Unit - III

Define Paging. Discuss different page replacement algorithms are there in Memory

Paging is a memory-management scheme that permits the physical address space of a process to be non-contiguous thereby avoiding fragmentation problems. It also solves the considerable problem of fitting memory chunks of varying sizes onto the backing store. The basic method for implementing paging involves breaking physical memory into fixed-sized blocks called frames and breaking logical memory into blocks of the same size called frames. When a process is to be executed, its pages are loaded into any available memory frames from their source (a file system or the backing store). The backing store is divided into fixed-sized blocks that are of the same size as the memory frames.

Page replacement algorithms:

Page replacement is a process of swapping out an existing page from the frame of a main memory and replacing it with the required page. Page replacement is required when- All the frames of main memory are already occupied. Thus, a page has to be replaced to create a room for the required page.

Different page replacement algorithms can be used to decide which page has to be swapped out. Some are:

- FIFO
- LRU
- Optimal Page Replacement

Consider a reference string: 7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1 and 3 frames.

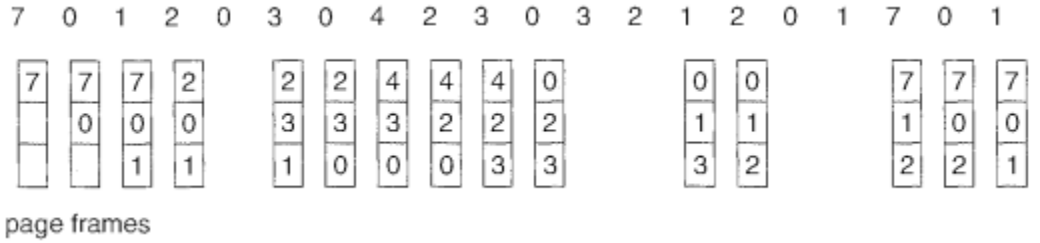
FIFO:

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. A FIFO queue can also be created to hold all pages in memory instead of recording times . We replace the page at the head of the queue. When a page is brought into memory, we insert it at the tail of the queue.

Consider a reference string: 7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1

For our example reference string, our three frames are initially empty. The first three references (7, 0, 1) cause page faults and are brought into these empty frames. The next reference (2) replaces page 7, because page 7 was brought in first. Since 0 is the next reference and 0 is already in memory, we have no fault for this reference. The first reference to 3 results in replacement of page 0, since it is now first in line. Because

of this replacement, the next reference, to 0, will fault. Page 1 is then replaced by page 0. This process continues as shown below

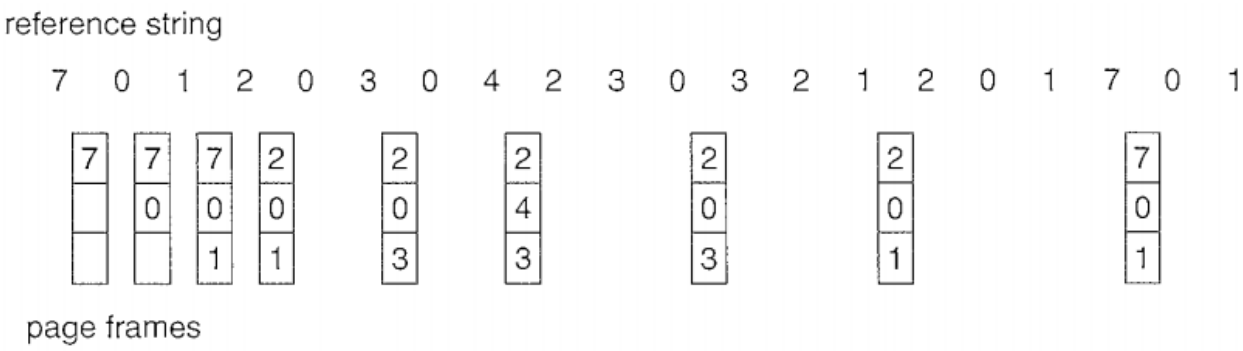


Optimal Page replacement:

Optimal Page replacement has the lowest page-fault rate of all algorithms and will never suffer from Belady's anomaly. Such an algorithm does exist and has been called OPT or MIN. It is simply this:

Replace the page that will not be used for the longest period of time.

Use of this page-replacement algorithm guarantees the lowest possible page-fault rate for a fixed number of frames. For example, on our sample reference string, the optimal page-replacement algorithm would yield nine page faults, as shown below.

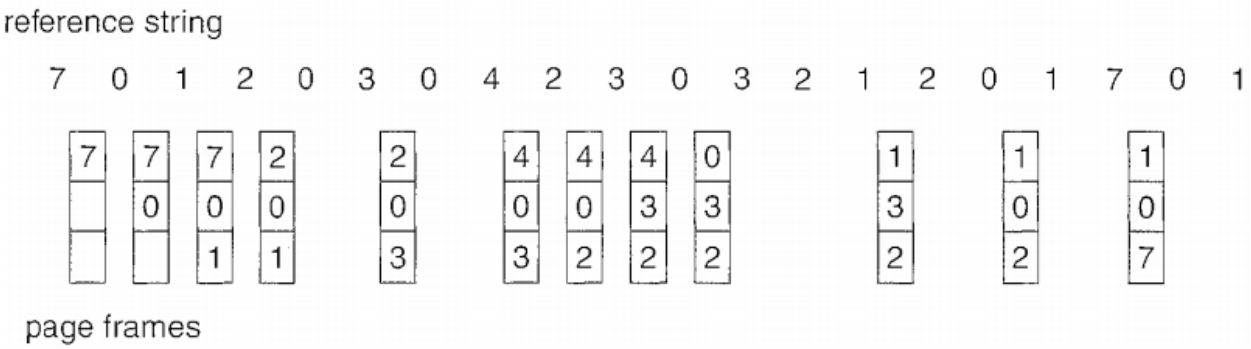


The first three references cause faults that fill the three empty frames. The reference to page 2 replaces page 7, because page 7 will not be used until reference 18, whereas page 0 will be used at 5, and page 1 at 14. The reference to page 3 replaces page 1, as page 1 will be the last of the three pages in memory to be referenced again. With only nine page faults, optimal replacement is much better than a FIFO algorithm, which results in fifteen faults. (If we ignore the first three, which all algorithms must suffer, then optimal replacement is twice as good as FIFO replacement.) In fact, no replacement algorithm can process this reference string in three frames with fewer than nine faults.

LRU:

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. This approach is called Least Recently Used page replacement. LRU replacement associates with each

page the time of that page's last use. When a page must be replaced, LRU chooses the page that has not been used for the longest period of time. We can think of this strategy as the optimal page-replacement algorithm looking backward in time, rather than forward.
The result of applying LRU replacement to our example reference string is shown below



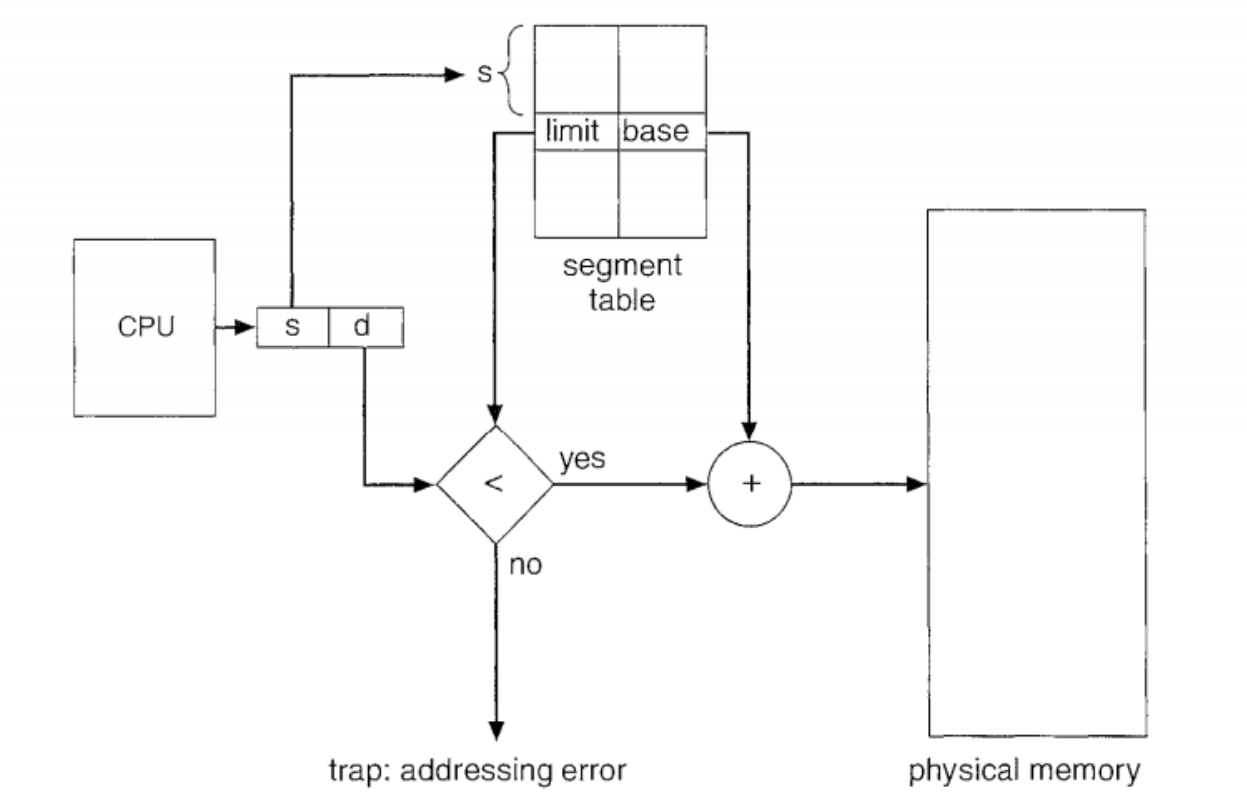
Notice that the first five faults are the same as those for optimal replacement. When the reference to page 4 occurs, however, LRU replacement sees that, of the three frames in memory, page 2 was used least recently. Thus, the LRU algorithm replaces page 2, not knowing that page 2 is about to be used.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

Segmentation

An important aspect of memory management that became unavoidable with paging is the separation of the user's view of memory from the actual physical memory. Users prefer to view memory as a collection of variable-sized segments, with no necessary ordering among segments. Segmentation is memory management that supports this user view of memory.
A logical address space is a collection of segments. Each segment has a name and a length. The addresses specify both the segment name and the offset within the segment. The user therefore specifies each address by two quantities: a segment name and an offset. For simplicity of implementation, segments are numbered and are referred to by a segment number, rather than by a segment name.

Although the user can now refer to objects in the program by a two-dimensional address(segment number, offset), the actual physical memory is still, of course, a one-dimensional sequence of bytes. Thus, we must define an implementation to map two-dimensional user-defined addresses into one-dimensional physical addresses. This mapping is effected by a segment table. 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.



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

Deadlock Characteristics or Characterization:

In a deadlock, processes never finish executing, and system resources are tied up, preventing other jobs from starting. A deadlock situation can arise if the following four conditions hold simultaneously in a system:

1. Mutual exclusion.

At least one resource must be held in a non-sharable mode; that is, only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.

2. Hold and wait.

A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.

3. No preemption.

Resources cannot be preempted; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task.

4. Circular wait.

A set { P0 , Pl, ... , P11 } of waiting processes must exist such that Po is waiting for a resource held by P1, P1 is waiting for a resource held by P2, ... , Pn-1 is waiting for a resource held by P,v and P11 is waiting for a resource held by Po.

Banker's algorithm(Incomplete):

When a new process enters the system, it must declare the maximum number of instances of each resource type that it may need. This number may not exceed the total number of resources in the system. When a user requests a set of resources, the system must determine whether the allocation of these resources will leave the system in a safe state. If it will, the resources are allocated; otherwise, the process must wait until some other process releases enough resources.

We need the following data structures, where n is the number of processes in the system and m is the number of resource types:

1. Available: A vector of length m indicates the number of available resources of each type. If Available[j] equals k, then k instances of resource type Rj are available.
2. Max: An n x m matrix defines the maximum demand of each process. If Max[i] [j] equals k, then process P; may request at most k instances of resource type Rj.
3. Allocation: An n x m matrix defines the number of resources of each type currently allocated to each process. If Allocation[i][j] equals k, then process P; is currently allocated k instances of resource type Rj.
4. Need: An n x m matrix indicates the remaining resource need of each process. If Need[i][j] equals k, then process P; may need k more instances of resource type Rj to complete its task. Note that Need[i][j] equals Max[i][j] - Allocation [i][j].

Note: Write Safety algorithm and Resource Request algorithm from the textbook without changes for complete Banker's algorithm.

Unit - IV

File Concept

A file is a named collection of related information that is recorded on secondary storage. From a user's perspective, a file is the smallest allotment of logical secondary storage; that is, data cannot be written to secondary storage unless they are within a file. Commonly, files represent programs (both source and object forms) and data. Data files may be numeric, alphabetic, alphanumeric, or binary. Files may be free form, such as text files, or may be formatted rigidly. In general, a file is a sequence of bits, bytes, lines, or records, the meaning of which is defined by the file's creator and user.

File Operations:

The operating system can provide system calls to create, write, read, reposition, delete, and truncate files.

1. 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.
2. Writing a file:
   To write a file, we make a system call specifying both the name of the file and the information to be written to the file. Given the name of the file, the system searches the directory to find the file's location. The system must keep a write pointer to the location in the file where the next write is to take place. The write pointer must be updated whenever a write occurs.
3. Reading a file:
   To read from a file, we use a system call that specifies the name of the file and where (in memory) the next block of the file should be put. Again, the directory is searched for the associated entry, and the system needs to keep a read pointer to the location in the file where the next read is to take place. Once the read has taken place, the read pointer is updated. Because a process is usually either reading from or writing to a file, the current operation location can be kept as a per-process . Both the read and write operations use this same pointer, saving space and reducing system complexity.
4. 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/0. This file operation is also known as a file seek.
5. 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.

6. 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.

File Access methods:

The information in the file can be accessed in several ways.

Sequential Access:

The simplest access method is 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. Similarly, the write operation-write next-appends to the end of the file and advances to the end of the newly written material (the new end of file). Such a file can be reset to the beginning.

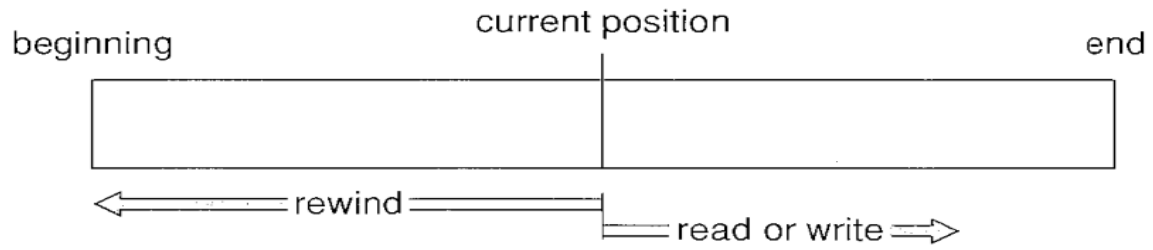


Figure 10.3 Sequential-access file.

Direct Access:

A 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. Databases are often of this type. When a

query concerning a particular subject arrives, we compute which block contains the answer and then read that block directly to provide the desired information.

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 effect a read n, we would position to n and then read next.

The block number by the user to the operating system is normally a relative block number. A relative block number is an index relative to the begirm.ing of the file. Thus, the first relative block of the file is 0, the next is 1, and so on, even though the absolute disk address may be 14703 for the first block and 3192 for the second.

File System mounting:

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

The mount procedure is as follows:

The operating system is given the name of the device and the *mountpoint* - the location within the file structure where the file system is to be attached. Some operating systems require that a file system type be provided, while others inspect the structures of the device and determine the type of file system. Typically, a mount point is an empty directory. For instance, on a UNIX system, a file system containing a user's home directories might be mounted as /home; then, to access the directory structure within that file system, we could precede the directory names with /home, as in /home/jane. Mounting that file system under /users would result in the path name /users/jane, which we could use to reach the same directory. 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.

To illustrate file mounting, consider the file system depicted below, where the triangles represent subtrees of directories that are of interest. Figure (a) shows an existing file system, while Figure (b) shows an unmounted volume residing on /device/dsk. At this

point, only the files on the existing file system can be accessed.

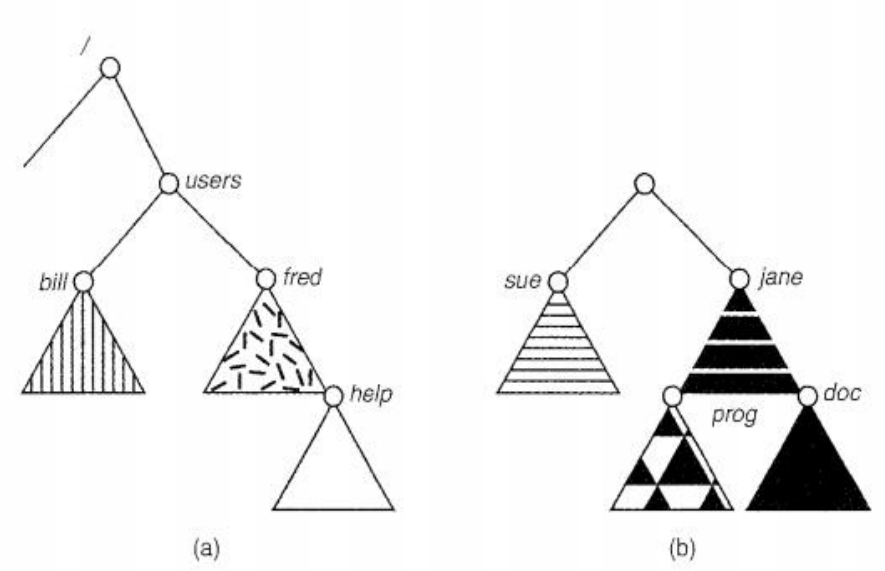


Figure 10.13 File system. (a) Existing system. (b) Unmounted volume.

Allocation Methods:

Three major methods of allocating disk space are in wide use: contiguous, linked, and indexed. Each method has advantages and disadvantages.

Contiguous Allocation:

Contiguous allocation requires that each file occupies a set of contiguous(continuous) blocks on disk. Disk addresses define a linear ordering on the disk. With this ordering, assuming that only one job is accessil1.g the disk, accessing block b + 1 after block b normally requires no head movement. When head movement is needed (from the last sector of one cylil1.der to the first sector of the next cylinder), the head needs to only move from one track to the next. Thus, the number of disk seeks required for accessing contiguously allocated files is minimal, as is seek time when a seek is finally needed.

Contiguous allocation of a file is defined by the disk address and length of the first block. If the file is n blocks long and starts at location b, then it occupies blocks b, b + 1, b + 2, ... , b + n - 1. The directory entry for each file indicates the address of the

starting block and the length of the area allocated for this file

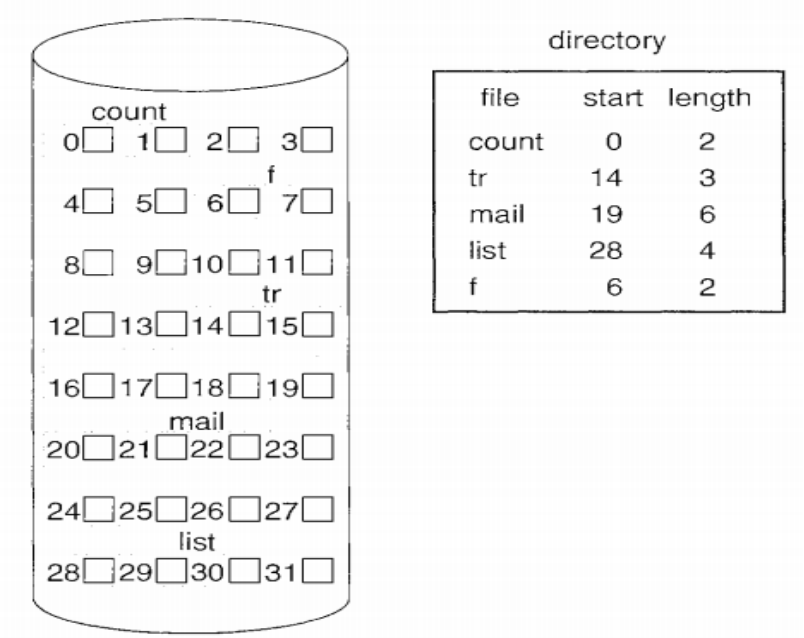


Figure 11.5 Contiguous allocation of disk space.

Contiguous allocation has some problems, however.
One difficulty is finding space for a new file. The system chosen to manage free space determines how this task is accomplished. All these algorithms suffer from the problem of *external fragmentation*. As files are allocated and deleted, the free disk space is broken into pieces.
Another problem with contiguous allocation is determining how much space is needed for a file. When the file is created, the total amount of space it will need must be found and allocated.

Linked Allocation:

Linked Allocation solves all problems of contiguous allocation. With linked allocation, each file is a linked list of disk blocks; the disk blocks may be scattered anywhere on the disk. The directory contains a pointer to the first and last blocks of the file. For example, a file of five blocks might start at block 9 and continue at block 16, then block 1, then block 10, and finally block 25. Each block contains a pointer to the next block.

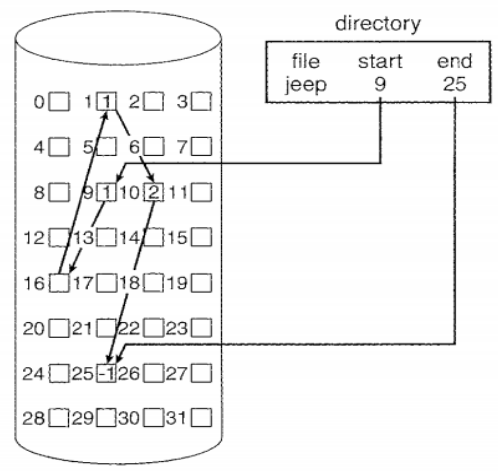


Figure 11.6 Linked allocation of disk space.

These pointers are not made available to the user. Thus, if each block is 512 bytes in size, and a disk address (the poileter) requires 4 bytes, then the user sees blocks of 508 bytes.

To create a new file, we simply create a new entry ile the directory. With linked allocation, each directory entry has a pointer to the first disk block of the file. This pointer is initialized to nil (the end-of-list pointer value) to signify an empty file. The size field is also set to 0. A write to the file causes the free-space management system to find a free block, and this new block is written to and is linked to the end of the file. To read a file, we simply read blocks by following the pointers from block to block. There is no external fragmentation with linked allocation, and any free block on the free-space list can be used to satisfy a request. The size of a file need not be declared when that file is created. A file can continue to grow as long as free blocks are available. Consequently, it is never necessary to compact disk space.

Linked allocation does have disadvantages, however. The major problem is that it can be used effectively only for sequential-access files. To find the ith block of a file, we must start at the beginning of that file and follow the pointers until we get to the ith block.

Another disadvantage is the space required for the pointers. If a pointer requires 4 bytes out of a 512-byte block, then 0.78 percent of the disk is being used for pointers, rather than for information.

Indexed Allocation:

Linked allocation cannot support efficient direct access, since the pointers to the blocks are scattered with the blocks themselves all over the disk and must be retrieved in order. Indexed allocation solves this problem by bringing all the pointers together into one location: the index block.

Each file has its own index block, which is an array of disk-block addresses. The i th entry in the index block points to the i 111 block of the file. The directory contains the address of the index block. To find and read the i block, we use the pointer in the ith index-block entry.

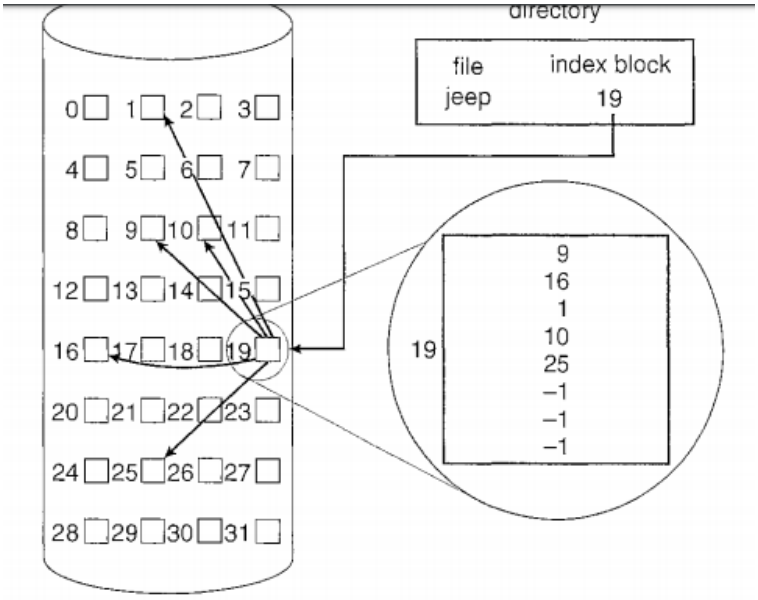


Figure 11.8 Indexed allocation of disk space.

With indexed allocation, an entire index block must be allocated, even if only one or two pointers will be non-nil. This point raises the question of how large the index block should be. Mechanisms for this purpose include the following:

1. Linked scheme: An index block is normally one disk block. Thus, it can be read and written directly by itself. To allow for large files, we can link together several index blocks. For example, an index block might contain a small header giving the name of the file and a set of the first 100 disk-block addresses. The next address (the last word in the index block) is nil (for a small file) or is a pointer to another index block (for a large file).
2. Multilevel index: A variant of linked representation uses a first-level index block to point to a set of second-level index blocks, which in turn point to the file blocks. To access a block, the operating system uses the first-level index to find a second-level index block and then uses that block to find the desired data block. This approach could be continued to a third or fourth level, depending on the desired maximum file size.
3. Combined scheme: Another alternative, used in the UFS, is to keep the first, say, 15 pointers of the index block in the file's inode. The first 12 of these pointers point to direct blocks; that is, they contain addresses of blocks that contain data

of the file. Thus, the data for small files (of no more than 12 blocks) do not need a separate index block. If the block size is 4 KB, then up to 48 KB of data can be accessed directly. The next three pointers point to indirect blocks. The first points to a single indirect block, which is an index block containing not data but the addresses of blocks that do contain data. The second points to a double indirect block, which contains the address of a block that contains the addresses of blocks that contain pointers to the actual data blocks. The last pointer contains the address of a triple indirect block.

Life Cycle of I/O request:

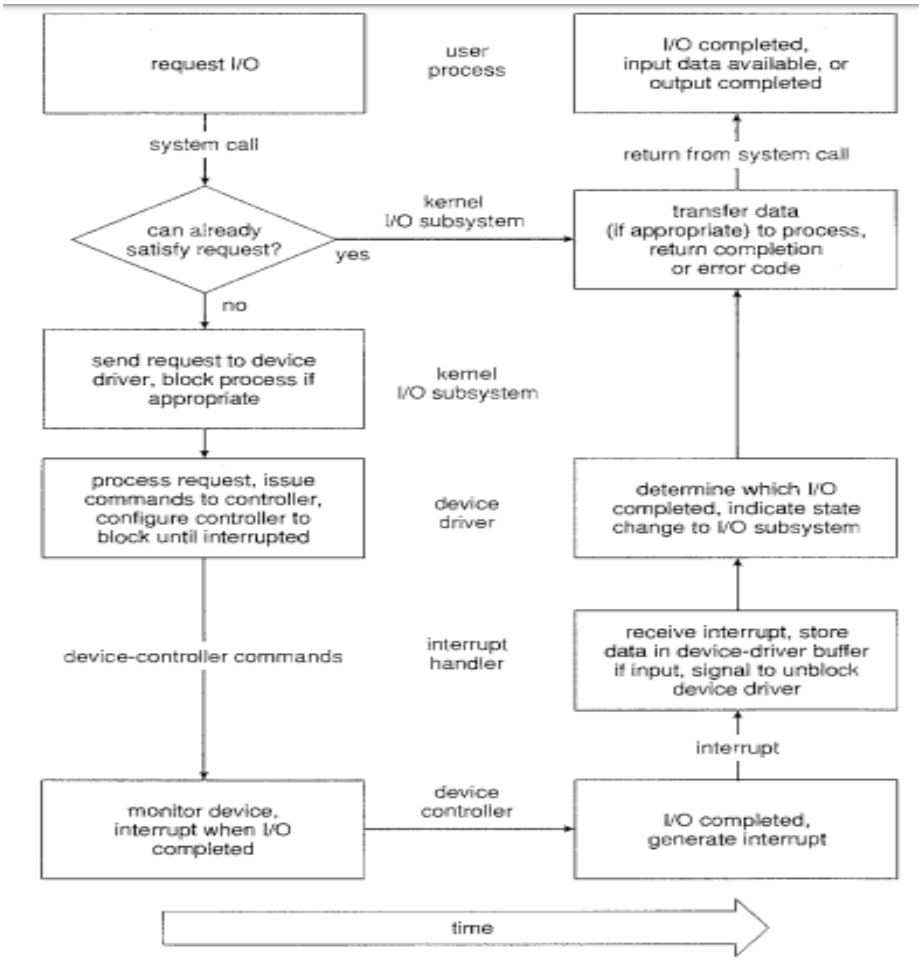


Figure 13.13 The life cycle of an I/O request.

1. A process issues a blocking read () system call to a file descriptor of a file that has been opened previously.
2. The system-call code in the kernel checks the parameters for correctness. In the case of input, if the data are already available irl the buffer cache, the data are returned to the process, and the I/O request is completed.

3. Otherwise, a physical I/0 must be performed. The process is removed from the run queue and is placed on the wait queue for the device, and the I/0 request is scheduled. Eventually, the I/0 subsystem sends the request to the device driver. Depending on the operating system, the request is sent via a subroutine call or an in-kernel message.
4. The device driver allocates kernel buffer space to receive the data and schedules the I/0. Eventually, the driver sends commands to the device controller by writing into the device-control registers.
5. The device controller operates the device hardware to perform the data transfer.
6. The driver may poll for status and data, or it may have set up a DMA transfer into kernel memory. We assume that the transfer is managed by a DMA controller, which generates an interrupt when the transfer completes.
7. The correct interrupt handler receives the interrupt via the interrupt vector table, stores any necessary data, signals the device driver, and returns from the interrupt.
8. The device driver receives the signal, determines which I/0 request has completed, determines the request's status, and signals the kernel I/0 subsystem that the request has been completed.
9. The kernel transfers data or return codes to the address space of the requesting process and moves the process from the wait queue back to the ready queue.
10. Moving the process to the ready queue unblocks the process. When the scheduler assigns the process to the CPU, the process resumes execution at the completion of the system call.

Note*:* Learn the diagrams of directory and disk structures and have a light read of the concept.

Unit V

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 5.25 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 contain hundreds of sectors.

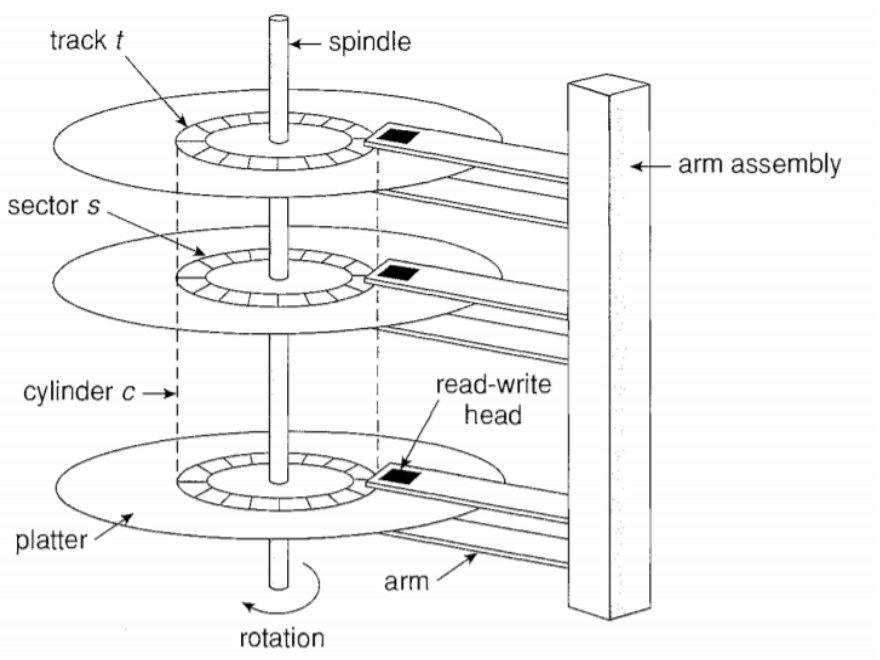


Figure 12.1 Moving-head disk mechanism.

Disk speed has two parts. The transfer rate is the rate at which data flow between the drive and the computer. The positioning time sometimes called the random-access time consists of 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. Typical disks can transfer several megabytes of data per second, and they seek times and rotational latencies of several milliseconds.

A disk can be removable allowing different disks to be mounted as needed. Removable magnetic disks generally consist of one platter, held in a plastic case to prevent damage while not in the disk drive. Floppy disks are inexpensive removable magnetic disks that have a soft plastic case containing a flexible platter. The head of a floppy-disk drive generally sits directly on the disk surface, so the drive is designed to rotate more slowly than a hard-disk drive to reduce the wear on the disk surface. The storage capacity of a floppy disk is typically only 1.44MB or so.

Magnetic Tapes:

Magnetic Tapes was used as an early secondary-storage medium. Although it is relatively permanent and can hold large quantities of dataf its access time is slow compared with that of main memory and magnetic disk. In addition, random access to magnetic tape is about a thousand times slower than random access to magnetic disk, so tapes are not very useful for secondary storage. Tapes are used mainly 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. Tape capacities vary greatly, depending on the particular kind of tape drive. Typically, they store from 20GB to 200GB. Some have built-in compression that can more than double the effective storage. Tapes and their drivers are usually categorized by width, including 4, 8, and 19 millimeters and 1/4 and 1/2 inch. Some are named according to technology, such as LT0-2 and SDLT

Disk Scheduling

Disk scheduling is a technique used by the operating system to schedule multiple requests for accessing the disk. For a multiprogramming system with many processes, the disk queue may often have several pending requests. Thus, when one request is completed, the operating system chooses which pending request to service next. Any one of several disk scheduling algorithms can be used to make that choice.

FCFS:

The simplest form of disk scheduling is, of course, the first-come, first-served (FCFS) algorithm. This algorithm is intrinsically fair, but it generally does not provide the fastest service.
Consider, for example, a disk queue with requests for I/0 to blocks on cylinders 98, 183, 37, 122, 14, 124, 65, 67,

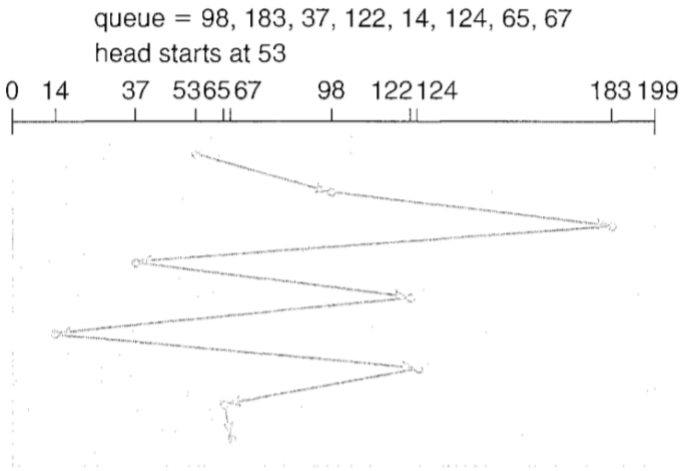


Figure 12.4 FCFS disk scheduling.

in that order. If the disk head is initially at cylinder 53, it will first move from 53 to 98, then to 183, 37, 122, 14, 124, 65, and finally to 67, for a total head movement of 640 cylinders. This schedule is diagrammed in Figure 12.4. The wild swing from 122 to 14 and then back to 124 illustrates the problem with this schedule. If the requests for cylinders 37 and 14 could be serviced together, before or after the requests for 122 and 124, the total head movement could be decreased substantially, and performance could be thereby improved.

SSTF: The SSTF algorithm selects the request with the least seek time from the current head position. Since seek time increases with the number of cylinders traversed by the head, SSTF chooses the pending request closest 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 at 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 method results in a total head movement of only 236 cylinders-little more than one-third of the distance needed for FCFS scheduling of this request queue. Clearly, this algorithm gives a substantial improvement in performance.

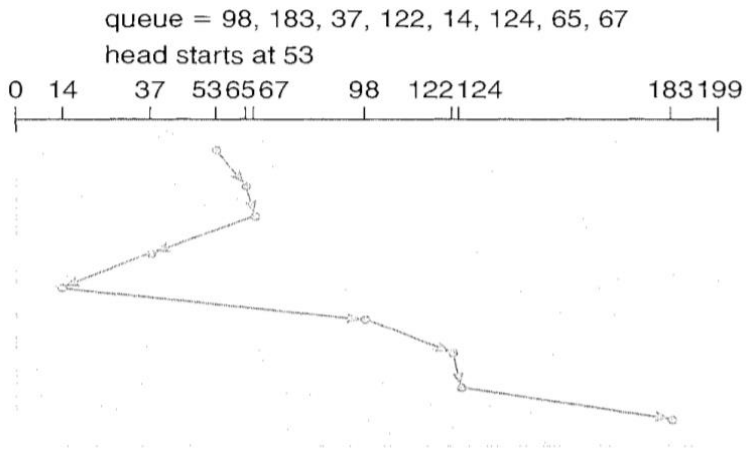


Figure 12.5 SSTF disk scheduling.

Suppose that we have two requests in the queue, for cylinders 14 and 186, and while the request from 14 is being serviced, a new request near 14 arrives. This new request will be serviced next, making the request at 186 wait. While this request is being serviced, another request close to 14 could arrive. In theory, a continual stream of requests near one another could cause the request for cylinder 186 to wait indefinitely.

SCAN:

In the SCAN algorithm, the disk arm starts at one end of the disk and moves toward 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 continuously scans back and forth across the disk. The SCAN algorithm is 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.

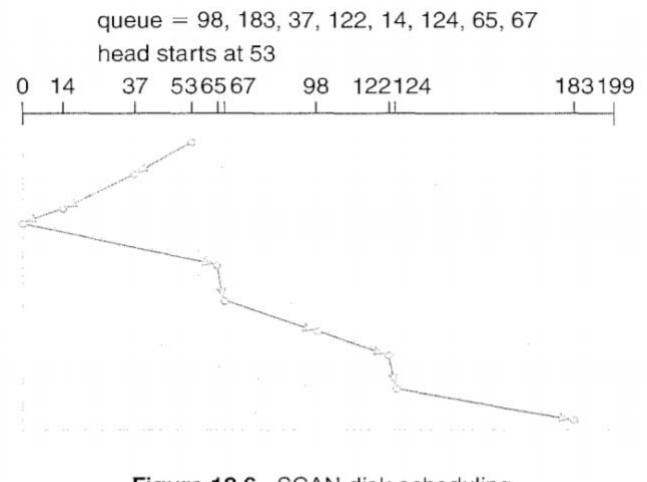


Figure 12.6 SCAN disk scheduling.

Assuming that the disk arm is moving toward 0 and that the initial head position is again 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.

C-SCAN Scheduling:

Circular SCAN(C-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 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.

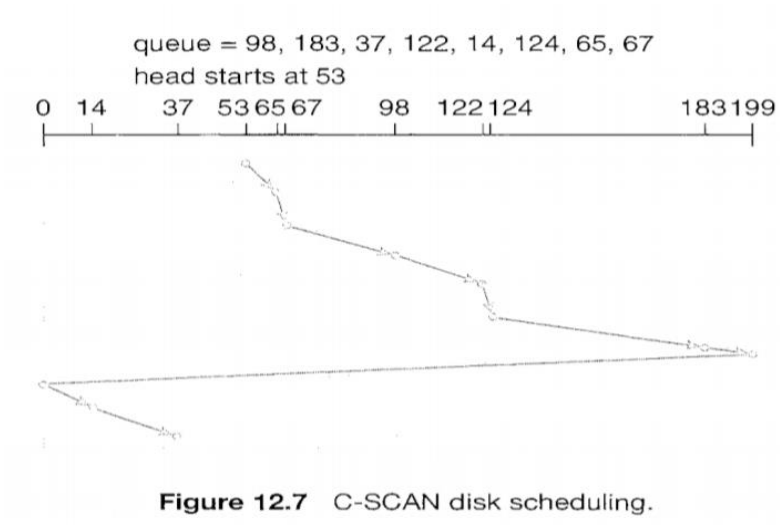


Figure 12.7 C-SCAN disk scheduling.

The C-SCAN scheduling algorithm essentially treats the cylinders as a circular list that wraps around from the final cylinder to the first one.

RAID

Disk drives have continued to get smaller and cheaper, so it is now economically feasible to attach many disks to a computer system. Having a large number of disks in a system presents opportunities for improving the rate at which data can be read or written, if the disks are operated in parallel.

A variety of disk-organization techniques, collectively called redundant array of independent disks (RAIDs), are commonly used to address the performance and reliability issues.

RAID levels:

Numerous schemes to provide redundancy at lower cost by using disk striping combined with "parity" bits have been proposed. These schemes have different cost-performance trade-offs and are classified according to levels called RAID levels.

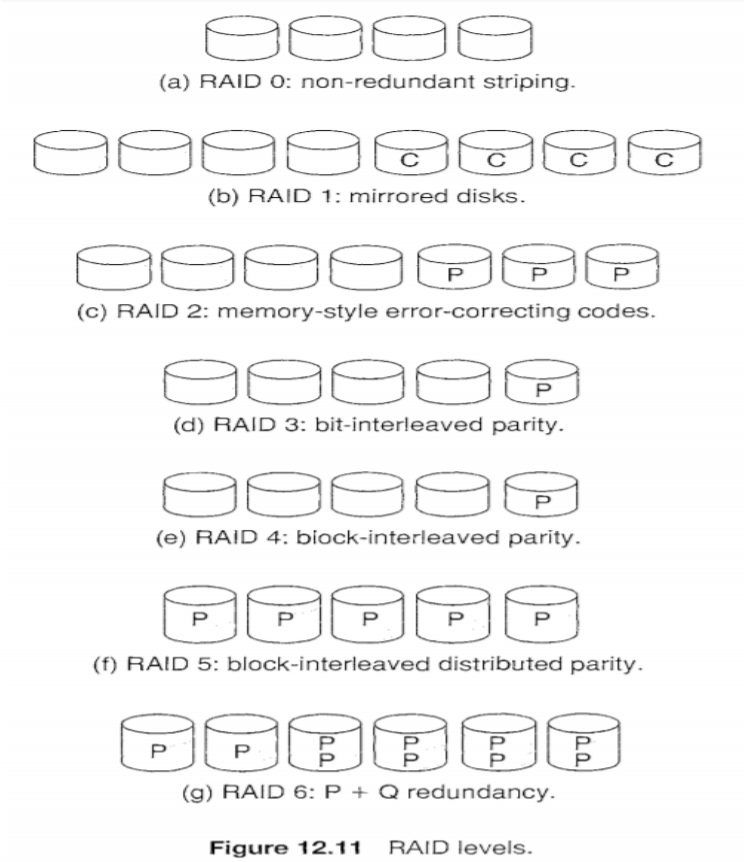


Figure 12.11 RAID levels.

RAID level 0:

RAID level 0 refers to disk arrays with striping at the level of blocks but without any redundancy (such as mirroring or parity bits), as shown in Figure. Each Read/Write request requires access to almost all of the disks. This results in delay in request processing.

RAID Level 1:

RAID level 1 refers to disk mirroring. Data in N disks are copied to other N disks. Figure shows a mirrored organization. With disk mirroring, the rate at which read requests can be handled is doubled, since read requests can be sent to either disk (as long as both disks in a pair are functional as is almost always the case). The transfer rate of each read is the same as in a single-disk system, but the number of reads per unit time has doubled.

RAID level 2:

RAID level 2 is also known as memory-style error-correcting code (ECC) organization. Memory systems have long detected certain errors by using parity bits. Each byte in a memory system may have a parity bit associated

with it that records whether the number of bits in the byte set to 1 is even (parity= 0) or odd (parity= 1). If one of the bits in the byte is damaged (either a 1 becomes a 0, or a 0 becomes and the parity of the byte changes and thus does not match the stored parity. Similarly, if the stored parity bit is damaged, it does not match the computed parity. Thus, all single-bit errors are detected by the menwry system .. Error-correcting schemes store two or more extra bits and can reconstruct the data if a single bit is damaged. The idea of ECC can be ·used directly in disk arrays via striping of bytes across disks.

RAID level 3:

RAID level 3, or bit-interleaved parity organization improves on level 2 by taking into account the fact that, unlike memory systems, disk controllers can detect whether a sector has been read correctly, so a single parity bit can be used for error correction as well as for detection. The idea is as follows: If one of the sectors is damaged, we know exactly which sector it is, and we can figure out whether any bit in the sector is a 1 or a 0 by computing the parity of the corresponding bits from sectors in the other disks. If the parity of the remaining bits is equal to the stored parity, the missing bit is 0; otherwise, it is 1. RAID level3 is as good as level 2 but is less expensive in the number of extra disks required (it has only a one-disk overhead), so level 2 is not used in practice.

RAID level 3 has two advantages over level 1. First, the storage overhead is reduced because only one parity disk is needed for several regular disks, whereas one mirror disk is needed for every disk in level1. Second, since reads and writes of a byte are spread out over multiple disks with N-way striping of data, the transfer rate for reading or writing a single block is N times as fast as with RAID level 1. On the negative side, RAID level3 supports fewer l/Os per second, since every disk has to participate in every I/0 request.

RAID level 4:

RAID level4, or block-interleaved parity uses block-level striping, as in RAID 0, and in addition keeps a parity block on a separate disk for

corresponding blocks from N other disks. If one of the disks fails, the parity block can be used with the corresponding blocks from the other disks to restore the blocks of the failed disk. A block read accesses only one disk, allowing other requests to be processed by the other disks. Thus, the data-transfer rate for each access is slowe1~ but multiple read accesses can proceed in parallel, leading to a higher overall I/0 rate. The transfer rates for large reads are high, since all the disks can be read in parallel; large writes also have high transfer rates, since the data and parity can be written in parallel.

RAID level 5:

RAID level 5, or block interleaved distributed parity, differs from level 4 by spreading data and parity among all N + 1 disks, rather than storing data in N disks and parity in one disk. For each block, one of the disks stores the parity, and the others store data. For example, with an array of five disks, the parity for the nth block is stored in disk (n mod 5)+ 1; the nth blocks of the other four disks store actual data for that block. A parity block cannot store parity for blocks in the same disk, because a disk failure would result in loss of data as well as of parity, and hence the loss would not be recoverable. By spreading the parity across all the disks in the set, RAID 5 avoids potential overuse of a single parity disk, which can occur with RAID 4.

RAID level 6:

RAID level 6, also called P+Q redundancy scheme is much like RAID level 5 but stores extra redundant information to guard against disk failures. Instead of parity, error-correcting codes such as the Read Solomon codes are used. 2 bits of redundant data are stored for every 4 bits of data compared with 1 parity bit in level 5-and the system can tolerate two disk failures.

RAID levels 0 + 1 and 1 + 0:

RAID level 0 + 1 refers to a combination of RAID levels 0 and 1. RAID 0 provides the performance, while RAID 1 provides the reliability. Generally, this level provides better performance than RAID 5. It is common in environments where both performance and reliability are important.

Unfortunately, like RAID 1, it doubles the number of disks needed for storage, so it is also relatively expensive. In RAID 0 + 1, a set of disks are striped, and then the stripe is mirrored to another, equivalent stripe.

Another RAID option that is becoming available commercially is RAID level 1 + 0, in which disks are mirrored in pairs and then the resulting mirrored pairs are striped. This scheme has some theoretical advantages over RAID 0 + 1. For example, if a single disk fails in RAID 0 + 1, an entire stripe is inaccessible, leaving only the other stripe available. With a failure in RAID 1 + 0, a single disk is unavailable, but the disk that mirrors it is still available, as are all the rest of the disks.

Goals of protection

As computer systems have become more sophisticated and pervasive in their applications, the need to protect their integrity has also grown. Protection was originally conceived as an adjunct to multiprogramming operating systems, so that untrustworthy users might safely share a common logical name space, such as a directory of files, or share a common physical namespace, such as memory. Modern protection concepts have evolved to increase the reliability of any complex system that makes use of shared resources. We need to provide protection for several reasons. The most obvious is the need to prevent the mischievous, intentional violation of an access restriction by a user. Of more general importance, however, is the need to ensure that each program component active in a system uses system resources only in ways consistent with stated policies. This requirement is an absolute one for a reliable system. Protection can improve reliability by detecting latent errors at the interfaces between component subsystems. Early detection of interface errors can often prevent contamination of a healthy subsystem by a malfunctioning subsystem. Also, an unprotected resource cannot defend against use (or misuse) by an unauthorized or incompetent user. A protection-oriented system provides means to distinguish between authorized and unauthorized usage. The role of protection in a computer system is to provide a mechanism for the enforcement of the policies governing resource use. These policies can be established in a variety of ways. Some are fixed in the design of the system, while others are formulated by the

management of a system. Still others are defined by the individual users to protect their own files and programs. A protection system must have the flexibility to enforce a variety of policies. Policies for resource use may vary by application, and they may change over time. For these reasons, protection is no longer the concern solely of the designer of an operating system. The application programmer needs to use protection mechanisms as well, to guard resources created and supported by an application subsystem against misuse. In this chapter, we describe the protection mechanisms the operating system should provide, but application designers can use them as well in designing their own protection software. Note that mechanisms are distinct from policies. Mechanisms determine how something will be done; policies decide what will be done. The separation of policy and mechanism is important for flexibility. Policies are likely to change from place to place or time to time. In the worst case, every change in policy would require a change in the underlying mechanism. Using general mechanisms enables us to avoid such a situation.

Principles of protection

Frequently, a guiding principle can be used throughout a project, such as the design of an operating system. Following this principle simplifies design decisions and keeps the system consistent and easy to understand. A key, time-tested guiding principle for protection is the principle of least privilege. It dictates that programs, users, and even systems be given just enough privileges to perform their tasks. Consider the analogy of a security guard with a passkey. If this key allows the guard into just the public areas that she guards, then misuse of the key will result in minimal damage. If, however, the passkey allows access to all areas, then damage from its being lost, stolen, misused, copied, or otherwise compromised will be much greater. An operating system following the principle of least privilege implements its features, programs, system calls, and data structures so that failure or compromise of a component does the minimum damage and allows the minimum damage to be done. The overflow of a buffer in a system daemon might cause the daemon process to fail, for example, but should not allow the execution of code from the daemon process's stack

that would enable a remote user to gain maximum privileges and access to the entire system (as happens too often today). Such an operating system also provides system calls and services that allow applications to be written with fine-grained access controls. It provides mechanisms to enable privileges when they are needed and to disable them when they are not needed. Also beneficial is the creation of audit trails for all privileged function access. The audit trail allows the programmer, systems administrator, or law-enforcement officer to trace all protection and security activities on the system. Managing users with the principle of least privilege entails creating a separate account for each user, with just the privileges that the user needs. An operator who needs to mount tapes and back up files on the system has access to just those commands and files needed to accomplish the job. Some systems implement role-based access control (RBAC) to provide this functionality. Computers implemented in a computing facility under the principle of least privilege can be limited to running specific services, accessing specific remote hosts via specific services, and doing so during specific times.

Access Matrix

Access Matrix is a security model of protection state in computer systems. It is represented as a **matrix**. **Access matrix** is used to define the rights of each process executing in the domain with respect to each object. The rows of the access matrix represent domains, and the columns represent objects. Each entry in the matrix consists of a set of access rights. Because the column defines objects explicitly, we can omit the object name from the access right. The entry access(i,j) defines the set of operations that a process executing in domain Di can invoke on object Oj.

There are four domains and four objects-three files (F1, F2, F3) and one laser printer. A process executing in domain D1 can read files F1 and F3 . A process executing in domain D4 has the same privileges as one executing in domain D1; but in addition, it can also write onto files F1 and F3 . Note that the laser printer can be accessed only by a process executing in domain D2. The access-matrix scheme provides us with the mechanism for specifying a variety of policies.

object / domain	F_1	F_2	F_3	printer
D_1	read		read	
D_2				print
D_3		read	execute	
D_4	read write		read write	

Figure 14.3 Access matrix.

The access matrix can implement policy decisions concerning protection. The policy decisions involve which rights should be included in the (i,j)th entry. We must also decide the domain in which each process executes. This last policy is usually decided by the operating system. The users normally decide the contents of the access-matrix entries. When a user creates a new object Oj, the column Oj is added to the access matrix with the appropriate initialization entries, as dictated by the creator. The user may decide to enter some rights in some entries in column j and other rights in other entries, as needed.

The access matrix provides an appropriate mechanism for defining and implementing strict control for both the static and dynamic association between processes and domains. When we switch a process from one domain to another, we are executing an operation (switch) on an object (the domain). We can control domain switching by including domains among the objects of the access matrix. Similarly, when we change the content of the access matrix, we are performing an operation on an object: the access matrix. Again, we can control these changes by including the access matrix itself as an object. Actually, since each entry in the access matrix may be modified individually, we must consider each entry in the access matrix as an object to be protected. Now, we need to consider only the operations possible on these new objects (domains and the access matrix) and decide how we want processes to be able to execute these operations.

Processes should be able to switch from one domain to another. Switching from domain Di to domain Dj is allowed if and only if the access right switch E access(i,j). Thus, a process executing in domain D2 can switch to domain D3 or to domain D4 . A process in domain D4 can switch to D1, and one in domain D1 can switch to D2. Allowing controlled change in the contents of the access-matrix entries requires three additional operations: copy, owner, and control.

object / domain	F_1	F_2	F_3	laser printer	D_1	D_2	D_3	D_4
D_1	read		read			switch		
D_2				print			switch	switch
D_3		read	execute					
D_4	read write		read write		switch			

Figure 14.4 Access matrix of Figure 14.3 with domains as objects.

The ability to copy an access right from one domain (or row) of the access matrix to another is denoted by an asterisk (*) appended to the access right. The copy right allows the access right to be copied only within the column. (that is, for the object) for which the right is defined.

object / domain	F_1	F_2	F_3
D_1	execute		write*
D_2	execute	read*	execute
D_3	execute		

(a)

object / domain	F_1	F_2	F_3
D_1	execute		write*
D_2	execute	read*	execute
D_3	execute	read	

(b)

This scheme has two variants:

A right is copied from access(i, j) to access(k, j); it is then removed from access(i, j). This action is a transfer of a right, rather than a copy.

Propagation of the copy right may be limited. That is, when the right R* is copied from access(i,j) to access(k,j), only the right R (not R*) is created. A process executing in domain Dk cannot further copy the right R.

Program Threats

Processes, along with the kernel, are the only means of accomplishing work on a computer. Therefore, writing a program that creates a breach of security, or causing a normal process to change its behavior and create a breach, is a common goal of crackers. In fact even most nonprogram security events have as their goal causing a program threat.

Trojan Horse:

A code segment that misuses its environment is called a *Trojan Horse*. A text-editor program, for example, may include code to search the file to be edited for certain keywords. If any are found, the entire file may be copied to a special area accessible to the creator of the text editor.

Long search paths, such as are common on UNIX systems, exacerbate the Trojan Horse The search path lists the set of directories to search when a program name is given. The path is searched for a file of that name, and the file is executed. All the directories in such a search path must be secure, or a trojan horse could be slipped into the user's path and executed.

A variation of trojan horse is a program that emulates a login program. An unsuspecting user starts to login at a terminal and notices that he has apparently mistyped his password. He tries again and gets successful. What actually has happened is his login id and password have been stolen by the emulator, which is left running on the terminal by the thief.

Another variation on the Trojan horse is spyware. A Spyware sometimes accompanies a program that the user has chosen to install. Most frequently, it comes along with freeware or shareware programs, but sometimes it is included with commercial software. The goal of spyware is to download ads to display on the user's system, create pop-up windows when certain sites are visited, or capture information from the user's system and return it to a central site.

Trap Door:

The designer of a program or system might leave a hole in the software that only he is capable of using. For instance, the code Insight checks a specific user ID or password, and it might circumvent normal security procedures. Programmers have been arrested for embezzling from banks by including rounding errors in their code and having the occasional half-cent credited to their accounts. This account crediting can add up to a large amount of money, considering the number of transactions that a large bank executes. A clever trap door could be included in a compiler. The compiler could generate standard object code as well as a trap door, regardless of the source code being compiled. This activity is particularly nefarious, since a search of the source code of the program will not reveal any problems. Only the source code of the compiler would contain the information. Trap doors pose a difficult problem because, to detect them, we have to analyze all the source code for all components of a system.

Logic Bomb:

Consider a program that initiates a security incident only under certain circUmstances. It would be hard to detect because under normal operations, there would be no security hole. However, when a predefined set of parameters were met, the security hole would be created. This scenario is known as a logic bomb. A programmer, for example, might write code to detect whether he/she was still employed; if that check failed, a daemon could be spawned to allow remote access, or code could be launched to cause damage to the site.

Stack or Buffer Overflow:

The stack- or buffer-overflow attack is the most common way for an attacker outside the system, on a network or dial-up connection, to gain unauthorized access to the target system. An authorized user of the system may also use this exploit for privilege escalation.

Essentially, the attack exploits a bug in a program. The bug can be a simple case of poor programming, in which the programmer neglected to code bounds checking on an input field. In this case, the attacker sends more data than the program was expecting. By using trial and error, or by examining the source code of the attacked program if it is available, the attacker determines the vulnerability and writes a program to do the following:

1. Overflow an input field, command-line argument, or input buffer-for example, on a network daemon-until it writes into the stack.
2. Overwrite the current return address on the stack with the address of the exploit code loaded in step 3.
3. Write a simple set of code for the next space in the stack that includes the commands that the attacker wishes to execute-for instance, spawn a shell.

The result of this attack program's execution will be a root shell or other privileged command execution.

Consider the simple C program. This program creates a character array size BUFFER_SIZE and copies the contents of the parameter provided on the command line-argv [1]. As long as the size of this parameter is less than BUFFER_SIZE (we need one byte to store the null terminator), this program works properly. But consider what happens if the

parameter provided on the command line is longer than BUFFER_SIZE. In this scenario, the strcpy () function will begin copying from argv [1] until it encounters a null terminator (\0) or until the program crashes. Thus, this program suffers from a potential problem in which copied data overflows the buffer array.

```
#include <stdio.h>
#define BUFFER_SIZE 256
int main(int argc, char *argv[]) {
char buffer[BUFFER_SIZE];
if (argc < 2) return -1; else { strcpy(buffer,argv[1]); return 0; }
}
```

When a function is invoked in a typical computer architecture, the variables defined locally to the function, the parameters passed to the function, and the address to which control returns once the function exits are stored in a stack frame.

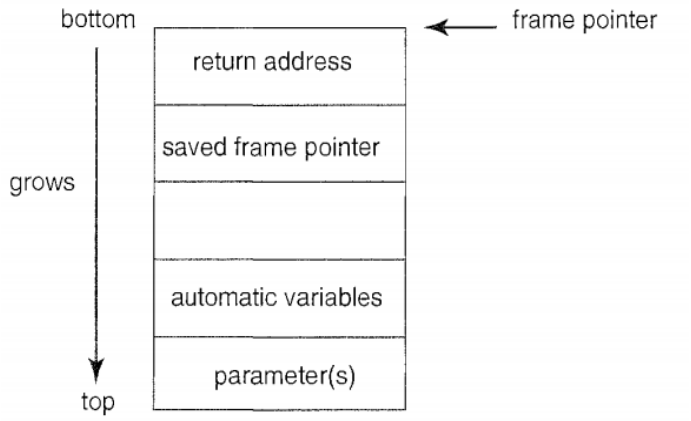


Figure 15.3 The layout for a typical stack frame.

Given this standard memory layout, a cracker could execute a buffer overflow attack. Her goal is to replace the return address in the stack frame so that it now points to the code segment containing the attacking program. The programmer first writes a short code segment such as the following:

```
#include <stdio.h>
int main(int argc, char *argv[])
{
execvp(' "\bin\sh", "\bin \sh",  NULL);
return 0;
```

```
}
```

Using the execvp () system call, this code segment creates a shell process. If the program being attacked runs with system-wide permissions, this newly created shell will gain complete access to the system.