MODULE - VI

6.1 Overview of Mass-Storage Structure

This session gives an overview of the physical structure of secondary and tertiary storage devices.

1. Magnetic Disks

- Magnetic disks provide the bulk of secondary storage for modern computer systems. Conceptually, disks are relatively simple each disk platter has a flat circular shape, like a CD. Common platter diameters range from1.8 to 3.5 inches. The two surfaces of a platter are covered with a magnetic material. We store information by recording it magnetically on the platters.
- A read–write head "flies" just above each surface of every platter. The heads are attached to a disk arm that moves all the heads as a unit. The surface of a platter is logically divided into circular tracks, which are subdivided into sectors. The set of tracks that are at one arm position makes up a cylinder. There may be thousands of concentric cylinders in a disk drive, and each track may contain hundreds of sectors. The storage capacity of common disk drives is measured in gigabytes.
- When the disk is in use, a drive motor spins it at high speed. Most drives rotate 60 to 250 times per second, specified in terms of rotations per minute (RPM). Disk speed has two parts. The transfer rate is the rate at which data flow between the drive and the computer. The positioning time, or random-access time, consists of two parts: the time necessary to move the disk arm to the desired cylinder, called the seek time, and the time necessary for the desired sector to rotate to the disk head, called the rotational latency.

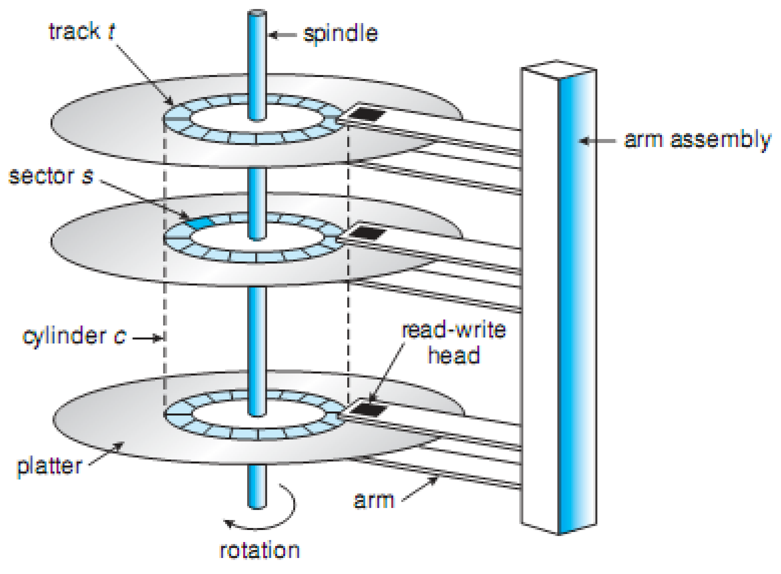


Figure 10.1 Moving-head disk mechanism.

- A disk drive is attached to a computer by a set of wires called an I/O bus. Several kinds of buses are available, including advanced technology attachment (ATA), serial ATA (SATA), e-SATA, universal serial bus (USB), and fiber channel (FC). The data transfers on a bus are carried out by special electronic processors called controllers.

2. Magnetic Tapes

- Magnetic tape was used as an early secondary-storage medium. Although it is relatively permanent and can hold large quantities of data, 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.
- 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 LTO-5 and SDLT.

6.2 Disk Structure

- Modern magnetic disk drives are addressed as large one-dimensional arrays of logical blocks, where the logical block is the smallest unit of transfer. The size of a logical block is usually 512 bytes, although some disks can be low-level formatted to have a different logical block size, such as 1,024 bytes.

- The one-dimensional array of logical blocks is mapped onto the sectors of the disk sequentially. Sector 0 is the first sector of the first track on the outermost cylinder. The mapping proceeds in order through that track, then through the rest of the tracks in that cylinder, and then through the rest of the cylinders from outermost to innermost. By using this mapping, we an—at least in theory—convert a logical block number into an old-style disk address that consists of a cylinder number, a track number within that cylinder, and a sector number within that track.
- In practice it is difficult to perform this translation because of the following reasons
    1. most disks have some defective sectors, but themapping hides this by substituting spare sectors from elsewhere on the disk.
    2. Second, the number of sectors per track is not a constant on some drives.
- Arranging tracks in different manner , which includes

    1. **Constant Linear Velocity (CLV):** The density of bits per track is uniform. The farther a track is from the center of the disk, the greater its length, so the more sectors it can hold. As we move from outer zones to inner zones, the number of sectors per track decreases.
    2. **Constant Angular Velocity (CAV):** Tracks in the outermost zone typically hold 40 percent more sectors than do tracks in the innermost zone. The drive increases its rotation speed as the head moves from the outer to the inner tracks to keep the same rate of data moving under the head. This method is used in CD-ROM and DVD-ROM drives. Alternatively, the disk rotation speed can stay constant; in this case, the density of bits decreases from inner tracks to outer tracks to keep the data rate constant. This method is used in hard disks and is known as constant angular velocity (CAV).
- The number of sectors per track has been increasing as disk technology improves, and the outer zone of a disk usually has several hundred sectors per track. Similarly, the number of cylinders per disk has been increasing; large disks have tens of thousands of cylinders.

6.3 Disk Scheduling

- The **seek time** is the time for the disk arm to move the heads to the cylinder containing the desired sector.
- The **rotational latency** is the additional time for the disk to rotate the desired sector to the disk head.
- The **disk bandwidth** is the total number of bytes transferred, divided by the total time between the first request for service and the completion of the last transfer.
- We can improve both the access time and the bandwidth by managing the order in which disk I/O requests are serviced.
- 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. The operating system makes this choice by any one of several disk-schedulingalgorithms.

1. FCFS Scheduling

- It is the simplest of the scheduling algorithms. Consider, for example, a disk queue with requests for I/O to blocks on cylinders in that order.

98, 183, 37, 122, 14, 124, 65, 67

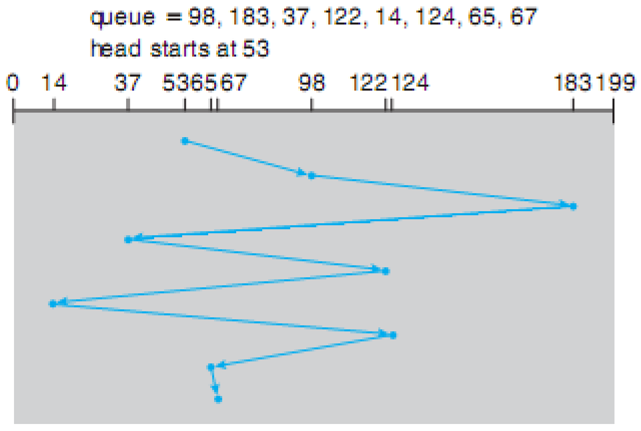


Figure 10.4 FCFS disk scheduling.

- 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.

- ➢ 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

2. SSTF Scheduling

- ➢ It seems reasonable to service all the requests close to the current head position before moving the head far away to service other requests. This assumption is the basis for the shortest-seek-time-first (SSTF) algorithm. The SSTF algorithm selects the request with the least seek time from the current head position. In other words, SSTF chooses the pending request closest to the current head position.

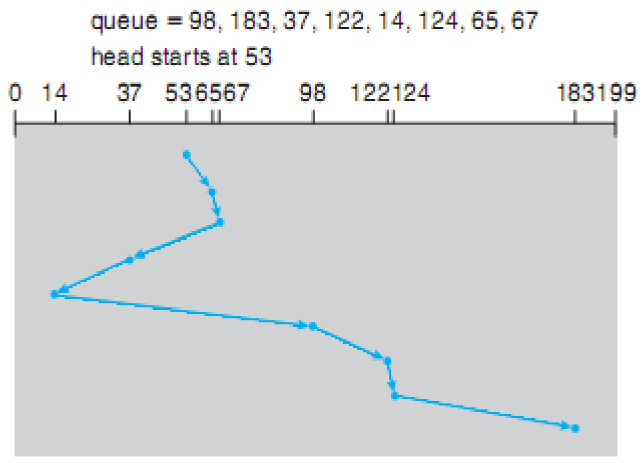


Figure 10.5 SSTF disk scheduling.

- ➢ 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, thn 98, 122, 124, and finally 183 (Figure 10.5). 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
- ➢ This scheduling algorithm has the disadvantages of starvation.
- ➢ Although it is not optimal ie.. In the example, we can do better by moving the

head from 53 to 37, even though the latter is not closest, and then to 14, before turning around to service 65, 67, 98, 122, 124, and 183. This strategy reduces the total head movement to 208 cylinders.

3. SCAN Scheduling

- 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.

Example: 98, 183, 37, 122, 14, 124, 65, 67

- Before applying SCAN to schedule the requests on cylinders 98, 183, 37, 122, 14, 124, 65, and 67, we need to know the direction of head movement in addition to the head's current position.
- Assuming that the disk arm is moving toward 0 and that the initial head position is again 53, the head will next service 37 and then 14.

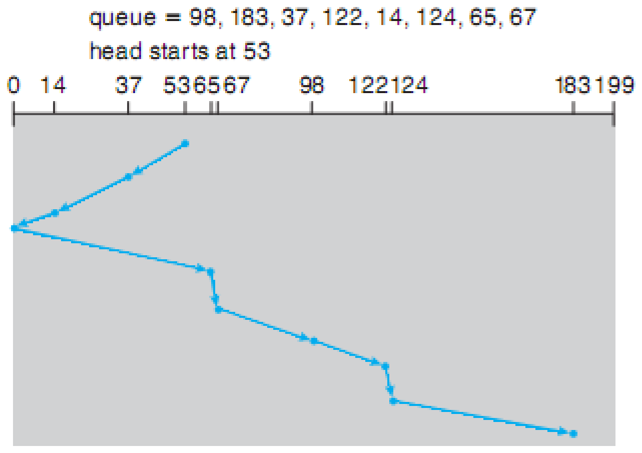


Figure 10.6 SCAN disk scheduling.

- 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.

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

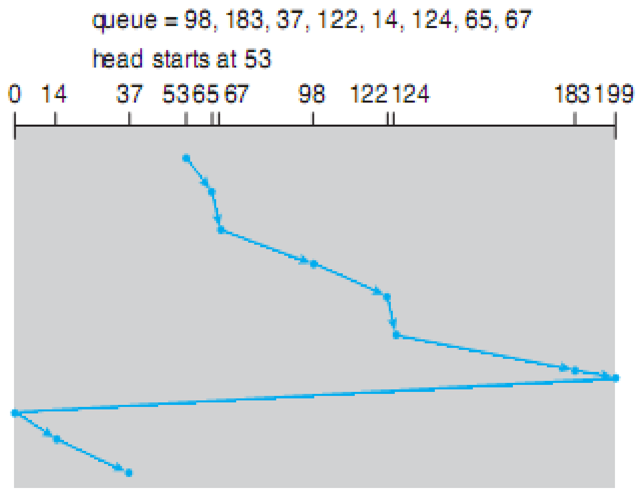


Figure 10.7 C-SCAN disk scheduling.

5. LOOK and C-LOOK Scheduling

- As we described them, both SCAN and C-SCAN move the disk arm across the full width of the disk. In practice, neither algorithm is often implemented this way.
- More commonly, the arm goes only as far as the final request in each direction. Then, it reverses direction immediately, without going all the way to the end of the disk.

- Versions of SCAN and C-SCAN that follow this pattern are called LOOK and C-LOOK scheduling, because they look for a request before continuing to move in a given direction

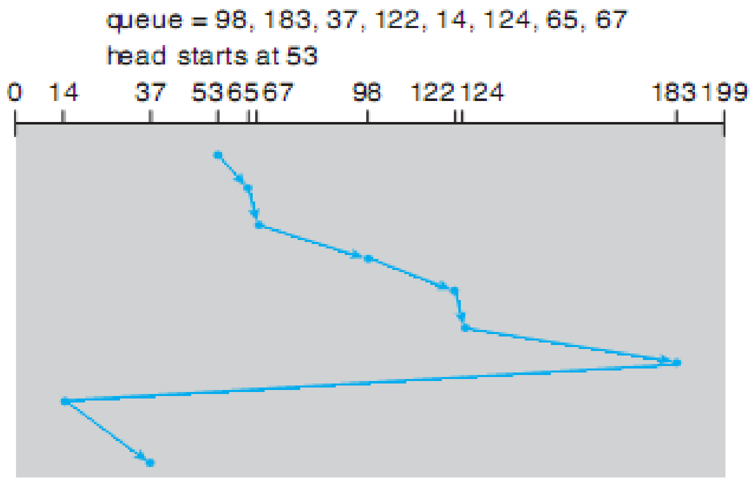


Figure 10.8 C-LOOK disk scheduling.

6.4 Disk Management

- Here we discuss disk initialization, booting from disk, and bad-block recovery.

1. Disk Formatting

- A new magnetic disk is a blank slate: it is just a platter of a magnetic recording material. Before a disk can store data, it must be divided into sectors that the disk controller can read and write. This process is called low-level formatting,or physical formatting.
- Low-level formatting fills the disk with a special data structure for each sector. The data structure for a sector typically consists of a header, a data area (usually 512 bytes in size), and a trailer.
- The header and trailer contain information used by the disk controller, such as a sector number and an error-correcting code (ECC).
- When the controller writes a sector of data during normal I/O, the ECC is updated with a value calculated from all the bytes in the data area. When the sector is read, the ECC is recalculated and compared with the stored value. If the stored and calculated numbers are different, this mismatch indicates that the data area of the sector has become corrupted and that the disk sector may be bad.

- The ECC is an error-correcting code because it contains enough information, if only a few bits of data have been corrupted, to enable the controller to identify which bits have changed and calculate what their correct values should be.
- It then reports a recoverable soft error. The controller automatically does the ECC processing whenever a sector is read or written.
- For many hard disks, when the disk controller is instructed to low-level-format the disk, it can also be told how many bytes of data space to leave between the header and trailer of all sectors.
- It is usually possible to choose among a few sizes, such as 256, 512, and 1,024 bytes. Formatting a disk with a larger sector size means that fewer sectors can fit on each track; but it also means that fewer headers and trailers are written on each track and more space is available for user data.
- Some operating systems can handle only a sector size of 512 bytes.
- The first step is to partition the disk into one or more groups of cylinders. The operating system can treat each partition as though it were a separate disk.
- The second step is logical formatting, or creation of a file system. In this step, the operating system stores the initial file-system data structures onto the disk.
- To increase efficiency, most file systems group blocks together into larger chunks, frequently called clusters.
- Some operating systems give special programs the ability to use a disk partition as a large sequential array of logical blocks, without any file-system data structures.

2. Boot Block

- For a computer to start running it must have an initial program to run. This initial bootstrap program initializes all aspects of the system, from CPU registers to device controllers and the contents of main memory, and then starts the operating system. To do its job, the bootstrap program finds the operating-system kernel on disk, loads that kernel into memory, and jumps to an initial address to begin the operating-system execution.
- For most computers, the bootstrap is stored in read-only memory (ROM). Because ROM needs no initialization and is at a fixed location that the processor can start executing when powered up or reset and it cannot be infected by a computer virus.
- Most systems store a tiny bootstrap loader program in the boot ROM whose only job is to bring in a full bootstrap program from disk.
- The full bootstrap program is stored in the "boot blocks" at a fixed location on the disk. A disk that has a boot partition is called a boot disk or system disk.
- The code in the boot ROM instructs the disk controller to read the boot blocks

into memory (no device drivers are loaded at this point) and then starts executing that code

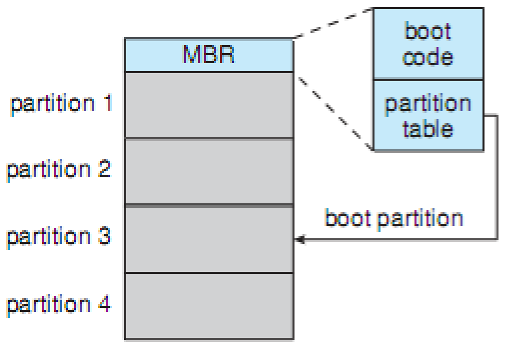


Figure 10.9 Booting from disk in Windows.

Example: Boot process in windows –

1. Windows allows a hard disk to be divided into partitions, and one partition contains the operating system and device drivers.
2. The Windows system places its boot code in the first sector on the hard disk, which it terms the master boot record, or **MBR**.
3. Booting begins by running code that is resident in the system's ROM memory. This code directs the system to read the boot code from the MBR. In addition to containing boot code, the MBR contains a table listing the partitions for the hard disk and a flag indicating which partition the system is to be booted from.
4. Once the system identifies the boot partition, it reads the first sector from that partition (which is called the **boot sector**) and continues with the remainder of the boot process, which includes loading the various subsystems and system services.

3. Bad Blocks

- ➢ Because disks have moving parts and small tolerances they are prone to failure. Sometimes the failure is complete; in this case, the disk needs to be replaced and its contents restored from backup media to the new disk. More frequently, one or more sectors become defective.
- ➢ Most disks even come from the factory with bad blocks. Depending on the disk and controller in use, these blocks are handled in a variety of ways.
- ➢ Bad blocks are handled manually in small devices or by special programs like bad blocks in Linux.
- ➢ More sophisticated disks are smarter about bad-block recovery. The controller

maintains a list of bad blocks on the disk. The list is initialized during the low-level formatting at the factory and is updated over the life of the disk. Low-level formatting also sets aside spare sectors not visible to the operating system. The controller can be told to replace each bad sector logically with one of the spare sectors. This scheme is known as **sector sparing** or **forwarding**.

- A typical bad-sector transaction might be as follows
  1. The operating system tries to read logical block 87
  2. The controller calculates the ECC and finds that the sector is bad. It reports this finding to the operating system.
  3. The next time the system is rebooted, a special command is run to tell the controller to replace the bad sector with a spare.
  4. After that, whenever the system requests logical block 87, the request is translated into the replacement sector's address by the controller
- When a bad block is remapped, the controller uses a spare sector from the same cylinder, if possible.
- As an alternative to sector sparing, some controllers can be instructed to replace a bad block by **sector slipping**.

  Example: Suppose that logical block 17 becomes defective and the first available spare follows sector 202. Sector slipping then remaps all the sectors from 17 to 202, moving them all down one spot. That is, sector 202 is copied into the spare, then sector 201 into 202, then 200 into 201, and so on, until sector 18 is copied into sector 19. Slipping the sectors in this way frees up the space of sector 18 so that sector 17 can be mapped to it.

- The replacement of a bad block generally is not totally automatic, because the data in the bad block are usually lost. Soft errors may trigger a process in which a copy of the block data is made and the block is spared or slipped. An unrecoverable hard error, however, results in lost data. Whatever file was using that block must be repaired (for instance, by restoration from a backup tape), and that requires manual intervention.

6.5 Swap Space Management

- It is moving entire processes between disk and main memory. It occurs when the amount of physical memory reaches a critically low point and processes are moved from memory to swap space to free available memory.
- In practice most systems combine swapping with virtual memory techniques. That is the merging of these two concepts "swapping" and "paging".
- The main goal for the design and implementation of swap space is to provide the best throughput for the virtual memory system.

- In this section, we discuss how swap space is used, where swap space is located on disk, and how swap space is managed.

1. Swap-Space Use

- Swap space is used in various ways by different operating systems, depending on the memory-management algorithms in use.
- The amount of swap space needed on a systemcan therefore vary froma few megabytes of disk space to gigabytes, depending on the amount of physical memory, the amount of virtual memory it is backing, and the way in which the virtual memory is used.
- These swap spaces are usually placed on separate disks so that the load placed on the I/O system by paging and swapping can be spread over the system's I/O Bandwidth.

2. Swap-Space Location

- A swap space can reside in one of two places: it can be carved out of the normal file system, or it can be in a separate disk partition.
- External fragmentation can greatly increase swapping times by forcing multiple seeks during reading or writing of a process image. We can improve performance by caching the block location information in physical memory and by using special tools to allocate physically contiguous blocks for the swap file, but the cost of traversing the file-system data structures remains.
- Alternatively, swap space can be created in a separate raw partition. No file system or directory structure is placed in this space. Rather, a separate swap-space storage manager is used to allocate and deallocate the blocks from the raw partition.
- This manager uses algorithms optimized for speed rather than for storage efficiency, because swap space is accessed much more frequently than file systems (when it is used).
- Internal fragmentation may increase, but this trade-off is acceptable because the life of data in the swap space generally is much shorter than that of files in the file system.
- Since swap space is reinitialized at boot time, any fragmentation is short-lived. The raw-partition approach creates a fixed amount of swap space during disk partitioning.

- Adding more swap space requires either repartitioning the disk (which involves moving the other file-system partitions or destroying them and restoring them from backup) or adding another swap space elsewhere.

2. Swap-Space Management: An Example

- The traditional UNIX kernel started with an implementation of swapping that copied entire processes between contiguous disk regions and memory.
- UNIX later evolved to a combination of swapping and paging as paging hardware became available.
- In Solaris 1 (SunOS), the designers changed standard UNIX methods to improve efficiency and reflect technological developments. When a process executes, text-segment pages containing code are brought in from the file system, accessed in main memory, and thrown away if selected for page out. Swap space is only used as a backing store for pages of **anonymous memory**, which includes memory allocated for the stack, heap, and uninitialized data of a process.

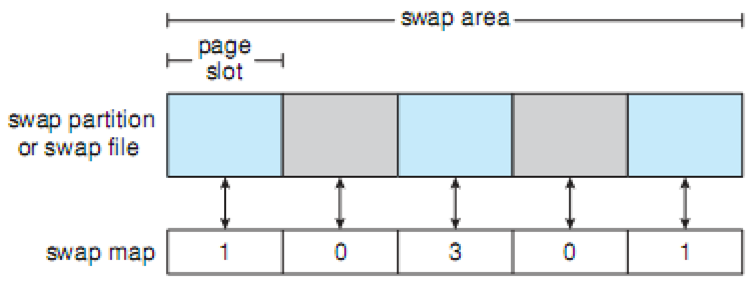


Figure 10.10 The data structures for swapping on Linux systems.

- More changes were made in later versions of Solaris. The biggest change is that Solaris now allocates swap space only when a page is forced out of physical memory, rather than when the virtual memory page is first created. This scheme gives better performance on modern computers, which have more physical memory than older systems and tend to page less.
- Linux is similar to Solaris in that swap space is used only for anonymous memory—that is, memory not backed by any file. Linux allows one or more swap areas to be established. A swap area may be in either a swap file on a regular file system or a dedicated swap partition. Each swap area consists of a series of 4-KB page slots, which are used to hold swapped pages.
- Associated with each swap area is a swap map—an array of integer counters, each corresponding to a page slot in the swap area. If the value of a counter is 0,

the corresponding page slot is available.

- ➢ Values greater than 0 indicate that the page slot is occupied by a swapped page. The value of the counter indicates the number of mappings to the swapped page. For example, a value of 3 indicates that the swapped page is mapped to three different processes (which can occur if the swapped page is storing a region of memory shared by three processes).

6.6 File Concepts

- ➢ A file is a named collection of related information that is recorded on secondary storage.
- ➢ files represent programs and data.
- ➢ The information in a file is defined by its creator.
- ➢ A file has a certain defined structure which depends on its type.
  - **text file, source file, executable file**

1. File Attributes

- ➢ A file is named, for the convenience of its human users, and is referred to by its name.
- ➢ 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.

2. File Operations

- A file is an abstract data type.
- To define a file properly, it needs 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.

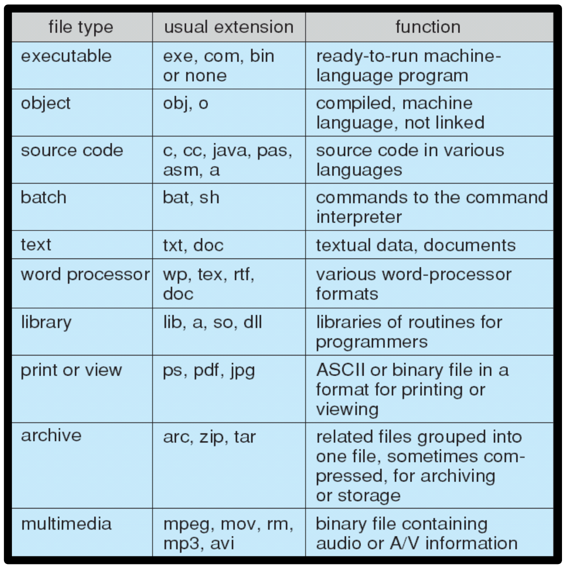
  Writing a file: To write a file, we make a system call specifying both the Name of the file and the information to be written to the file. The system must keep a write pointer to the location in the file where the next write is to take place.

- **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.
- **Repositioning within a file.** The directory is searched for the appropriate entry, and the current-file-position pointer is repositioned to a given value (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.
- Other common operations include
  - Appending, renaming and copy
  - Several pieces of information are associated with an open file.
  - File pointer.
  - File-open count.
  - Disk location of the file.
  - Access rights

3. File Types

- 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.

➢ The system uses the extension to indicate the type of the file and the type of operations that can be done on that file.



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

4. File Structure

➢ File types also can be used to indicate the internal structure of the file.

➢ The operating system requires that an executable file have a specific structure so that it can determine where in memory to load the file and what the location of the first instruction is.

➢ None - sequence of words, bytes

➢ Simple record structure
- Lines
- Fixed length
- Variable length

➢ Complex Structures
- Formatted document
- Relocatable load file

➢ Can simulate last two with first method by inserting appropriate control characters

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.

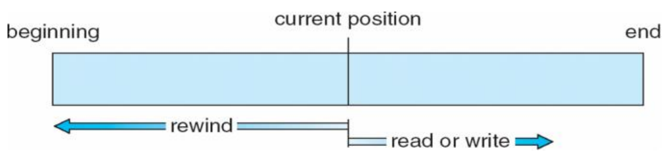
- All disk I/0 is performed in units of one block (physical record), and all blocks are the same size.
- It is unlikely that the physical record size will exactly match the length of the desired logical record. Logical records may even vary in length. Packing a number of logical records into physical blocks is a common solution to this problem.

6.7 Access Methods

- Files store information.
- When it is used, this information must be accessed and read into computer memory.
- The information in the file can be accessed in several ways.
- Some systems provide only one access method for files while others provide many access methods.
- A major design problem is choosing one among them.

1. Sequential Access

- Information in the file is processed in order, one record after the other.
- Sequential Access file operations
    - **read_next()** - Read next portion of file and automatically advances a file pointer.
    - **write_next()** – Appends to the end of the file and advances to the end of the newly written material.
    - **Reset** – Back to the beginning of file
    - no read after last write



2. Direct Access (Relative Accesses)

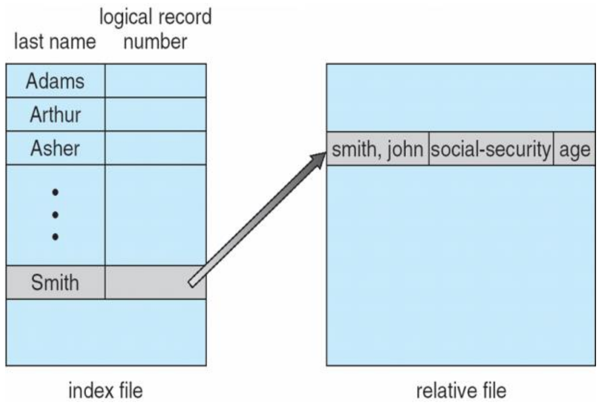
- A file is made up of fixed length logical records, that allow programs to read and write records rapidly in no particular order.
- For direct access disk is viewed as numbered sequence of blocks or record.

- ➢ Direct access file operations
  - ○ The file operations were modified to include the block number as a parameter.
  - ○ Read(*n*)
  - ○ Write(*n*)
  - ○ Position_file(n)

    n = relative block number

3. Other Access Methods

- ➢ Can be built on top of base methods
- ➢ General involve creation of an index for the file
- ➢ Keep index in memory for fast determination of location of data to be operated on (consider UPC code plus record of data about that item)
- ➢ If too large, index (in memory) of the index (on disk)
- ➢ To find a record, we first search the index and then use pointer to access the file directly and to find desired record.



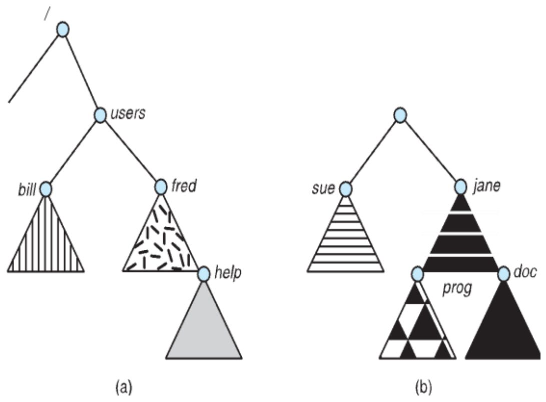
Example of index and relative files.

6.8 File System Mounting

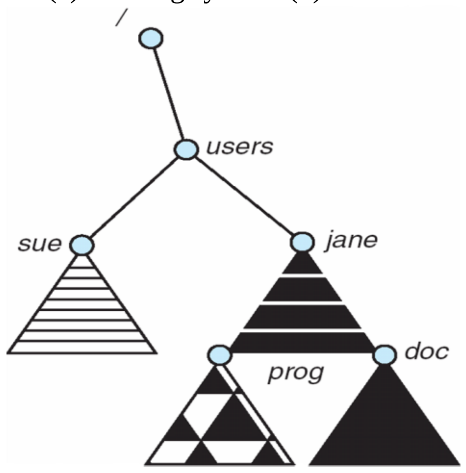
- ➢ A file system must be **mounted** before it can be accessed
- ➢ An unmounted filesystem is mounted at a mount point-the location within the file structure where the file system is to be attached.
- ➢ A mount point is an empty directory.
- ➢ The operating system verifies that the device contains a 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.
- ➢ The operating system notes in its directory structure that a filesystem is mounted

at the specified mount point.

- This scheme enables the operating system to traverse its directory structure, switching among file systems, and even file systems of varying types, as appropriate.



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



Mount point.

6.9 Protection

- When information is stored in a computer system, it must be keep safe from physical damage (the issue of reliability) and improper access (the issue of protection).
- Reliability is generally provided by duplicate copies of files.
- Protection can be provided in many ways.
  - Types of Access
  - Access Control
- Other Protection Approaches

1. Types of Access

- ➢ Access is permitted or denied depending on several factors, one of which is the type of access requested.
- ➢ Several different types of operations may be controlled:
  - ○ **Read:** Read from the file.
  - ○ **Write:** Write or rewrite the file.
  - ○ **Execute:** Load the file into memory and execute it.
  - ○ **Append:** Write new information at the end of the file.
  - ○ **Delete:** Delete the file and free its space for possible reuse.
  - ○ **List**: List the name and attributes of the file.

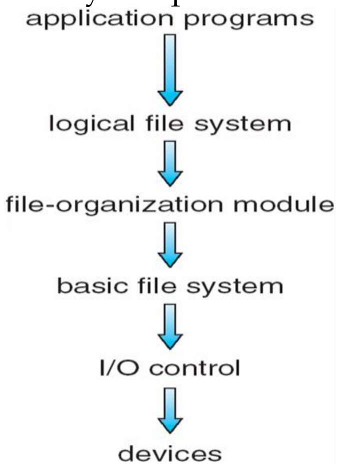
2. Access Control

- ➢ The most common approach to the protection problem is to make access dependent on the identity of the user.
- ➢ Different users may need different types of access to a file or directory.
- ➢ To implement dependent access is to associate with each file and directory an access control list (ACL)specifying user names and the types of access allowed for each user.
- ➢ Mode of access: read, write, execute
- ➢ Many systems recognize three classifications of users in connection with each file:
  - ○ **Owner:** The user who created the file is the owner.
  - ○ **Group**: A set of users who are sharing the file and need similar access is a group, or work group.
  - ○ **Universe:** All other users in the system constitute the universe.

6.10 File-System Structure

- ➢ Disks provide the bulk of secondary storage on which a file system is maintained.
- ➢ To improve I/0 efficiency, I/0 transfers between memory and disk are performed in units of blocks.Each block has one or more sectors.

- File systems provide efficient and convenient access to the disk by allowing data to be stored, located, and retrieved easily.
- A file system poses two quite different design problems
    - How the file system should look to the user
    - Creating algorithms and data structures to map the logical file system onto the physical secondary-storage devices.
- The file system itself is generally composed of many different levels



Layered file system.

- 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. I
- The basic file system needs only to issue generic commands to the appropriate device driver to read and write physical blocks on the disk.
- 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.
- the logical file system manages metadata information. Metadata includes all of the file-system structure except the actual data .
- A file-control block contains information about the file, including ownership, permissions, and location of the file contents.

6.11 File-System Implementation

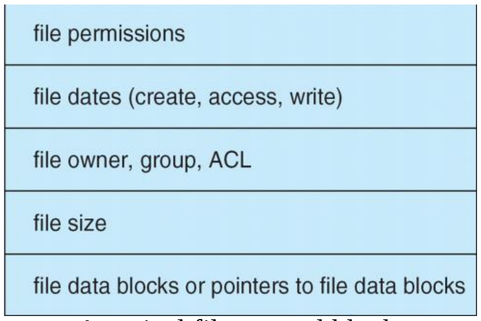
- Several on-disk and in-memory structures are used to implement a file system.
- The file system may contain information about how to boot an operating system

stored there, the total number of blocks, the number and location of free blocks, the directory structure, and individual files.

- Boot control block contains info needed by system to boot OS from that volume
  - Needed if volume contains OS, usually first block of volume
- Volume control block (superblock, master file table) contains volume details

Total number of blocks, number of free blocks, blocks size, free blocks pointers or array

- Directory structure organizes the files
  - Names and inode numbers, master file table
- Per-file File Control Block (FCB) contains many details about the file
  - inode number, permissions, size, dates
  - NFTS stores into in master file table using relational DB structures



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

A typical file-control block.

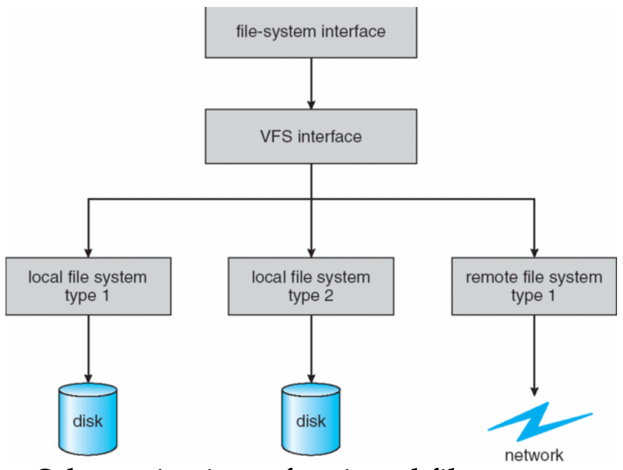
- The in-memory information is used for both file-system management and performance improvement via caching.
- The data are loaded at mount time, updated during file-system operations, and discarded at dismount.
- Several types of structures may be included.
  - An in-memory mount table contains information about each mounted volume.
  - An in-memory directory-structure cache holds the directory information of recently accessed directories.
  - The system wide open file table contains a copy of the FCB of each open file, as well as other information.
  - The per process open file table contains a pointer to the appropriate entry in the system-wide open-file table, as well as other information.
- Buffers hold file-system blocks when they are being read from disk or written to disk.

Partitions and Mounting

- Partition can be a volume containing a file system ("cooked") or raw – just a sequence of blocks with no file system
- Boot block can point to boot volume or boot loader set of blocks that contain enough code to know how to load the kernel from the file system
  - Or a boot management program for multi-os booting
- Root partition contains the OS, other partitions can hold other Oses, other file systems, or be raw
  - Mounted at boot time
  - Other partitions can mount automatically or manually
- At mount time, file system consistency checked
  - Is all metadata correct?
    - If not, fix it, try again
    - If yes, add to mount table, allow access

Virtual File Systems

- Virtual File Systems (VFS) on Unix provide an object-oriented way of implementing file systems
- VFS allows the same system call interface (the API) to be used for different types of file systems
  - Separates file-system generic operations from implementation details
  - Implementation can be one of many file systems types, or network file system
    - Implements **vnodes** which hold inodes or network file details
  - Then dispatches operation to appropriate file system implementation routines



Schematic view of a virtual file system.

6.12 Directory Implementation

- ➢ The selection of directory-allocation and directory-management algorithms significantly affects the efficiency, performance, and reliability of the file system

Linear List

- ➢ The simplest method of implementing a directory is to use a linear list of file names with pointers to the data blocks.
- ➢ This method is simple to program but time-consuming to execute.
- ➢ The real disadvantage of a linear list of directory entries is that finding a file requires a linear search.

Hash Table

- ➢ With this method, a linear list stores the directory entries, but a hash data structure is also used. The hash table takes a value computed from the file name and returns a pointer to the file name in the linear list.
- ➢ Therefore, it can greatly decrease the directory search time.
- ➢ Insertion and deletion are also fairly straightforward, although some provision must be made for collisions-situations in which two file names hash to the same location.
- ➢ The major difficulties with a hash table are its generally fixed size and the dependence of the hash function on that size.
- ➢ a chained-overflow hash table can be used.
- ➢ Each hash entry can be a linked list instead of an individual value, and we can resolve collisions by adding the new entry to the linked list.

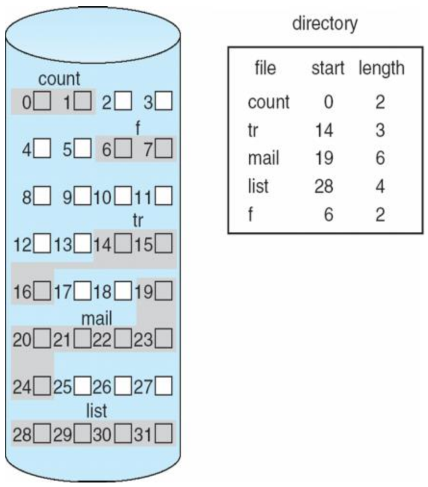
6.13 Allocation Methods

- ➢ An allocation method refers to how disk blocks are allocated for files.

1. Contiguous Allocation

- ➢ requires that each file occupy a set of contiguous blocks on disk.

➢ Disk addresses define a linear ordering on the disk.
➢ Best performance in most cases
➢ Simple – only starting location (block number) and length (number of blocks) are required
➢ Problems include finding space for file, knowing file size, external fragmentation, need for **compaction off-line (downtime) or on-line**



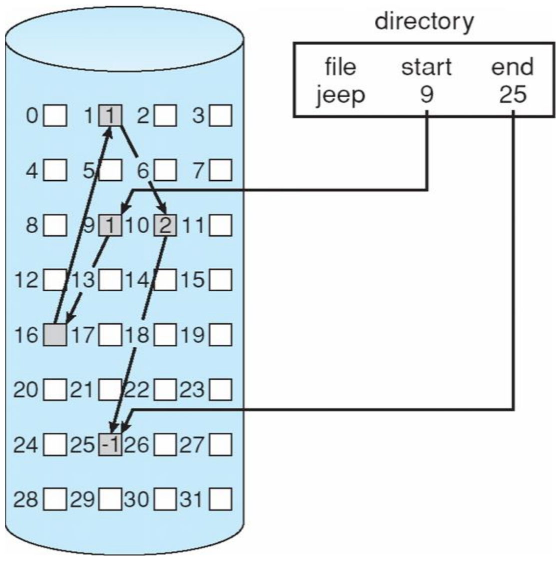
file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

2. Linked Allocation

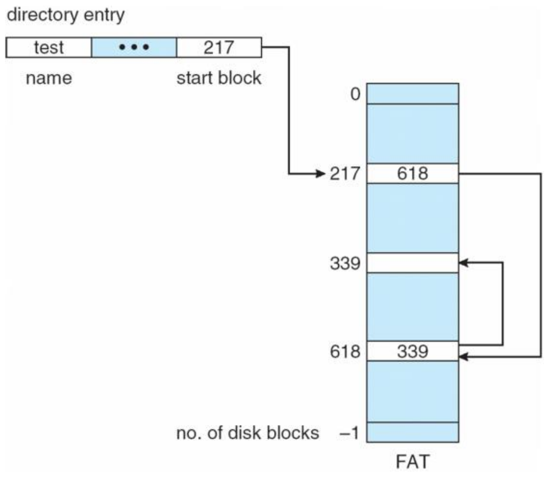
➢ With linked allocation, each file is a linked list of disk blocks; the disk blocks may be scattered anywhere on the disk.
➢ The directory points to the first and last blocks of the file.
➢ File ends at nil pointer
➢ No external fragmentation
➢ Each block contains pointer to next block
➢ No compaction, external fragmentation
➢ Free space management system called when new block needed
➢ Improve efficiency by clustering blocks into groups but increases internal fragmentation
➢ Reliability can be a problem
➢ Locating a block can take many I/Os and disk seeks

FAT (File Allocation Table) variation
➢ Beginning of volume has table, indexed by block number
➢ Much like a linked list, but faster on disk and cacheable

➢ New block allocation simple



Linked allocation of disk space.



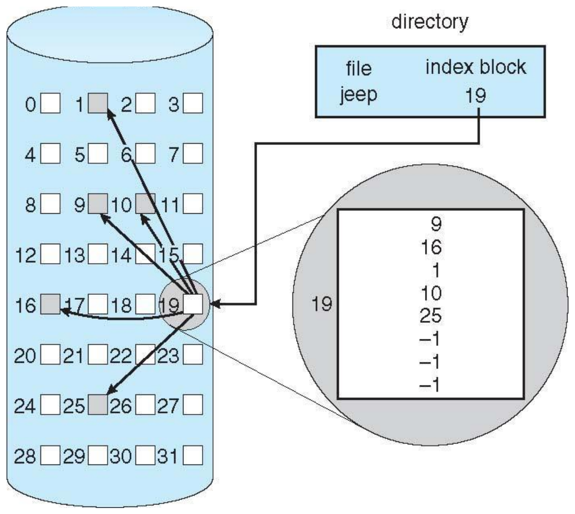
File-allocation table.

3. Indexed Allocation

➢ Linked allocation solves the external-fragmentation and size-declaration problems of contiguous allocation.

➢ Indexed allocation solves this problem by bringing all the pointers together into one location: the index block.

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



Indexed allocation of disk space.

- When the file is created, all pointers in the index block are set to null. When the i^th^ block is first written, a block is obtained from the free-space manager, and its address is put in the i^th^ index-block entry.
- Indexed allocation supports direct access, without suffering from external fragmentation, because any free block on the disk can satisfy a request for more space. Indexed allocation does suffer from wasted space, however.

6.14 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
- It ca be implemented as follows

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.
- 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.
- One technique for finding the first free block on a system that uses a bit-vector to allocate disk space is to sequentially check each word in the bit map to see whether that value is not 0, since a 0-valuedword contains only 0 bits and represents a set of allocated blocks.

Ex: 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 first non-0 word is scanned for the first 1 bit, which is the location of the first free block. The calculation of the block number is

  (number of bits per word) × (number of 0-value words) + offset of first 1 bit

- Unfortunately, bit vectors are inefficient unless the entire vector is kept in main memory (an is written to disk occasionally for recovery needs).
- Keeping it in main memory is possible for smaller disks but not necessarily for larger ones.

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.
- Recall our earlier example in which blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26, and 27 were free and the rest of the blocks were allocated.
- In this situation, we would keep a pointer to block 2 as the first free block. Block 2 would contain a pointer to block 3, which would point to block 4,

which would point to block 5, which would point to block 8, and so on.

- This scheme is not efficient; to traverse the list, we must read each block, which requires substantial I/O time.

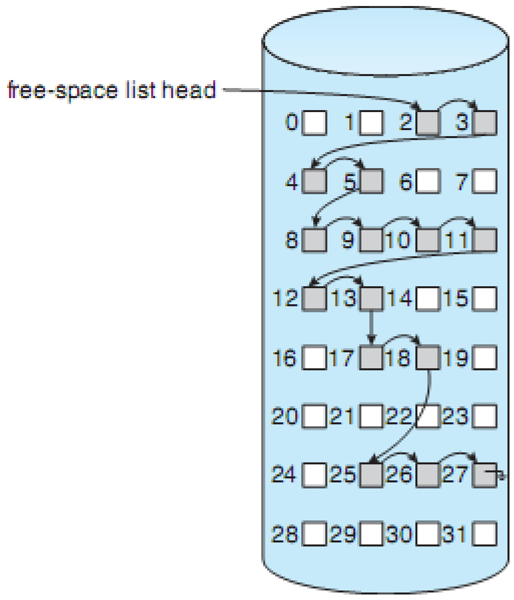


Figure 12.10 Linked free-space list on disk.

- 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 stores the addresses of n free blocks in the first free block.
- The first n−1 of these blocks are actually free. The last block contains the addresses of other 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

- Another approach takes advantage of the fact that, generally, several contiguous blocks may be allocated or freed simultaneously, particularly when space is allocated with the contiguous-allocation algorithm or through clustering.

- Thus, rather than keeping a list of n free disk addresses, we can keep the address of the first free block and the number (n) of free contiguous blocks that follow the first block.
- Each entry in the free-space list then consists of a disk address and a count.
- Although each entry requires more space than would a simple disk address, the overall list is shorter, as long as the count is generally greater than 1.
- Note that this method of tracking free space is similar to the extent method of allocating blocks.
- These entries can be stored in a balanced tree, rather than a linked list, for efficient lookup, insertion, and deletion.

5. Space Maps

- Oracle's ZFS file system (found in Solaris and other operating systems) was designed to encompass huge numbers of files, directories, and even file systems(in ZFS, we can create file-system hierarchies).
- On these scales, metadata I/O can have a large performance impact. Consider, for example, that if the free-space
- list is implemented as a bit map, bit maps must be modified both when blocks are allocated and when they are freed.
- Freeing 1 GB of data on a 1-TB disk could cause thousands of blocks of bitmaps to be updated, because those data blocks could be scattered over the entire disk.
- Clearly, the data structures for such a system could be large and inefficient.
- In its management of free space, ZFS uses a combination of techniques to control the size of data structures and minimize the I/O needed to manage those structures.
- First, ZFS creates Meta slabs to divide the space on the device into chunks of manageable size.
- A given volume may contain hundreds of meta slabs. Each meta slab has an associated space map.
- ZFS uses the counting algorithm to store information about free blocks. Rather than write countingstructures to disk, it uses log-structured file-system techniques to record them.

- The space map is a log of all block activity (allocating and freeing), in time order, in counting format.
- When ZFS decides to allocate or free space from a meta slab, it loads the associated space map into memory in a balanced-tree structure (for very efficient operation), indexed by offset, and replays the log into that structure.
- The in-memory space map is then an accurate representation of the allocated and free space in the meta slab.
- ZFS also condenses the map as much as possible by combining contiguous free blocks into a single entry.
- Finally, the free-space list is updated on disk as part of the transaction-oriented operations of ZFS.
- During the collection and sorting phase, block requests can still occur, and ZFS satisfies these requests from the log.
- In essence, the log plus the balanced tree is the free list.

6.15 Goals of Protection

- Protection mechanisms control access to a system by limiting the types of file access permitted to users. In addition, protection must ensure that only processes that have gained proper authorization from the operating system can operate on memory segments, the CPU, and other resources.
- Protection is provided by a mechanism that controls the access of programs, processes, or users to the resources defined by a computer system. This mechanism must provide a means for specifying the controls to be imposed, together with a means of enforcing them.
- Security ensures the authentication of system users to protect the integrity of the information stored in the system (both data and code), as well as the physical resources of the computer system. The security system prevents unauthorized access, malicious destruction or alteration of data, and accidental introduction of inconsistency.
- Untrustworthy users might safely share a common logical name space, such as a directory of files, or share a common physical name space, such as memory.
- Protection is necessary to
    1. Prevent the mischievous, intentional violation of an access restriction

by user.

2. To ensure that each program component active in a system uses system resources only in ways consistent with stated policies.
3. A protection-oriented system provides means to distinguish between authorized and unauthorized usage.
4. The role of protection in a computer system is to provide a mechanism for the enforcement of the policies governing resource use. A protection system must have the flexibility to enforce a variety of policies.

➢ 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.

6.16 Principles of Protection

➢ 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.

➢ The overflow of a buffer in a system daemon might cause the daemon process to fail 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

➢ The creation of audit trails for all privileged function access. The audit trail allows the programmer, system 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.

6.17 Domain of Protection

- A computer system is a collection of processes and objects. By objects, we mean both hardware objects (such as the CPU, memory segments, printers, disks, and tape drives) and software objects (such as files, programs, and semaphores). Each object has a unique name that differentiates it from all other objects in the system, and each can be accessed only through well-defined and meaningful operations. Objects are essentially abstract data types.
- The operations that are possible may depend on the object.
- A process should be allowed to access only those resources for which it has authorization.
- At any time, a process should be able to access only those resources that it currently requires to complete its task. This second requirement, commonly referred to as the need-to-know principle, is useful in limiting the amount of damage a faulty process can cause in the system.

1. Domain Structure

- A protection domain, which specifies the resources that the process may access. Each domain defines a set of objects and the types of operations that may be invoked on each object.
- The ability to execute an operation on an object is an accessright. A domain is a collection of access rights, each of which is an ordered pair <object-name, rights-set>.
- For example, if domain D has the access right <file F, {read,write}>, then a process executing in domain D can both read and write file F. It cannot, however, perform any other operation on that object.
- Domains may share access rights. Ex. we have three domains: D1, D2,and D3. The access right <O4, {print}> is shared by D2 and D3, implying that a process executing in either of these two domains can print object O4.
- The association between a process and a domain may be either static, if the set of resources available to the process is fixed throughout the process’s lifetime, or dynamic.
- If the association between processes and domains is fixed, and we want to adhere to the need-to-know principle, then a mechanism must be available to change the content of a domain.

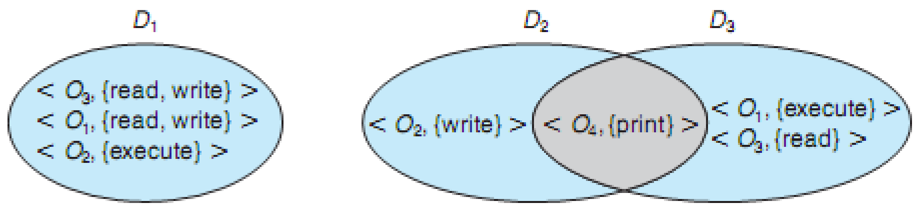


Figure 14.1 System with three protection domains.

- If the association is dynamic, a mechanism is available to allow domain switching, enabling the process to switch from one domain to another. We may also want to allow the content of a domain to be changed. If we cannot change the content of a domain, we can provide the same effect by creating a new domain with the changed content and switching to that new domain when we want to change the domain content.
- A domain can be realized in a variety of ways:
    1. Each user may be a domain.
    2. Each process may be a domain.
    3. Each procedure may be a domain.
- Consider the standard dual-mode (monitor–user mode) model of operating-system execution. When a process executes in monitor mode, it can execute privileged instructions and thus gain complete control of the computer system.
- In contrast, when a process executes in user mode, it can invoke only non privileged instructions. Consequently, it can execute only within its predefined memory space.
- These two modes protect the operating system (executing in monitor domain) from the user processes (executing in user domain).
- In a multi programmed operating system, two protection domains are insufficient, since users also want to be protected from one another.

2. An Example: UNIX

- In the UNIX operating system, a domain is associated with the user. Switching the domain corresponds to changing the user identification temporarily.
- This change is accomplished through the file system as follows.
    1. An owner identification and a domain bit (known as the setuid bit) are associated with each file.
    2. When the setuid bit is on, and a user executes that file, the userID is set to that of the owner of the file.

- Other methods are used to change domains in operating systems in which userIDs are used for domain definition, because almost all systems need to provide such a mechanism. This mechanism is used when an otherwise privileged facility needs to be made available to the general user population.
- An alternative to this method used in some other operating systems is to place privileged programs in a special directory.
- The operating system is designed to change the userID of any program run from this directory, either to the equivalent of root or to the userID of the owner of the directory. This eliminates one security problem, which occurs when intruders create programs to manipulate the setuid feature and hide the programs in the system for later use .This method is less flexible than that used in UNIX, however.
- Even more restrictive, and thus more protective, are systems that simply do not allow a change of userID. In these instances, special techniques must be used to allow users access to privileged facilities. For instance, a daemon process may be started at boot time and run as a special userID. Users then run a separate program, which sends requests to this process whenever they need to use the facility.

3. An Example: MULTICS

- In the MULTICS system, the protection domains are organized hierarchically into a ring structure.
- Each ring corresponds to a single domain.

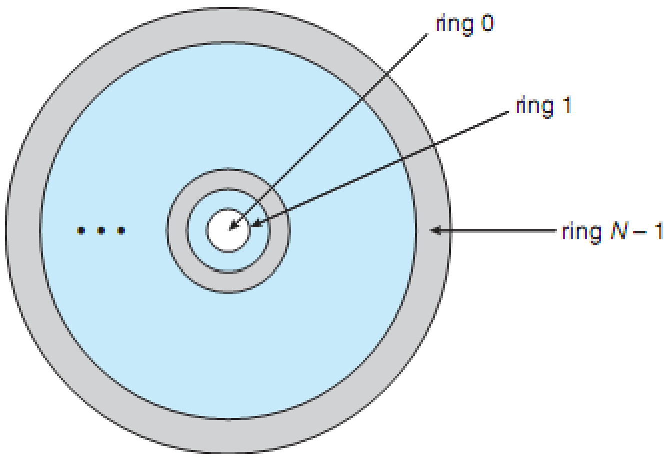


Figure 14.2 MULTICS ring structure.

- The rings are numbered from0to7. Let Di and Dj be any two domain rings. If j <i ,

then Di is a subset of Dj . That is, a process executing in domain Dj has more privileges than does a process executing in domain Di.

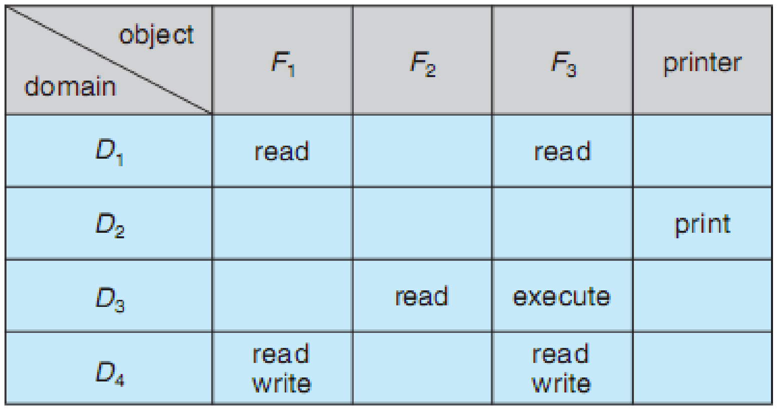
- A process executing in domain D0 has the most privileges. If only two rings exist, this scheme is equivalent to the monitor–user mode of execution, where monitor mode corresponds to D0 and user mode corresponds to D1.
- MULTICS has a segmented address space; each segment is a file, and each segment is associated with one of the rings. A segment description includes an entry that identifies the ring number.
- In addition, it includes three access bits to control reading, writing, and execution
- A current-ring-number counter is associated with each process, identifying the ring in which the process is executing currently. When a process is executing in ring i, it cannot access a segment associated with ring j ($j < i$). It can access a segment associated with ring k ($k \geq i$). The type of access, however, is restricted according to the access bits associated with that segment.
- Domain switching in MULTICS occurs when a process crosses from one ring to another by calling a procedure in a different ring. Obviously, this switch must be done in a controlled manner; otherwise, a process could start executing in ring 0, and no protection would be provided.
- To allow controlled domain switching, we modify the ring field of the segment descriptor to include the following:
    1. **Access bracket:** A pair of integers, b1 and b2,suchthat $b1 \leq b2$.
    2. **Limit:** An integer b3 such that $b3 > b2$.
    3. **List of gates:** Identifies the entry points (or gates) at which the segments may be called.
- If a process executing in ring i calls a procedure (or segment)with access bracket (b1,b2), then the call is allowed if $b1 \leq i \leq b2$, and the current ring number of the process remains i. Otherwise, a trap to the operating system occurs, and the situation is handled as follows:

1. If $i < b1$, then the call is allowed to occur, because we have a transfer to ring (or domain)with fewer privileges. However, if parameters are passed that refer to segments in a lower ring (that is, segments not accessible to the called procedure), then these segments must be copied into an area that can be accessed by the called procedure.
2. If $i > b2$, then the call is allowed to occur only if b3 is greater than or equal to i and the call has been directed to one of the designated entry points in the list of gates. This scheme allows processes with limited access rights to call procedures in lower rings that have more access rights, but only in a carefully controlled manner.

- he main disadvantage of the ring (or hierarchical) structure is that it does not allow us to enforce the need-to-know principle. In particular, if an object must be accessible in domain Dj but not accessible in domain Di,then we must have j < i. But this requirement means that every segment accessible in Di is also accessible in Dj.

6.18 Access Matrix

- Our general model of protection can be viewed abstractly as a matrix, called an access matrix.
- 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.
- Consider a sample Access Matrix



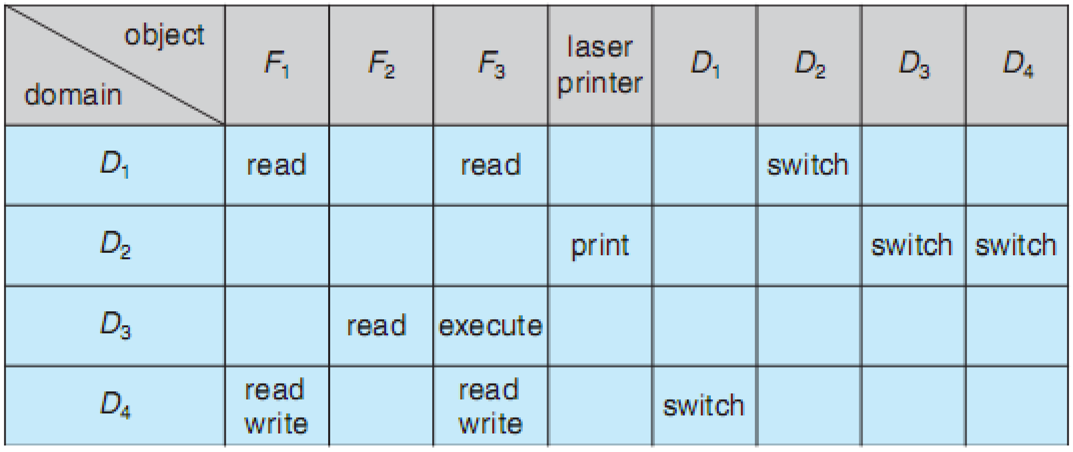
object / domain	F_1	F_2	F_3	printer
D_1	read		read	
D_2				print
D_3		read	execute	
D_4	read write		read write	

Figure 14.3 Access matrix.

- 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.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. The mechanism consists of implementing the access matrix and ensuring that the semantic properties we have outlined hold. More specifically, we must ensure that a process executing in domain Di can access

only those objects specified in row i, and then only as allowed by the access-matrix entries.

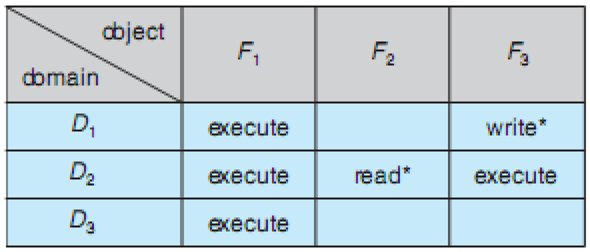
- The access matrix can implement policy decisions concerning protection.
- The access matrix provides an appropriate mechanism for defining and implementing strict control for both static and dynamic association between processes and domains.
- Processes should be able to switch from one domain to another. Switching from domain Di to domain Dj is allowed if and only if the access right switch ∈ access(i, j).
- In the following figure , 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.



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			

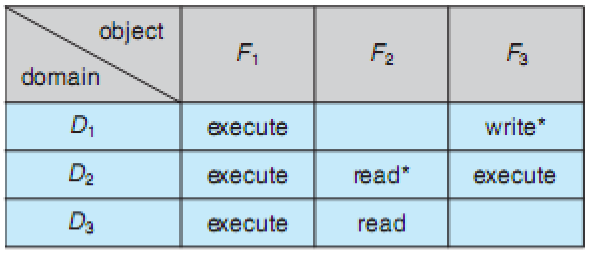
Figure 14.4 Access matrix of Figure 14.3 with domains as objects.

- Allowing controlled change in the contents of the access-matrix entries requires three additional operations: copy, owner, and control.
- 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.
- Example : In figure (a) a process executing in domain D2 can copy the read operation into any entry associated with file F2. Hence, the access matrix of Figure (a) can be modified to the access matrix in figure (b).



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

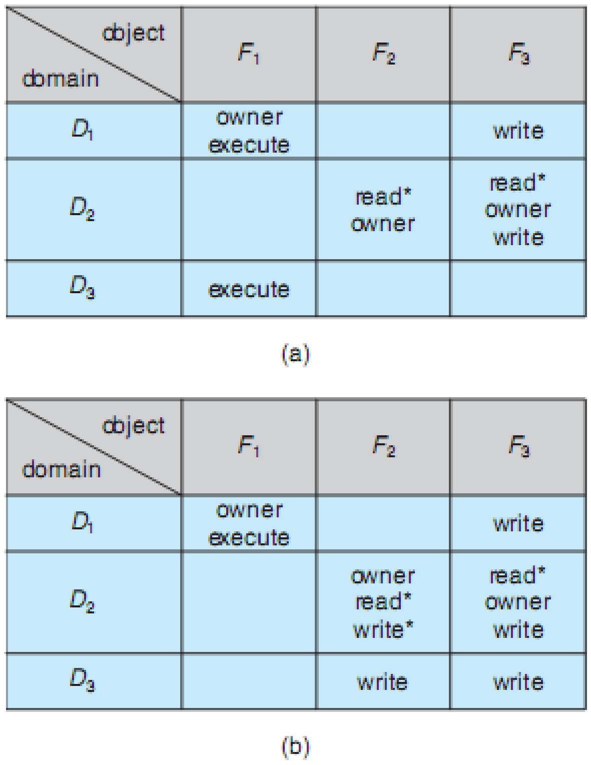
(a)



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 additional variants:
    1. A right is copied from access(i, j)to access(k, j); it is then removed from access(i, j). This action is a of a right, rather than a copy.
    2. 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.
- A system may select only one of these three copy rights, or it may provide all three by identifying them as separate rights: copy, transfer, and limited copy.
- The owner right mechanism allows addition and removal of new rights.
- If access(i, j) includes the owner right, then a process executing in domain Di can add and remove any right in any entry in column j.
- For example, in below Figure (a), domain D1 is the owner of F1 and thus can add and delete any valid right in column F1. Similarly, domain D2 is the owner of F2 and F3 and thus can add and remove any valid right within these two columns. Thus, the access matrix of Figure (a) can be modified to the access matrix shown in Figure (b).



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

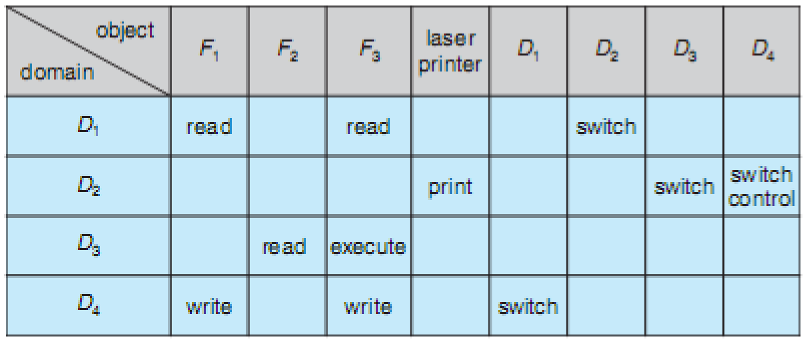
(a)

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

(b)

Figure 14.6 Access matrix with `owner` rights.

- The copy and owner rights allow a process to change the entries in a column. A mechanism is also needed to change the entries in a row. The control right is applicable only to domain objects.
- The copy and owner rights provide us with a mechanism to limit the propagation of access rights.However, they do not give us the appropriate tools for preventing the propagation (or disclosure) of information. The problem of guaranteeing that no information initially held in an object can migrate outside of its execution environment is called the **confinement problem.**
- These operations on the domains and the access matrix are not in them- selves important, but they illustrate the ability of the access-matrix model to allow us to implement and control dynamic protection requirement.



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 control
D_3		read	execute					
D_4	write		write		switch			

Figure 14.7 Modified access matrix of Figure 14.4.

- New objects and new domains can be created dynamically and included in the access-matrixmodel.