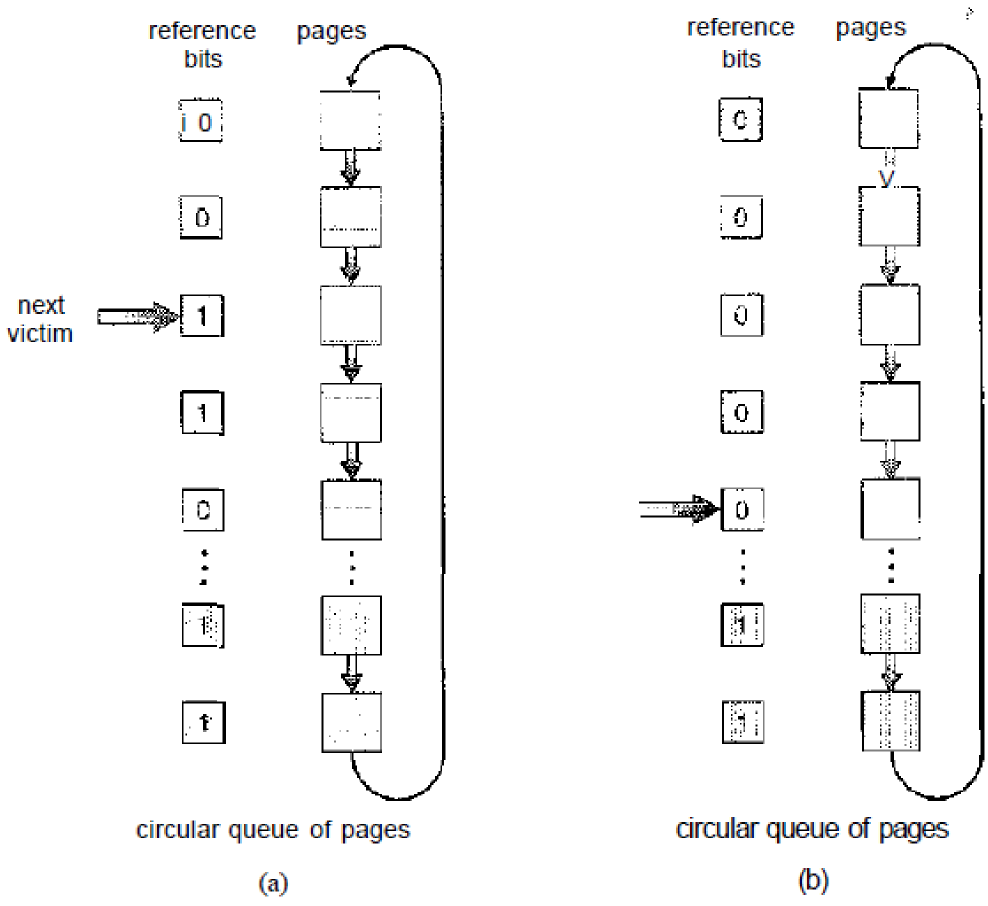


Figure 9.17 Second-chance (clock) page-replacement algorithm.

4.5.4.3 Enhanced Second-Chance Algorithm

We can enhance the second-chance algorithm by considering the reference bit and the modify bit as an ordered pair. With these two bits, we have the following four possible classes:

- (0, 0) neither recently used nor modified—best page to replace
- (0, 1) not recently used but modified—not quite as good, because the page will need to be written out before replacement
- (1., 0) recently used but clean—probably will be used again soon
- (1,1) recently used and modified—probably will be used again soon, and the page will be need to be written out to disk before it can be replaced

4.6 Counting-Based Page Replacement:

There are many other algorithms that can be used for page replacement. For example, we can keep a counter of the number of references that have been made to each page and develop the following two schemes.

• The **least frequently used (LFU) page-replacement algorithm** requires that the page with the smallest count be replaced. The reason for this selection is that an actively used page should have a large reference count.

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

4.7 Page-Buffering Algorithms:

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

4.8 Allocation of Frames

- How do we allocate the fixed amount of free memory among the various processes? If we have 93 free frames and two processes, how many frames does each process get?
- The simplest case is the single-user system. Consider a single-user system with 128 KB of memory composed of pages 1 KB in size. This system has 128 frames.
- The operating system may take 35 KB, leaving 93 frames for the user process. Under pure demand paging, all 93 frames would initially be put on the free-frame list. When a user process started execution, it would generate a sequence of page faults.
- The first 93 page faults would all get free frames from the free-frame list. When the free-frame list was exhausted, a page-replacement algorithm would he used to select one of the 93 in-memory pages to be replaced with the 94th, and so on.
- When the process terminated, the 93 frames would once again be placed on the free-frame list.

4.8.1. Minimum Number of Frames

- We must also allocate at least a minimum number of frames. Here, we look more closely at the latter requirement.
- One reason for allocating at least a minimum number of frames involves performance. Obviously, as the number of frames allocated to each process decreases, the page-fault rate increases, slowing process execution.
- In addition, remember that, when a page fault occurs before an executing instruction is complete, the instruction must be restarted.
- Consequently, we must have enough frames to hold all the different pages that any single instruction can reference.

4.8.2. Allocation Algorithms

- The easiest way to split *m* frames among *n* processes is to give everyone an equal share, *m/n* frames.
- For instance, if there are 93 frames and five processes, each process will get 18 frames. The leftover three frames can be used as a free-frame buffer pool.
- This scheme is called **equal allocation.**
- An alternative is to recognize that various processes will need differing amounts of memory. Consider a system with a 1-KB frame size. If a small student process of 10 KB and an

interactive database of 127 KB are the only two processes running in a system with 62 free frames, it does not make much sense to give each process 31 frames. The student process does not need more than 10 frames, so the other 21 are, strictly speaking, wasted. To solve this problem, we can use **proportional** allocation, in which we allocate available memory to each process according to its size.

4.8.3. Global versus Local Allocation

- Another important factor in the way frames are allocated to the various processes is page replacement. With multiple processes competing for frames, we can classify page-replacement algorithms into two broad categories: **global replacement** and **local replacement.**
- Global replacement allows a process to select a replacement frame from the set of all frames, even if that frame is currently allocated to some other process; that is, one process can take a frame from another. Local replacement requires that each process select from only its own set of allocated frames.
- For example, consider an allocation scheme where we allow high-priority processes to select frames from low-priority processes for replacement. A process can select a replacement from among its own frames or the frames of any lower-priority process. This approach allows a high-priority process to increase its frame allocation at the expense of a low-priority process.

4.9 Thrashing

- If the process does not have the number of frames it needs to support pages in active use, it will quickly page-fault. At this point, it must replace some page.
- However, since all its pages are in active use, it must replace a page that will be needed again right away. Consequently, it quickly faults again, and again, and again, replacing pages that it must bring back in immediately.
- This high paging activity is called **thrashing.** A process is thrashing if it is spending more time paging than executing.

4.9.1 Cause of Thrashing

- Thrashing results in severe performance problems.
- 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 without regard to the process to which they belong.
- Now suppose that a process enters a new phase in its execution and needs more frames. It starts faulting and taking frames away from other processes.
- These processes need those pages, however, and so they also fault, taking frames from other processes. These faulting processes must use the paging device to swap pages in and out. As they queue up for the paging device, the ready queue empties. As processes wait for the paging device, CPU utilization decreases.

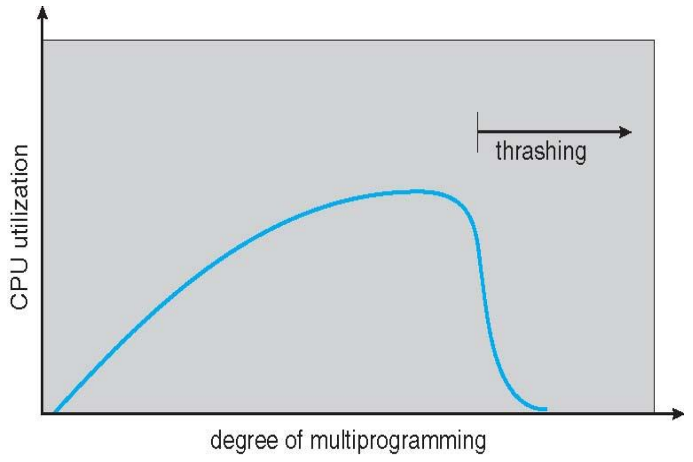


Figure 5.17 Thrashing.

- We can limit the effects of thrashing by using a **local replacement algorithm** (or **priority replacement algorithm).** With local replacement, if one process starts thrashing, it cannot steal frames from another process and cause the latter to thrash as well.

4.9.2 Working-Set Model

- The working-set model is based on the assumption of locality.
- This model uses a parameter, A, to define the working-set window.
- The idea is to examine the most recent A page references.
- The set of pages in the most recent A page references is the working set (Figure 9.20).

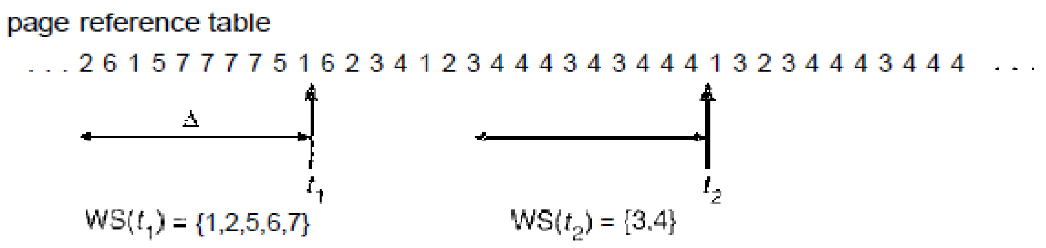


Figure 9.20 Working-set modef.

- If a page is in, active use, it will be in the working set.
- If it is no longer being used, it will drop from the working set A time units after its last reference.
- Thus, the working set is an approximation of the program's locality.
- The most important property of the working set, then, is its size.
- If we compute the working-set size, *WSSj*, for each process in the system, we can then consider that

$$D = \sum WSS_i ,$$

where *D* is the total demand for frames. Each process is actively using the pages in its working set.

4.9.3 Page-Fault Frequency :

- The working-set model is successful, and knowledge of the working set can be useful for prepaging but it seems a clumsy way to control thrashing.
- A strategy that uses the **page-fault frequency (PFF)** takes a more direct approach.
- Thrashing has a high page-fault rate.
- Thus, we want to control the page-fault rate.
- When it is too high, we know that the process needs more frames.
- Conversely, if the page-fault rate is too low, then the process may have too many frames.
- We can establish upper and lower bounds on the desired page-fault rate (Figure 9.21).
- If the actual page-fault rate exceeds the upper limit, we allocate the process another frame;
- If the page-fault rate falls below the lower limit, we remove a frame from the process.
- Thus, we can directly measure and control the page-fault rate to prevent thrashing.

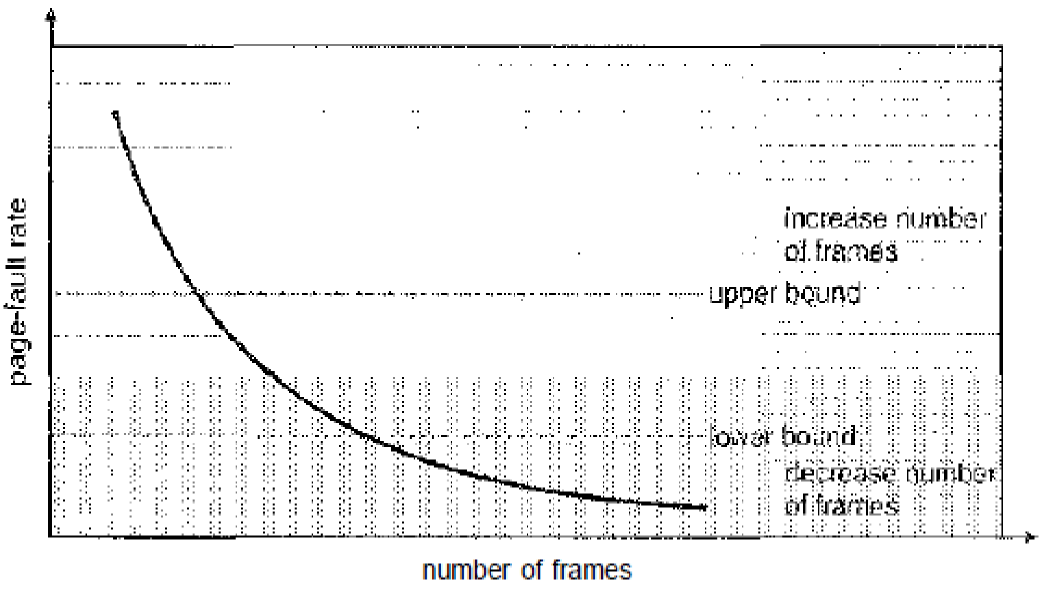


Figure 9.21 Page-fault frequency.

File System and Implementation

6.1 Introduction

- Computers can store information on various storage media, such as magnetic disks, magnetic tapes, and optical disks.
- The operating system abstracts from the physical properties of its storage devices to define a logical storage unit***, the file***.
- The information in a file is defined by its creator.
- Many different types of information may be stored in a file—source programs, object programs, executable programs, numeric data, text, payroll records, graphic images, sound recordings, and so on.
- A ***text*** file is a sequence of characters organized into lines, a ***source*** file is a sequence of subroutines and functions, each of which is further organized as declarations followed by executable statements, an ***object*** file is a sequence of bytes organized into blocks understandable by the system's linker and an ***executable*** file is a series of code sections that the loader can bring into memory and execute.

6.2 File Attributes

A file's attributes vary from one operating system to another but typically consist of these:

- **Name**. The symbolic file name is the only information kept in human readable form.
- **Identifier**. This unique tag, usually a number, identifies the file within the file system; it is the non-human-readable name for the file.
- **Type**. This information is needed for systems that support different types of files.
- **Location**. This information is a pointer to a device and to the location of the file on that device.
- **Size**. The current size of the file (in bytes, words, or blocks) and possibly the maximum allowed size are included in this attribute.
- **Protection**. Access-control information determines who can do reading, writing, executing, and so on.
- **Time**, **date**, and **user identification**. This information may be kept for creation, last modification, and last use. These data can be useful for protection, security, and usage monitoring.

6.3 File Operation

A file is an **abstract data type.** To define a file properly, we need to consider the operations that can be performed on files.

- **Creating a file.** Two steps are necessary to create a file. **First**, space in the file system must be found for the file. **Second**, an entry for the new file must be made in the directory.
- **Writing a file.** To write a file, we make a system call specifying both the name of the file and the information to be written to the file. Given the name of the file, the system searches the directory to find the file's location.
- **Reading a file.** To read from a file, we use a system call that specifies the name of the file and where the next block of the file should be put.
- **Repositioning within** a **file.** The directory is searched for the appropriate entry, and the current-file-position pointer is repositioned to a given value. Repositioning within a file need not involve any actual I/O. This file operation is also known as a file *seek.*
- **Deleting a file.** To delete a file, we search the directory for the named file. Having found the associated directory entry, we release all file space, so that it can be reused by other files, and erase the directory entry.

- **Truncating a file.** The user may want to erase the contents of a file but keep its attributes. Rather than forcing the user to delete the file and then recreate it, this function allows all attributes to remain unchanged—except for file length—but lets the file be reset to length zero and its file space released.
- Other common operations include ***appending*** new information to the end of an existing file and ***renaming*** an existing file.

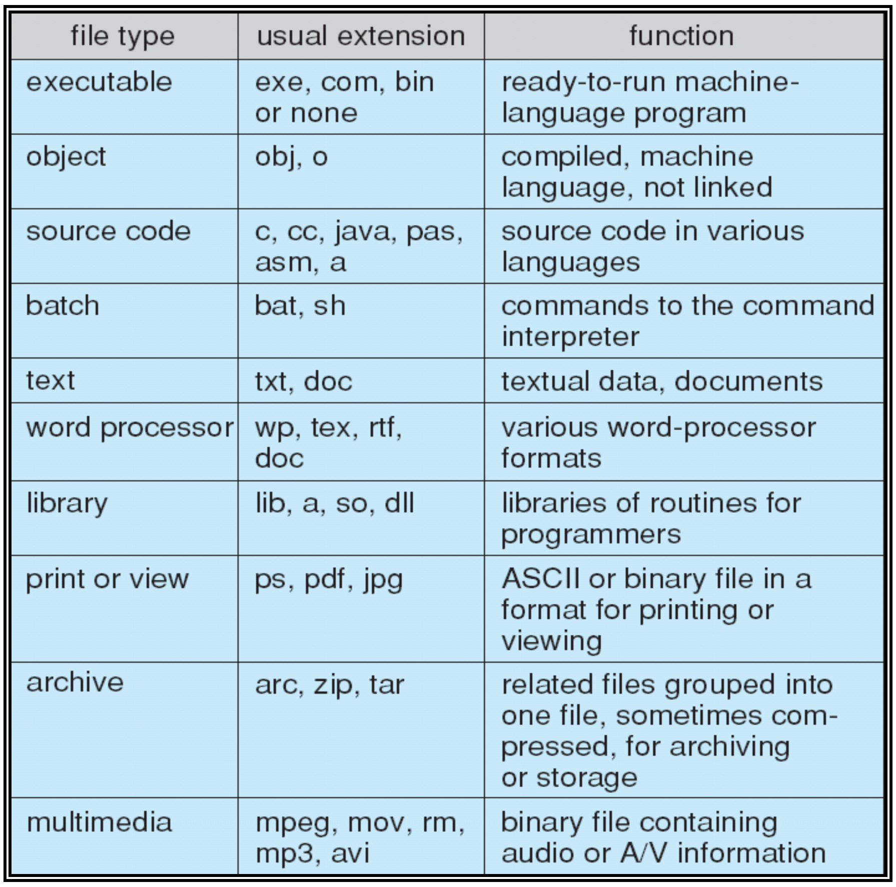
Most of the file operations mentioned involve searching the directory for the entry associated with the named file. To avoid this constant searching, many systems require that an open() system call be made before a file is first used actively. The operating system keeps a small table, called the **open-file table,** containing information about all open files. When a file operation is requested, the file is specified via an index into this table, so no searching is required. When the file is no longer being actively used, it is *closed* by the process, and the operating system removes its entry from the open-file table.

6.4 Information associated with an open file

- **File pointer**. On systems that do not include a file offset as part of the read() and write () system calls, the system must track the last read write location as a current-file-position pointer. This pointer is unique to each process operating on the file and therefore must be kept separate from the on-disk file attributes.
- **File-open count.** As files are closed, the operating system must reuse its open-file table entries, or it could run out of space in the table. Because multiple processes may have opened a file, the system must wait for the last file to close before removing the open-file table entry. The file-open counter tracks the number of opens and closes and reaches zero on the last close. The system can then remove the entry.
- **Disk location of the file.** Most file operations require the system to modify data within the file. The information needed to locate the file on disk is kept in memory so that the system does not have to read it from disk for each operation.
- **Access rights.** Each process opens a file in an access mode. This information is stored on the per-process table so the operating system can allow or deny subsequent I/O requests.

6.5 File Types

- When we design a file system—indeed, an entire operating system—we always consider whether the operating system should recognize and support file types.
- If an operating system recognizes the type of a file, it can then operate on the file in reasonable ways. For example, a common mistake occurs when a user tries to print the binary-object form of a program. This attempt normally produces garbage; however, the attempt can succeed *if* the operating system has been told that the file is a binary-object program.
- A common technique for implementing file types is to include the type as part of the file name. The name is split into **two parts**—a **name** and an ***extension***, usually separated by a period character. Figure 6.1 shows common file types.



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

Figure 6.1 Common file types

6.6 Internal File Structure

- Internally, locating an offset within a file can be complicated for the operating system. Disk systems typically have a well-defined block size determined by the size of a sector. All disk I/O is performed in units of one block (physical record), and all blocks are the same size. It is unlikely that the physical record size will exactly match the length of the desired logical record. Logical records may even vary in length. Packing a number of logical records into physical blocks is a common solution to this problem.
- For example, the UNIX operating system defines all files to be simply streams of bytes. Each byte is individually addressable by its offset from the beginning (or end) of the file. In this case, the logical record size is 1 byte. The file system automatically packs and unpacks bytes into physical disk blocks— say, 512 bytes per block—as necessary.
- The logical record size, physical block size, and packing technique determine how many logical records are in each physical block. The packing can be done either by the user's application program or by the operating system.
- Because disk space is always allocated in blocks, some portion of the last block of each file is generally wasted. If each block were 512 bytes, for example, then a file of 1,949 bytes would be allocated four blocks (2,048 bytes); the last 99 bytes would be wasted. The waste incurred to keep everything in units of blocks (instead of bytes) is **internal fragmentation**. All file systems suffer from internal fragmentation; the larger the block size, the greater the internal fragmentation.

6.7 Access Methods

Files store information. When it is used, this information must be accessed and read into computer memory.

6.7.1 Sequential Access

- Information in the file is processed in order, one record after the other. This mode of access is by far the most common; for example, editors and compilers usually access files in this fashion.
- Reads and writes make up the bulk of the operations on a file.
- A **read** operation—***read next***—reads the next portion of the file and automatically advances a file pointer, which tracks the I/O location.
- The **write** operation—***write next***—appends to the end of the file and advances to the end of the newly written material. Such a file can be reset to the beginning; and on some systems, a program may be able to skip forward or backward *n* records for some integer *n*—perhaps only for *n* = 1.

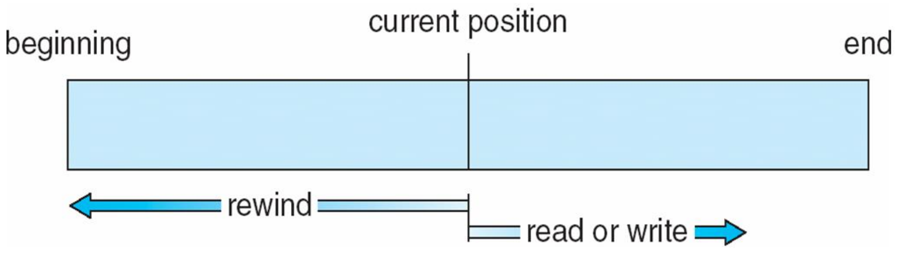
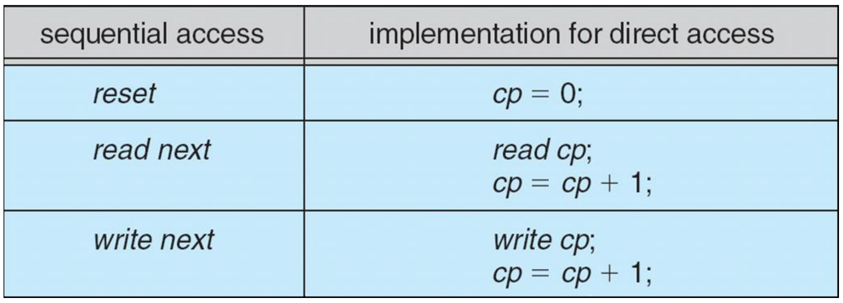


Figure 6.2 Sequential Access

6.7.2 Direct Access

- File is made up of fixed length **logical records** that allow programs to read and write records rapidly in no particular order.
- The direct-access method is based on a disk model of a file, since disks allow random access to any file block.
- For direct access, the file is viewed as a numbered sequence of blocks or records. Thus, we may read block 14, then read block 53, and then write block 7. There are no restrictions on the order of reading or writing for a direct-access file.
- For the direct-access method, the file operations must be modified to include the block number as a parameter.
- Thus, we have ***read n***, where *n* is the block number, rather than ***read next***, and ***write n*** rather than ***write next***. An alternative approach is to retain *read next* and *write next*, as with sequential access, and to add an operation ***position*** *file to* ***n***, where **n** is the block number. Then, to affect a *read n*, we would, ***position to n*** and then ***read next***.



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

Figure 6.3 Simulation of sequential access on a direct-access file.

6.8 Directory Storage Structure

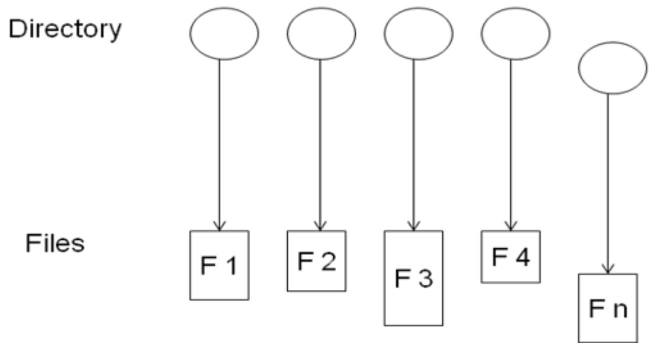


Figure 6.5 Directory vs. Files

A disk can be used in its entirety for a file system. Sometimes, though, it is desirable to place multiple file systems on a disk or to use parts of a disk for a file system and other parts for other things, such as swap space or unformatted (raw) disk space. These parts are known variously as **partitions, slices,** or (in the IBM world) **minidisks.** A file system can be created on each of these parts of the disk.

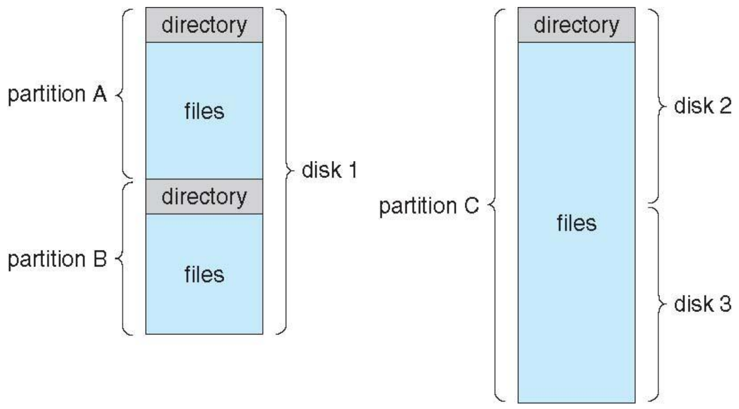


Figure 6.6 A typical file-system organization

6.9 Operations on Directory

- **Search for a file**. We need to be able to search a directory structure to find the entry for a particular file. Since files have symbolic names and similar names may indicate a relationship between files, we may want to be able to find all files whose names match a particular pattern.
- **Create a file.** New files need to be created and added to the directory.
- **Delete a file.** When a file is no longer needed, we want to be able to remove it from the directory.
- **List a directory.** We need to be able to list the files in a directory and the contents of the directory entry for each file in the list.
- **Rename a file.** Because the name of a file represents its contents to its users, we must be able to change the name when the contents or use of the file changes. Renaming a file may also allow its position within the directory structure to be changed.
- **Traverse the file system**. We may wish to access every directory and every file within a directory structure.

6.10 Common schemes for defining the logical structure of a directory

1. Single-Level Directory

- In single level directory structure, all files are contained in the same directory, which is easy to support and understand (Figure 6.7).

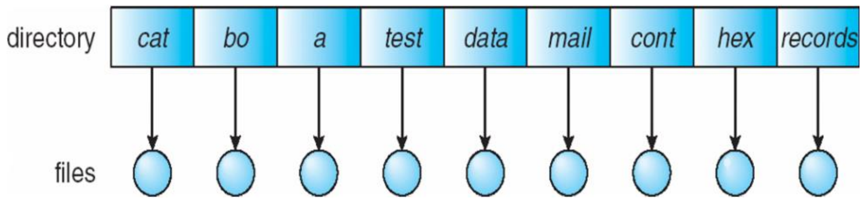


Figure 6.7 Single-level directory structure

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

2. Two-Level Directory

- In the two-level directory structure, each user has his own user file directory (LTD). The UFDs have similar structures, but each lists only the files of a single user. When a user job starts or a user logs in, the system's master file directory (MFD) is searched. The MFD is indexed by user name or account number, and each entry points to the UFD for that user (Figure 6.8).
- A two-level directory can be thought of as a tree, or an inverted tree, of height 2.

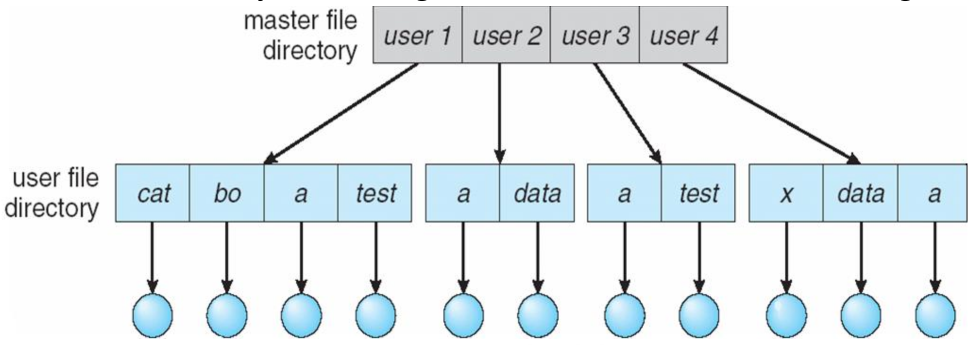


Figure 6.8 Two-level directory structure.

- The user directories themselves must be created and deleted as necessary.
- Although the two-level directory structure solves the name-collision problem, it still has disadvantages. This structure effectively isolates one user from another. Isolation is an advantage when the users are completely independent but is a disadvantage when the users *want* to cooperate on some task and to access one another's files.
- Every file in the system has a path name. To name a file uniquely, a user must know the path name of the file desired. For example, if user A wishes to access her own test file named *test*, she can simply refer to *test.* To access the file named *test* of user B (with directory-entry name *userb),* however, she might have to refer to */userb/test*

3. Tree-Structured Directories

- Once we have seen how to view a two-level directory as a two-level tree, the natural generalization is to extend the directory structure to a tree of arbitrary height (Figure 6.9).
- This generalization allows users to create their own subdirectories and to organize their files accordingly.

- A tree is the most common directory structure. The tree has a root directory, and every file in the system has a unique path name.

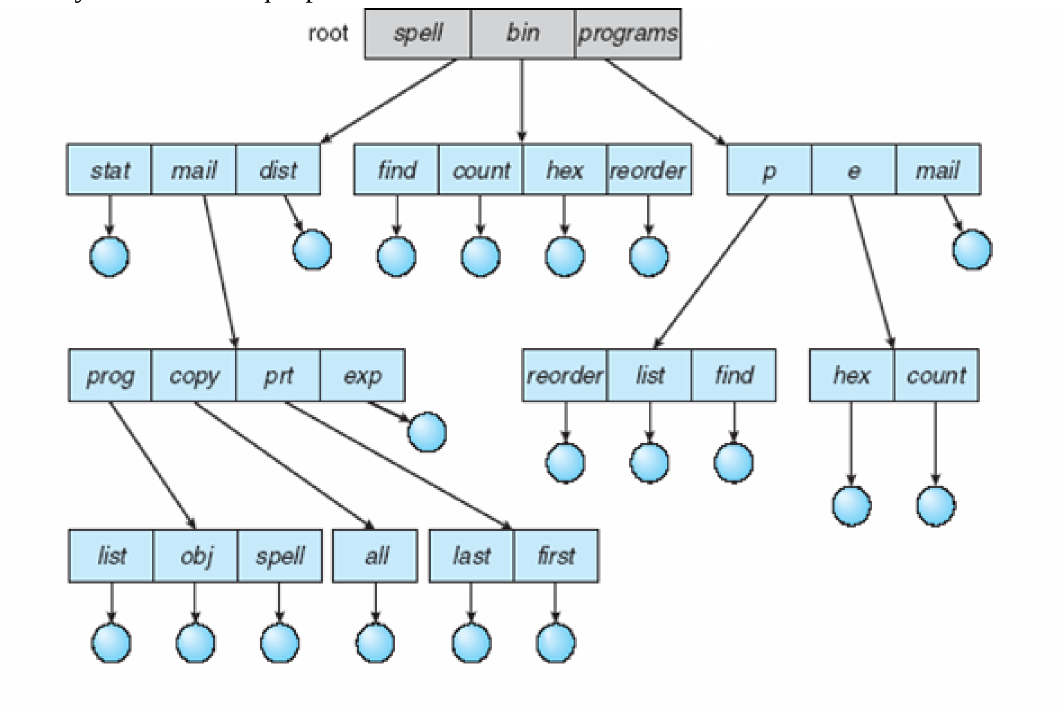


Figure 6.9 Tree-structured directory structure

- A directory (or subdirectory) contains a set of files or subdirectories. A directory is simply another file, but it is treated in a special way. All directories have the same internal format. One bit in each directory entry defines the entry as a file (0) or as a subdirectory (1). Special system calls are used to create and delete directories.
- Each process has a current directory. The **current directory** should contain most of the files that are of current interest to the process. When reference is made to a file, the current directory is searched. If a file is needed that is not in the current directory, then the user usually must either specify a path name or change the current directory to be the directory holding that file.
- The initial current directory of the login shell of a user is designated when the user job starts or the user logs in.
- Path names can be of two types: ***absolute*** and ***relative***.
- An absolute path **name** begins at the root and follows a path down to the specified file, giving the directory names on the path.
- A relative path **name** defines a path from the current directory.
- For example, in the tree-structured file system of Figure 6.9, if the current directory is *root/spell/mail*, then the relative path name *prt/first* refers to the same file as does the absolute path name *root/spell/mail/prt/firs*

4. Acyclic-Graph Directories

- A tree structure prohibits the sharing of files or directories. An **acyclic graph** —that is, a graph with no cycles—allows directories to share subdirectories and files (Figure 6.10). The *same* file or subdirectory may be in two different directories.

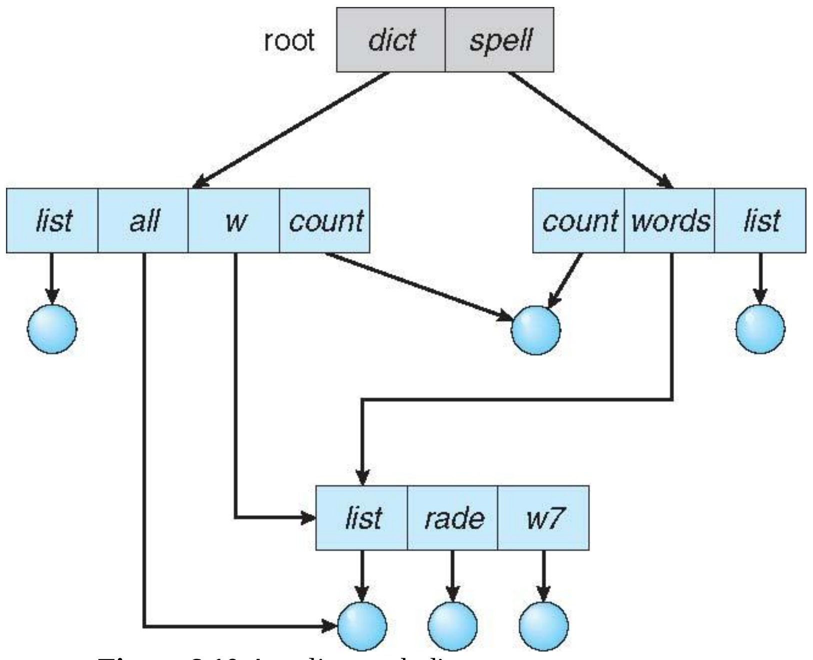


Figure 6.10 Acyclic-graph directory structure

- When people are working as a team, all the files they want to share can be put into one directory. The UFD of each team member will contain this directory of shared files as a subdirectory.
- Shared files and subdirectories can be implemented in several ways.
- A common way, exemplified by many of the UNIX systems, is to create a new directory entry called a **link**.
- Another common approach to implementing shared files is simply to **duplicate** all information about them in both sharing directories. Thus, both entries are identical and equal. A major problem with duplicate directory entries is maintaining consistency when a file is modified.
- Several problems must be considered carefully. When can the space allocated to a shared file be deallocated and reused? One possibility is to remove the file whenever anyone deletes it, but this action may leave dangling pointers to the **now-nonexistent file**. In a system where sharing is implemented by symbolic links, this situation is somewhat easier to handle. The deletion of a link need not affect the original file; only the link is removed.

5. General Graph Directory (with cycles)

- If we start with a two-level directory and allow users to create subdirectories, a tree-structured directory results. It should be fairly easy to see that simply adding new files and subdirectories to an existing tree-structured directory preserves the tree-structured nature. However, when we add links to an existing tree-structured directory, the tree structure is destroyed, resulting in a simple graph structure (Figure 6.11).

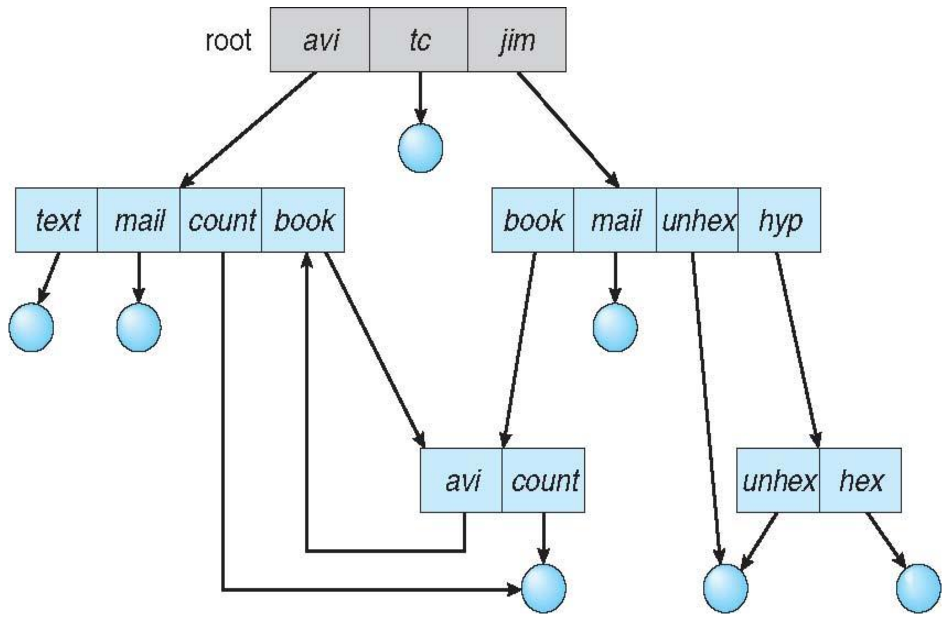


Figure 6.11 General graph directory

- The primary advantage of an acyclic graph is the relative simplicity of the algorithms to traverse the graph and to determine when there are no more references to a file.
- If cycles are allowed to exist in the directory, we likewise want to avoid searching any component twice, for reasons of correctness as well as performance.
- A similar problem exists when we are trying to determine when a file can be deleted. With acyclic-graph directory structures, a value of 0 in the reference count means that there are no more references to the file or directory, and the file can be deleted. However, when cycles exist, the reference count may not be 0 even when it is no longer possible to refer to a directory or file. In this case, we generally need to use a garbage-collection scheme to determine when the last reference has been deleted and the disk space can be reallocated.

6.11 File System Mounting

- As a file must be *opened* before it is used, a file system must be ***mounted*** before it can be available to processes on the system.
- First, the operating system is given the name of the device and the **mount point**—the location within the file structure where the file system is to be attached.
- Next, the operating system verifies that the device contains a valid file system. It does so by asking the device driver to read the device directory and verifying that the directory has the expected format.
- Finally, the operating system notes in its directory structure that a file system is mounted at the specified mount point. This scheme enables the operating system to traverse its directory structure, switching among file systems as appropriate.
- To illustrate file mounting, consider the file system depicted in Figure 6.12, where the triangles represent subtrees of directories that are of interest. Figure 6.12(a) shows an existing file system, while Figure 6.12(b) shows an unmounted volume residing on */device/dsk*. At this point, only the files on the existing file system can be accessed. Figure 6.13 shows the effects of mounting the volume residing on */device/dsk* over */users*. If the volume is unmounted, the file system is restored to the situation depicted in Figure 6.12.

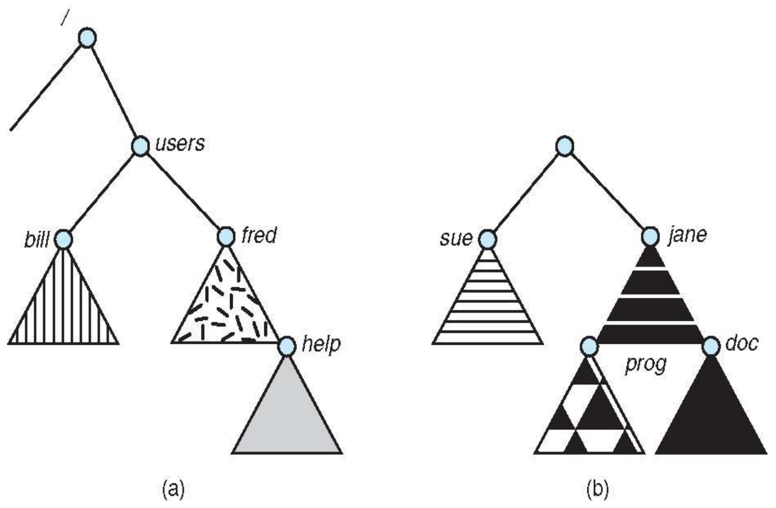


Figure 6.12 File system, (a) Existing system, (b) Unmounted volume.

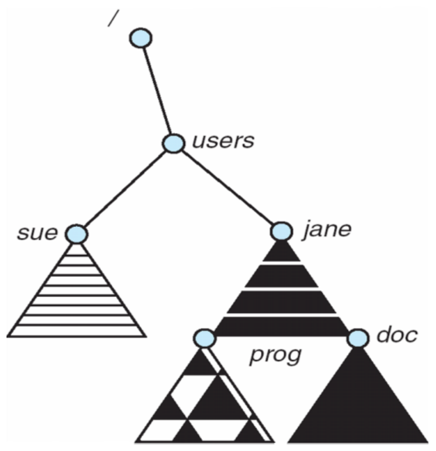


Figure 6.13 Mount point.

6.12 File Sharing

Multiple Users

- When an operating system accommodates multiple users, the issues of file sharing, file naming, and file protection become preeminent.
- Given a directory structure that allows files to be shared by users, the system must mediate the file sharing. The system can either allow a user to access the files of other users by default or require that a user specifically grant access to the files.
- To implement sharing and protection, the system must maintain more file and directory attributes than are needed on a single-user system.
- One approach to implement this is where, systems use the concepts of file (or directory) *owner* (or *user)* and *group*. The owner is the user who can change attributes and grant access and who has the most control over the file. The group attribute defines a subset of users who can share access to the file.

Remote File Systems (FILE SHARING)

- Through the evolution of network and file technology, remote file-sharing methods have changed. The first implemented method involves manually transferring files between machines via programs like **ftp (client server model)**.

1. **Distributed file system (DFS)** in which remote directories are visible from a local machine.

- Remote file systems allow a computer to mount one or more file systems from one or more remote machines. In this case, the machine containing the files is the *server,* and the machine seeking access to the files is the *client*. Generally, the server declares that a resource is available to clients and specifies exactly which files and exactly which clients. A server can serve multiple clients, and a client can use multiple servers, depending on the implementation details of a given client-server facility. Client identification is more difficult.
- A client can be specified by a network name or other identifier, such as an *IP address,* but these can be spoofed, or imitated. More secure solutions include secure authentication of the client via encrypted keys.

2. **Network file system (NFS),** authentication takes place via the client networking information, by default. In this scheme, the user's IDs on the client and server must match. If they do not, the server will be unable to determine access rights to files.

- Once the remote file system is mounted, file operation requests are sent on behalf of the user across the network to the server via the DFS protocol.
- Typically, a file-open request is sent along with the ID of the requesting user. The server then applies the standard access checks to determine if the user has credentials to access the file in the mode requested.
- The request is either allowed or denied. If it is allowed, a file handle is returned to the client application, and the application then can perform read, write, and other operations on the file. The client closes the file when access is completed.

6.13 Failure Modes

- Local file systems can fail for a variety of reasons, including failure of the disk containing the file system, corruption of the directory structure, disk-controller failure, cable failure, and host-adapter failure. User or systems-administrator failure can also cause files to be lost or entire directories or volumes to be deleted.

- Remote file systems have even more failure modes. Because of the complexity of network systems and the required interactions between remote machines, many more problems can interfere with the proper operation of remote file systems.
- In the case of networks, the network can be interrupted between two hosts. Such interruptions can result from hardware failure, poor hardware configuration, or networking implementation issues. Although some networks have built-in resiliency, including multiple paths between hosts, many do not. Any single failure can thus interrupt the flow of DFS commands.
- If both server and client maintain knowledge of their current activities and open files, then they can seamlessly recover from a failure. In the situation where the server crashes but must recognize that it has remotely mounted exported file systems and opened files, NFS takes a simple approach, implementing a stateless DFS.

6.14 Consistency Semantics

- Consistency semantics represent an important criterion for evaluating any file system that supports file sharing.
- These semantics specify how multiple users of a system are to access a shared file simultaneously.
- In particular, they specify when modifications of data by one user will be observable by other users. These semantics are typically implemented as code with the file system.

1. UNIX Semantics

The UNIX file system uses the following consistency semantics:

- Writes to an open file by a user are visible immediately to other users that have this file open.
- One mode of sharing allows users to share the pointer of current location into the file. Thus, the advancing of the pointer by one user affects all sharing users. Here, a file has a single image that interleaves all accesses, regardless of their origin.

In the UNIX semantics, a file is associated with a single physical image that is accessed as an exclusive resource.

2. Session Semantics

The Andrew file system (AFS) uses the following consistency semantics:

- Writes to an open file by a user are not visible immediately to other users that have the same file open.
- Once a file is closed, the changes made to it are visible only in sessions starting later. Already open instances of the file do not reflect these changes.

According to these semantics, a file may be associated temporarily with several (possibly different) images at the same time. Consequently, multiple users are allowed to perform both read and write accesses concurrently on their images of the file, without delay.

3. Immutable-Shared-Files Semantics

- A unique approach is that of **immutable shared files.** Once a file is declared as *shared* by its creator, it cannot be modified.
- An immutable file has two key properties: Its name may not be reused, and its contents may not be altered. Thus, the name of an immutable file signifies that the contents of the file are fixed.

11. File-System Structure

11.1 Introduction

- Disks provide the bulk of secondary storage on which a file system is maintained. They have two characteristics that make them a convenient medium for storing multiple files:
  - A disk can be rewritten in place; it is possible to read a block from the disk, modify the block, and write it back into the same place.
  - A disk can access directly any given block of information it contains.
- The file system itself is generally composed of many different levels. The structure shown in Figure 11.1

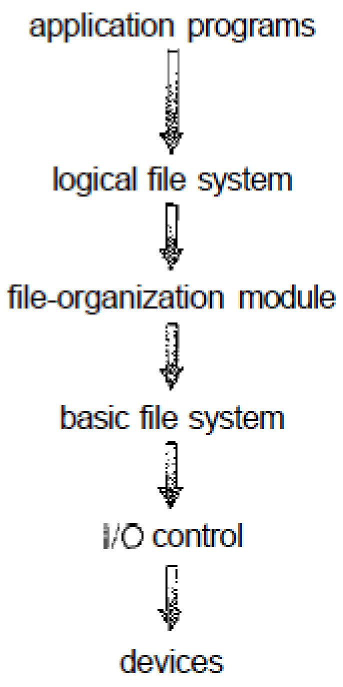


Figure 11.1 Layered file system.

1. The lowest level, the ***I/O control***, consists of **device drivers** and interrupts handlers to transfer information between the main memory and the disk system. A device driver can be thought of as a translator.
2. The **basic file system** needs only to issue generic commands to the appropriate device driver to read and write physical blocks on the disk. Each physical block is identified by its numeric disk address.
3. The **file-organization module** knows about files and their logical blocks, as well as physical blocks. By knowing the type of file allocation used and the location of the file, the file-organization module can translate logical block addresses to physical block addresses for the basic file system to transfer.

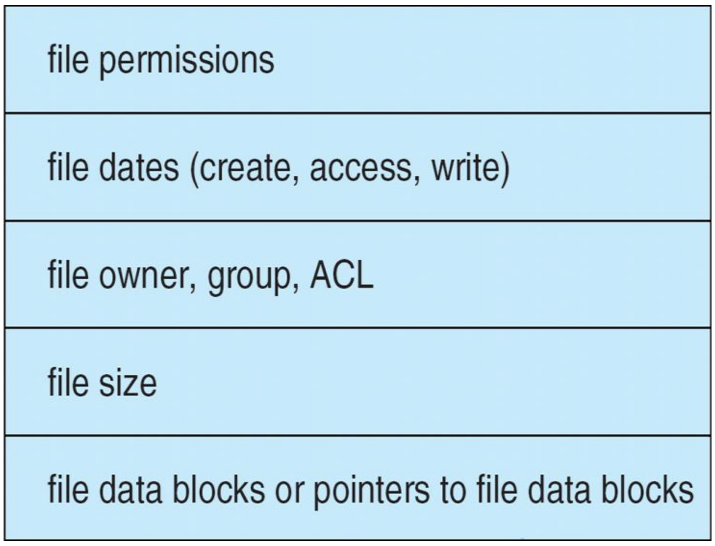
4. The **logical file system** manages metadata information. Metadata includes all of the file-system structure except the actual *data*. The logical file system is also responsible for protection and security.
5. Devices are the hardware resources available with the system such as disk, tape drives, memory etc.
6. Application programs are the user programs which needs to be stored in memory.

11.2 File System Implementation

File system is implemented on the disk and the memory. The implementation of the file system varies according to the OS and the file system, but there are some general principles. If the file system is implemented on the disk it contains the following information:

1. **Boot Control Block**:- can contain information needed by the system to boot an OS from that partition. If the disk has no OS, this block is empty. It is the first block of the partition. In UFS → boot block, In NTFS → partition boot sector.
2. **Partition (Partition) control Block**:- contains partition details such as the number of blocks in partition, size of the blocks, number of free blocks, free block pointer, free FCB count and FCB pointers. In NTFS→master file tables, In UFS→super block.
3. **Directory structure** is used to organize the files.
4. An FCB contains many of the files details, including file permissions, ownership, size, location of the data blocks. In UFS→inode, In NTFS this information is actually stored within master file table.

To create a new file, an application program calls the logical file system. The logical file system knows the format of the directory structures. To create a new file, it allocates a new FCB. The system then reads the appropriate directory into memory, updates it with the new file name and FCB, and writes it back to the disk. A typical FCB is shown in Figure 11.2.


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

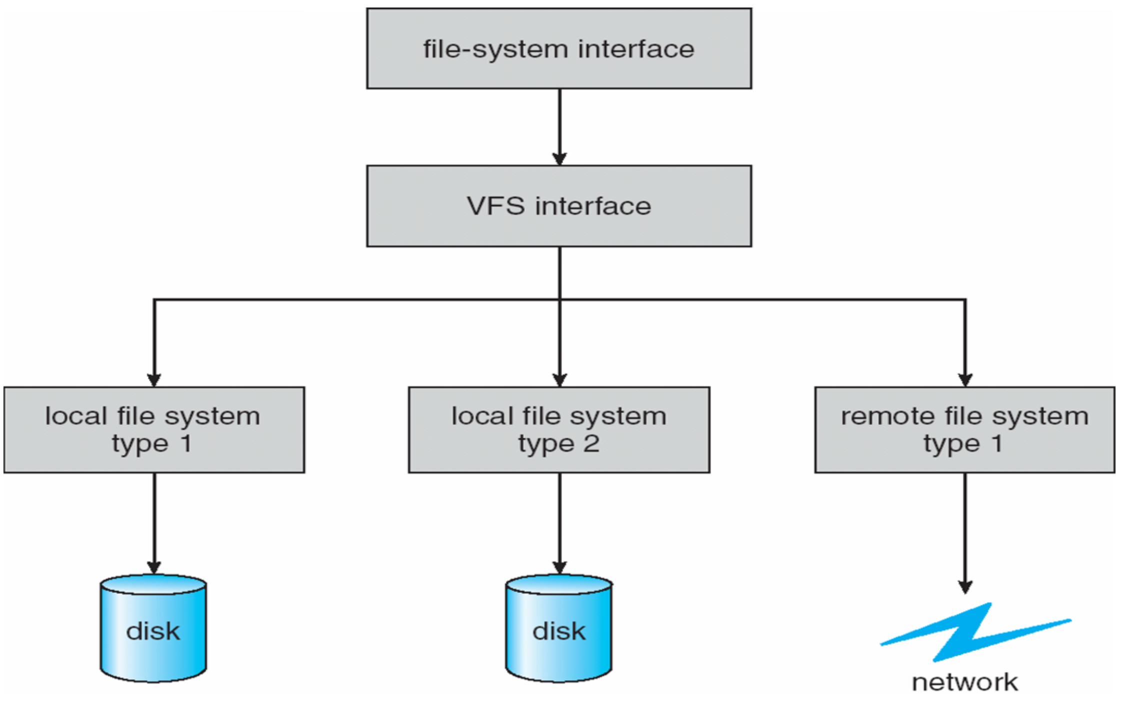
11.2 A Typical File Control Block (FCB)

11.2.2 Partitions and Mounting

- The layout of a disk can have many variations, depending on the operating system.
- A disk can be sliced into multiple partitions, or a volume can span multiple partitions on multiple disks.
- Each partition can be either "raw," containing no file system, or "cooked;' containing a file system.
- Raw disk is used where no file system is appropriate.
- UNIX swap space can use a raw partition.
- Boot information can be stored in a separate partition.
- The root **partition,** which contains the operating-system kernel and sometimes other system files, is mounted at boot time.
- The operating system notes in its in-memory **mount table** structure that a file system is mounted, along with the type of the file system.

11.2.3 Virtual File Systems

The file-system implementation consists of three major layers, as depicted schematically in Figure 11.4.



11.4 Schematic View of Virtual File System

- The first layer is the file-system interface, based on the open( ), read( ), write( ), and close( ) calls and on file descriptors.
- The second layer is called the virtual file system (VFS) layer; it serves two important functions:

- It separates file-system-generic operations from their implementation by defining a clean VFS interface.
- The VFS provides a mechanism for uniquely representing a file throughout a network. The VFS is based on a file-representation structure, called a vnode, that contains a numerical designator for a network-wide unique file.

The four main object types defined by the Linux VFS are:

- The **inode object,** which represents an individual file
- The **file object,** which represents an open file
- The **superblock object,** which represents an entire file system
- 8 The **dentry object,** which represents an individual directory entry

For each of these four object types, the VFS defines a set of operations that must be implemented.

An abbreviated API for some of the operations for the file object include:

- int open(. . .) —Open a file.
- ssize_t read(. . .)—Read from a file.
- ssize_t write (. . .) —Write to a file.
- int mmap(. . .) — Memory-map a file.

11.3 Directory Implementation

Directory is implemented in two ways:

1. **Linear list:**
   - Linear list is a simplest method.
   - It uses a linear list of file names with pointers to the data blocks.
   - Linear list uses a linear search to find a particular entry.
   - Simple for programming but time consuming to execute.
   - For creating a new file, it searches the directory for the name whether same name already exists.
   - Linear search is the main disadvantage.
   - Directory implementation is used frequently and uses would notice a slow implementation of access to it.
2. **Hash table:**
   - Hash table decreases the directory search time.
   - Insertion and deletion are fairly straight forward.
   - Hash table takes the value computed from that file name.
   - Then it returns a pointer to the file name in the linear list.
   - Hash table uses fixed size.
   - The major difficulties with a hash table are its generally fixed size and the dependence of the hash function on that size.

11.4 Allocation Methods

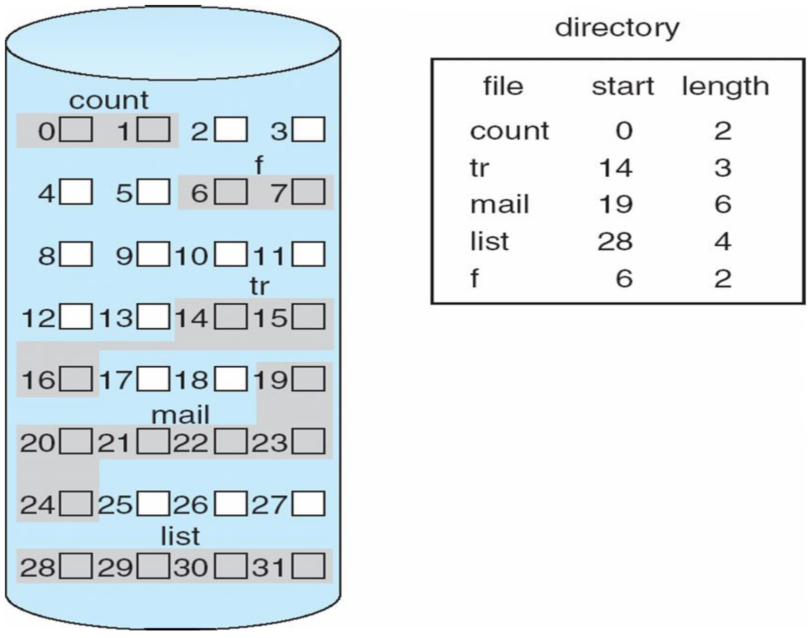
The space allocation strategy is closely related to the efficiency of the file accessing and of logical to physical mapping of disk addresses. A good space allocation strategy must take in to consideration several factors such as:

- Processing speed of sequential access to files, random access to files and allocation and de-allocation of blocks.
- Disk space utilization.
- Main memory requirement of a given algorithm.

Three major methods of allocating disk space are used.

1. Contiguous Allocation

- A single set of blocks is allocated to a file at the time of file creation. This is a pre-allocation strategy that uses portion of variable size. The file allocation table needs just a single entry for each file, showing the starting block and the length of the file. The figure 6.15 shows the contiguous allocation method. If the file is n blocks long and starts at location b, then it occupies blocks b, b+1, b+2…………….b+n
- The file allocation table entry for each file indicates the address of starting block and the length of the area allocated for this file. Contiguous allocation is the best from the point of view of individual sequential file. It is easy to retrieve a single block. Multiple blocks can be brought in one at a time to improve I/O performance for sequential processing. Sequential and direct access can be supported by contiguous allocation. Contiguous allocation algorithm suffers from external fragmentation. Depending on the amount of disk storage the external fragmentation can be a major or minor problem. Compaction is used to solve the problem of external fragmentation. The following figure shows the contiguous allocation of space after compaction. The original disk was then freed completely creating one large contiguous space. If the file is n blocks long and starts at location b, then it occupies blocks b, b+1, b+2…………….b+n
- The file allocation table entry for each file indicates the address of starting block and the length of the area allocated for this file. Contiguous allocation is the best from the point of view of individual sequential file. It is easy to retrieve a single block. Multiple blocks can be brought in one at a time to improve I/O performance for sequential processing. Sequential and direct access can be supported by contiguous allocation. Contiguous allocation algorithm suffers from external fragmentation.



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

Figure 6.15 Contiguous Allocation

Advantages:

- Both direct and sequential access is possible.
- Accessing a file is easy.
- It provides good performance.

Disadvantage:

- Pre-allocation is required, cannot be used in dynamic storage allocation.
- It suffers from external fragmentation.
- Difficult to find space for new file.

2. Linked Allocation

- It solves the problem of contiguous allocation.
- This allocation is on the basis of an individual block. Each block contains a pointer to the next block in the chain.
- The disk block can be scattered anywhere on the disk.
- The directory contains a pointer to the first and the last blocks of the file.
- The following figure shows the linked allocation. To create a new file, simply create a new entry in the directory.
- There is no external fragmentation since only one block is needed at a time.
- The size of a file need not be declared when it is created. A file can continue to grow as long as free blocks are available.

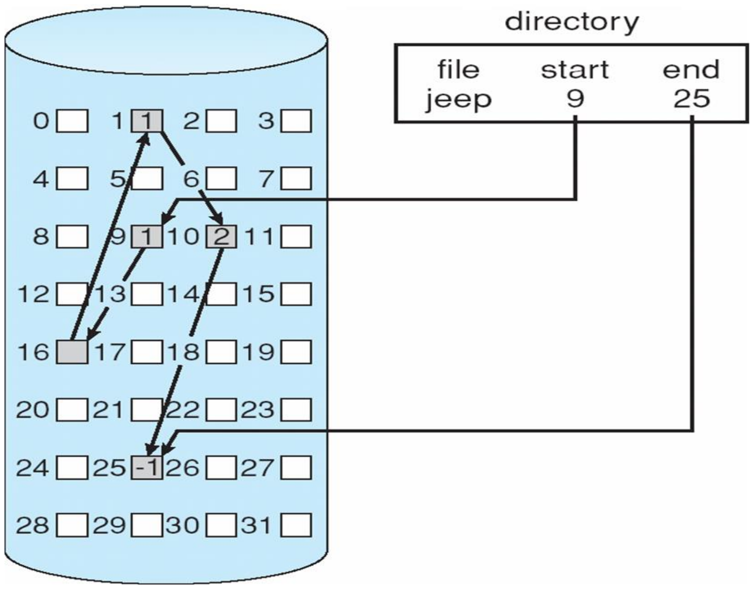


Figure 6.16 Linked Allocation

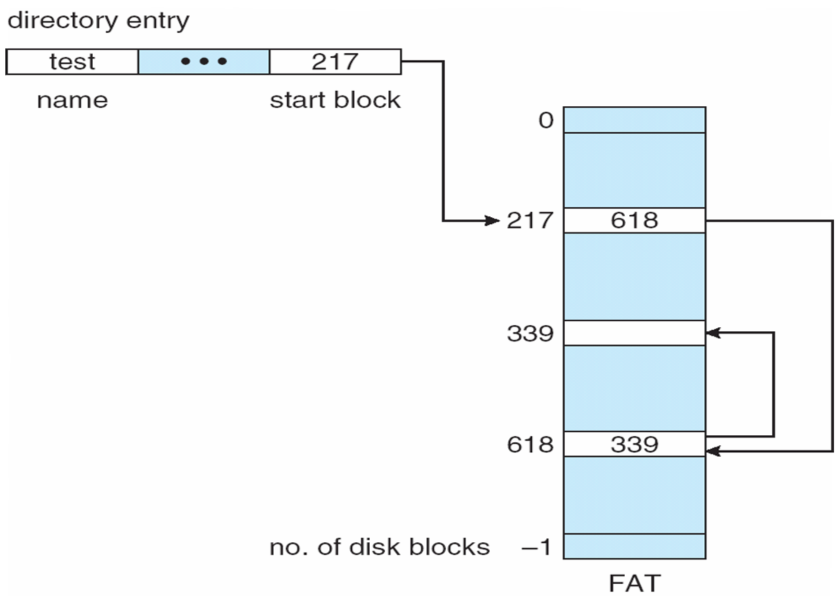


Figure 6.17 File Allocation Table (FAT)

Advantages:

- No external fragmentation.
- Compaction is never required.
- Pre-allocation is not required (support dynamic memory allocation)

Disadvantage:

- Files are accessed sequentially, cannot support direct access.
- Space required for pointers.
- Reliability is not good.

3. Indexed Allocation

- The file allocation table contains a separate one level index for each file. The index has one entry for each portion allocated to the file. The ith entry in the index block points to the ith block of the file.
- The following figure shows indexed allocation.
- The indexes are not stored as a part of file allocation table rather than the index is kept as a separate block and the entry in the file allocation table points to that block. Allocation can be made on either fixed size blocks or variable size blocks.
- When the file is created all pointers in the index block are set to nil.
- When an entry is made a block is obtained from free space manager.
- Allocation by fixed size blocks eliminates external fragmentation where as allocation by variable size blocks improves locality.
- Indexed allocation supports both direct access and sequential access to the file.

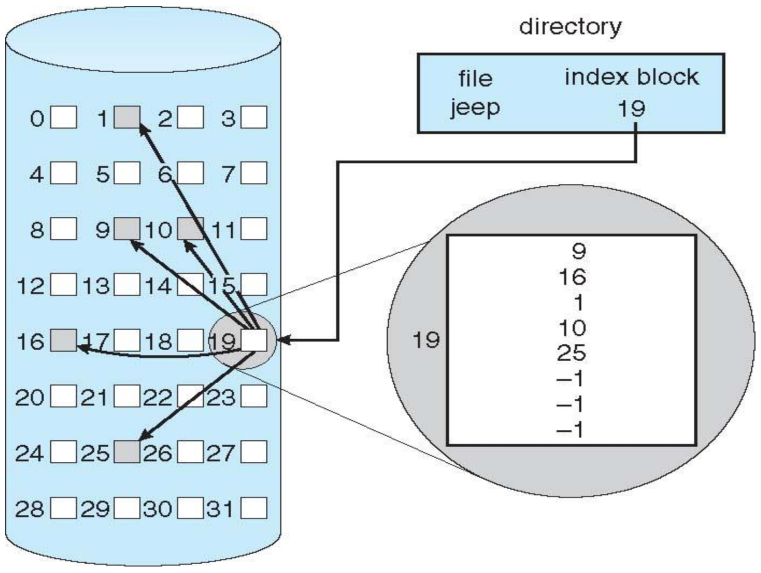


Figure 6.18 Indexed Allocation

Advantages:-

- Supports both sequential and direct access.
- No external fragmentation.
- Faster than other two methods.
- Supports fixed size and variable sized blocks.

Disadvantage:-

- Suffers from wasted space.
- Pointer overhead is generally greater

11.5 FREE SPACE MANAGEMENT

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

1. Bit Vector

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

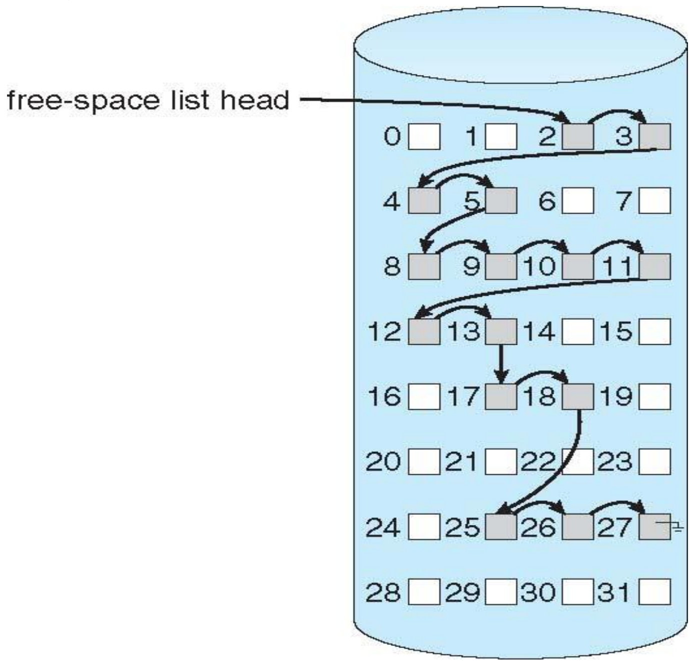
  001111001111110001100000011100000…

- The main advantage of this approach is its relative simplicity and its efficiency in finding the first free block or *n* consecutive free blocks on the disk. .

2. Linked List

- Another approach to free-space management is to link together all the free disk blocks, keeping a pointer to the first free block in a special location on the disk and caching it in memory.

- This first block contains a pointer to the next free disk block, and so on.
- We would keep a pointer to block 2 as the first free block. Block *2* would contain a pointer to block 3, which would point to block 4, which would point to block 5, which would point to block 8, and so on.
- This scheme is not efficient; to traverse the list, we must read each block, which requires substantial I/O time.



3. Grouping

- A modification of the free-list approach is to store the addresses of *n* free blocks in the first free block.
- The first n - 1 of these blocks are actually free. The last block contains the addresses of another *n* free blocks, and so on.
- The addresses of a large number of free blocks can now be found quickly, unlike the situation when the standard linked-list approach is used.

4. Counting

- Generally, several contiguous blocks may be allocated or freed simultaneously, particularly when space is allocated with the contiguous-allocation algorithm or through clustering.
- Thus, rather than keeping a list of *n* free disk addresses, we can keep the address of the first free block and the number *n* of free contiguous blocks that follow the first block.
- Each entry in the free-space list then consists of a disk address and a count. Although each entry requires more space than would a simple disk address, the overall list will be shorter, as long as the count is generally greater than 1.