Module V
Secondary - Storage Structure

Magnetic disks provide a bulk of secondary storage. Disks come in various sizes and speed. Here the information is stored magnetically. Each disk platter has a flat circular shape like CD. The two surfaces of a platter are covered with a magnetic material. The surface of a platter is logically divided into circular **tracks,** which are subdivided into **sectors.** Sector is the basic unit of storage. The set of tracks that are at one arm position makes up a **cylinder.**

The number of cylinders in the disk dirve equals the number of tracks in each platter. There may be thousands of concentric cylinders in a disk drive, and each track may contain hundreds of sectors. The storage capacity of disk drives is measured in gigabytes.

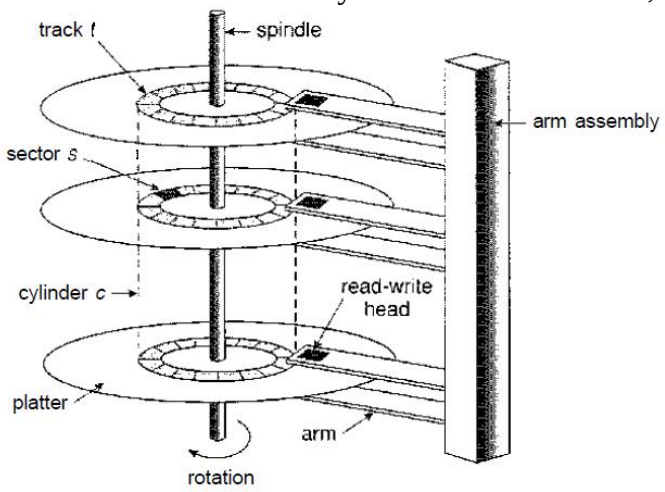


Figure 12.1 Moving-head disk mechanism.

The head moves from the inner track of the disk to the outer track. When the disk drive is operating the disks is rotating at a constant speed.

To read or write the head must be positioned at the desired track and at the beginning of the desired sector on that track.

- Seek Time:-Seek time is the time required to move the disk arm to the required track.
- Rotational Latency(Rotational Delay):-Rotational latency is the time taken for the disk to rotate so that the required sector comes under the r/w head.
- Positioning time or random access time is the summation of seek time and rotational delay.
- Disk Bandwidth:-Disk bandwidth is the total number of bytes transferred divided by total time between the first request for service and the completion of last transfer.
- Transfer rate is the rate at which data flow between the drive and the computer.

As the disk head flies on an extremely thin cushion of air, the head will make contact with the disk surface. Although the disk platters are coated with a thin protective layer, sometimes the head will damage the magnetic surface. This accident is called a **head crash.**

Magnetic Tapes

Magnetic tape is a secondary-storage medium. It is a permanent memory and can hold large quantities of data. The time taken to access data (access time) is large compared with that of magnetic disk, because here data is accessed sequentially. When the nth data has to be read, the tape starts moving from first and reaches the nth position and then data is read from nth position. It is not possible to directly move to the nth position. So tapes are used mainly for backup, for storage of infrequently used information.

DISK STRUCTURE

Each disk platter is divided into number of tracks and each track is divided into number of sectors. Sectors is the basic unit for read or write operation in the disk.

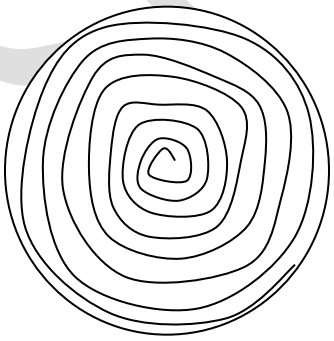
Modern disk drives are addressed as a large one-dimensional array. The one-dimensional array of logical blocks is mapped onto the sectors of the disk sequentially. Sector 0 is the first sector of the first track on the outermost cylinder. The mapping proceeds in order through that track, then through the rest of the tracks in that cylinder, and then through the rest of the cylinders from outermost to innermost.

The disk structure (architecture) can be of two types –

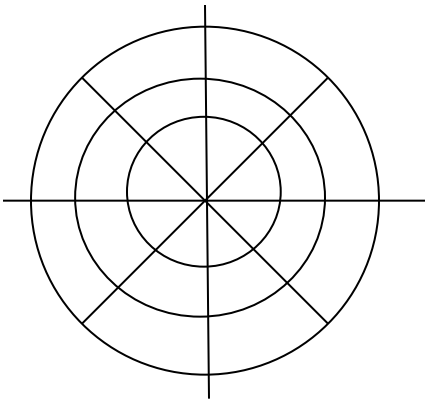
i) **Constant Linear Velocity** (CLV)
ii) **Constant Angular Velocity** (CAV)

i) 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. This architecture is used in CD-ROM and DVD-ROM.
ii) CAV – There is same number of sectors in each track. The sectors are densely packed in the inner tracks. The density of bits decreases from inner tracks to outer tracks to keep the data rate constant.

CLV



CAV



DISK ATTACHMENT

Computers can access data in two ways.

i) via I/O ports (or host-attached storage)
ii) via a remote host in a distributed file system(or network-attached storage)

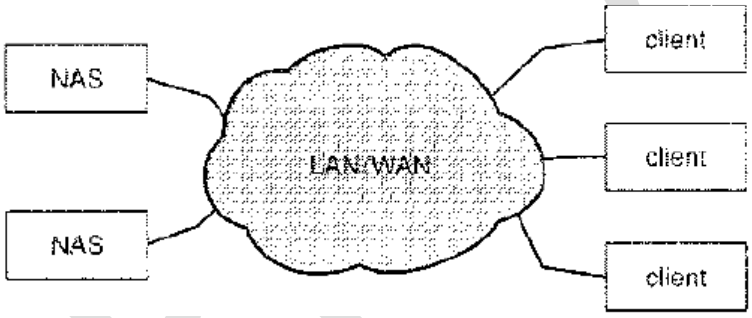
i) Host-Attached Storage

Host-attached storage is storage accessed through local I/O ports. Example : the typical desktop PC uses an I/O bus architecture called IDE or ATA. This architecture supports a maximum of two drives per I/O bus. The other cabling systems are – **SATA**(Serially Attached Technology Attachment), **SCSI**(Small Computer System Interface) and **fiber channel** (FC).

SCSI is a bus architecture. Its physical medium is usually a ribbon cable. FC is a high-speed serial architecture that can operate over optical fiber or over a four-conductor copper cable. An improved version of this architecture is the basis of **storage-area networks** (SANs).

ii) Network-Attached Storage

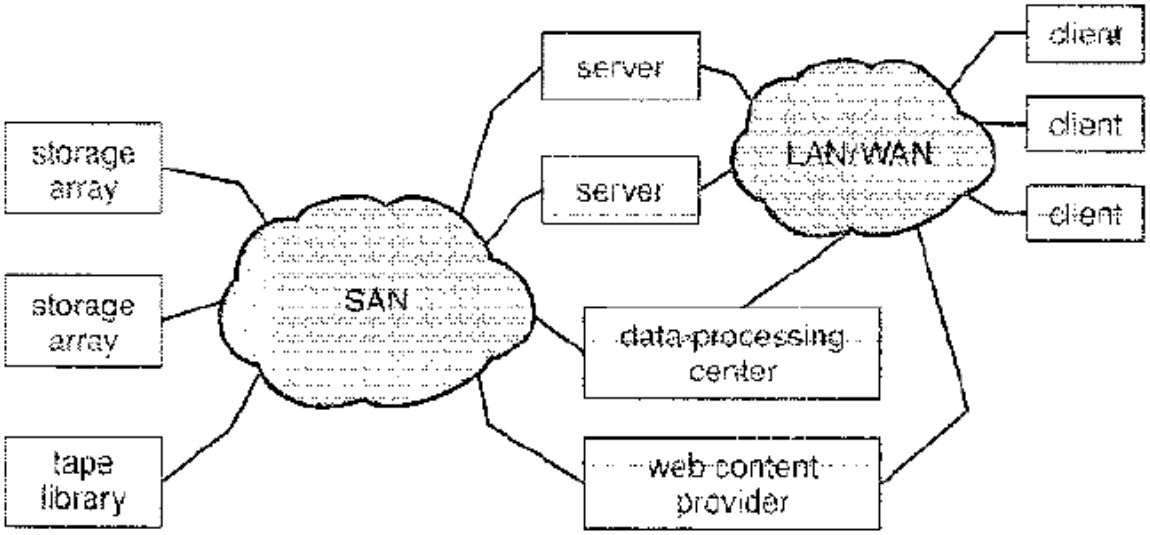
A network-attached storage (NAS) device is a special-purpose storage system that is accessed remotely over a network as shown in the figure. Clients access network-attached storage via a remote-procedure-call interface. The remote procedure calls (RPCs) are carried via TCP or UDP over an IP network—-usually the same local-area network (LAN) carries all data traffic to the clients.



Network-attached storage provides a convenient way for all the computers on a LAN to share a pool of storage files.

iii) Storage Area Network(SAN)

A storage-area network (SAN) is a private network connecting servers and storage units. The power of a SAN lies in its flexibility. Multiple hosts and multiple storage arrays can attach to the same SAN, and storage can be dynamically allocated to hosts. A SAN switch allows or prohibits access between the hosts and the storage. Fiber Chanel is the most common SAN interconnect.

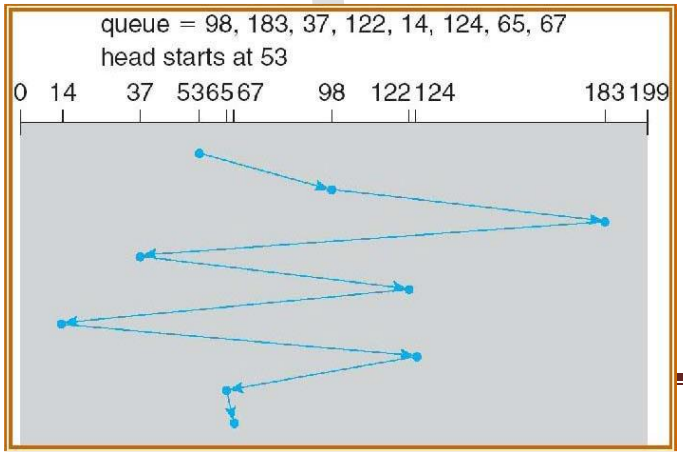


DISK SCHEDULING

Different types of disk scheduling algorithms are as follows:

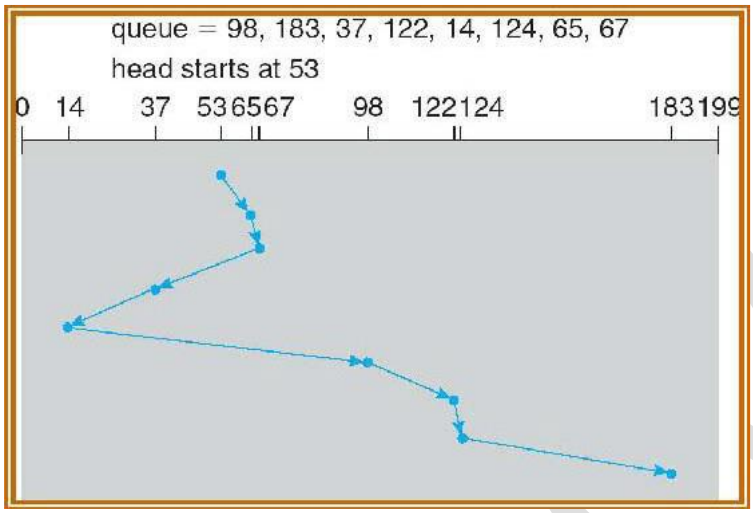
- FCFS (First Come First Serve)
- SSTF(Shortest Seek Time First)
- SCAN (Elecvator)
- C-SCAN
- LOOK
- C-LOOK

i) **FCFS scheduling algorithm**: This is the simplest form of disk scheduling algorithm. This services the request in the order they are received. This algorithm is fair but do not provide fastest service. It takes no special care to minimize the overall seek time. *Eg:*-consider a disk queue with request for i/o to blocks on cylinders. 98, 183, 37, 122, 14, 124, 65, 67



If the disk head is initially at 53, it will first move from 53 to 98 then to 183 and then to 37, 122, 14, 124, 65, 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.

ii) **SSTF** (Shortest Seek Time First) **algorithm**: This selects the request with minimum seek time from the current head position. SSTF chooses the pending request closest to the current head position. *Eg*:- consider a disk queue with request for i/o to blocks on cylinders. 98, 183, 37, 122, 14, 124, 65, 67

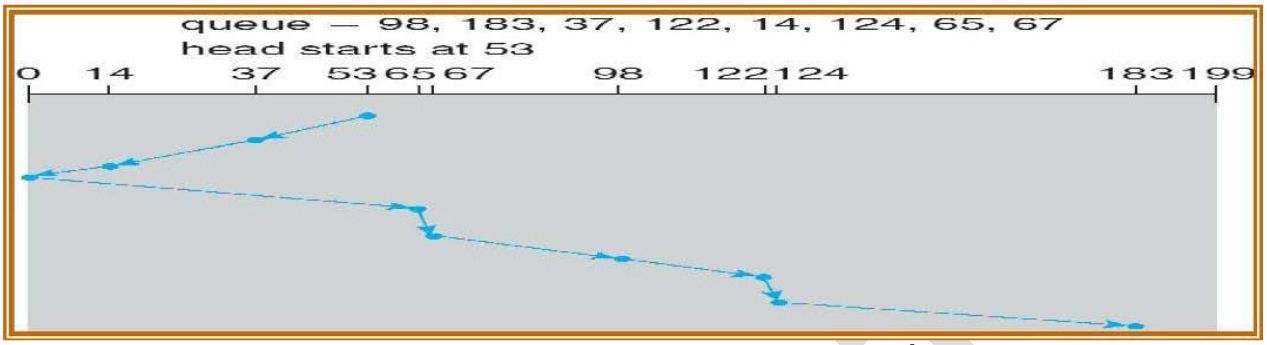


If the disk head is initially at 53, the closest is at cylinder 65, then 67, then 37 is closer then 98 to 67. So it services 37, continuing we service 14, 98, 122, 124 and finally 183. The total head movement is only 236 cylinders. SSTF is a substantial improvement over FCFS, it is not optimal.

iii) **SCAN algorithm**: In this the disk arm starts moving towards one end, servicing the request as it reaches each cylinder until it gets to the other end of the disk. At the other end, the direction of the head movement is reversed and servicing continues. The initial direction is chosen depending upon the direction of the head.

Eg:-*:*-consider a disk queue with request for i/o to blocks on cylinders. 98, 183, 37, 122, 14, 124, 65, 67

If the disk head is initially at 53 and if the head is moving towards the outer track, it services 65, 67, 98, 122, 124 and183. At cylinder 199 the arm will reverse and will move towards the other end of the disk servicing 37 and then 14. The SCAN is also called as **elevator** algorithm.



If the disk head is initially at 53 and if the head is moving towards **0th track**, it services 37 and then 14. At cylinder 0 the arm will reverse and will move towards the other end of the disk servicing 65, 67, 98, 122, 124 and 183.

iv) **C-SCAN** (Circular scan) **algorithm**:

C-SCAN is a variant of SCAN designed to provide a more uniform wait time. Like SCAN, C-SCAN moves the head from end of the disk to the other servicing the request along the way. When the head reaches the other end, it immediately returns to the beginning of the disk, without servicing any request on the return.

Eg:- consider a disk queue with request for i/o to blocks on cylinders. 98, 183, 37, 122, 14, 124, 65, 67

If the disk head is initially at 53 and if the head is moving towards the outer track, it services 65, 67, 98, 122, 124 and 183. At cylinder 199 the arm will reverse and will move immediately towards the other end of the disk, then changes the direction of head and serves 14 and then 37.

Note: If the disk head is initially at 53 and if the head is moving towards track 0, it services 37 and 14 first. At cylinder 0 the arm will reverse and will move immediately towards the other end of the disk servicing 65, 67, 98, 122, 124 and183.

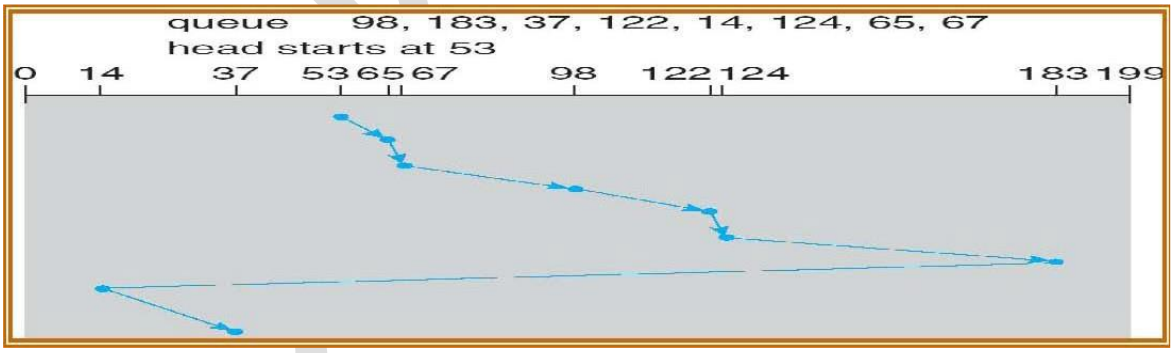
v) **Look Scheduling algorithm**:

Look and C-Look scheduling are different version of SCAN and C-SCAN respectively. Here the arm goes only as far as the final request in each direction. Then it reverses, without going all the way to the end of the disk. The Look and C-Look scheduling look for a request before continuing to move in a given direction.

Eg:-:-consider a disk queue with request for i/o to blocks on cylinders. 98, 183, 37, 122, 14, 124, 65, 67

If the disk head is initially at 53 and if the head is moving towards the outer track, it services 65, 67, 98, 122, 124 and183. At the final request 183, the arm will reverse and will move towards the first request 14 and then serves 37.

vi) C-Look Scheduling algorithm:



If the disk head is initially at 53 and if the head is moving towards the outer track, it services 65, 67, 98, 122, 124 and 183. At the last request, the arm will reverse and will move immediately towards the first request 14 and then serves 37.

Selection of a Disk-Scheduling Algorithm
SSTF is commonly used and it increases performance over FCFS.
SCAN and C-SCAN algorithm is better for a heavy load on disk.
SCAN and C-SCAN have less starvation problem.
Disk scheduling algorithm should be written as a separate module of the operating system.
SSTF or Look is a reasonable choice for a default algorithm.

SSTF is commonly used algorithms has it has a less seek time when compared with other algorithms. SCAN and C-SCAN perform better for systems with a heavy load on the disk, (ie. more read and write operations from disk).

Selection of disk scheduling algorithm is influenced by the file allocation method, if contiguous file allocation is choosen, then FCFS is best suitable, because the files are stored in contiguous blicks and there will be limited head movements required. A linked or indexed file, in contrast, may include blocks that are widely scattered on the disk, resulting in greater head movement.

The location of directories and index blocks is also important. Since every file must be opened to be used, and opening a file requires searching the directory structure, the directories will be accessed frequently. Suppose that a directory entry is on the first cylinder and a file's data are on the final cylinder. The disk head has to move the entire width of the disk. If the directory entry were on the middle cylinder, the head would have to move, at most, one-half the width. Caching the directories and index blocks in main memory can also help to reduce the disk-arm movement, particularly for read requests.

Because of these complexities, the disk-scheduling algorithm is very important and is written as a separate module of the operating system.

Disk Management

Disk Formatting
The process of dividing the disk into sectors and filling the disk with a special data structure is called **low-level formatting**. Sector is the smallest unit of area that is read/written by the disk controller. 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 a sector is read, the ECC is recalculated and is 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.

Most hard disks are low-level-forniatted at the factory as a part of the manufacturing process. This formatting enables the manufacturer to test the disk and to initialize the mapping from logical block numbers to defect-free sectors on the disk.

When the disk controller is instructed for low-level-formatting of the disk, the size of datablock of all sector sit can also be told how many bytes of data space to leave between the header and trailer of all sectors. It is of 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.

The operating system needs to record its own data structures on the disk. It does so in two steps. **partition** and **logical formatting.**
Partition – 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. For instance, one partition can hold a copy of the operating system's executable code, while another holds user files.
logical formatting (or creation of a file system) - Now, the operating system stores the initial file-system data structures onto the disk. These data structures may include maps of free and allocated space (a FAT or modes) and an initial empty directory.

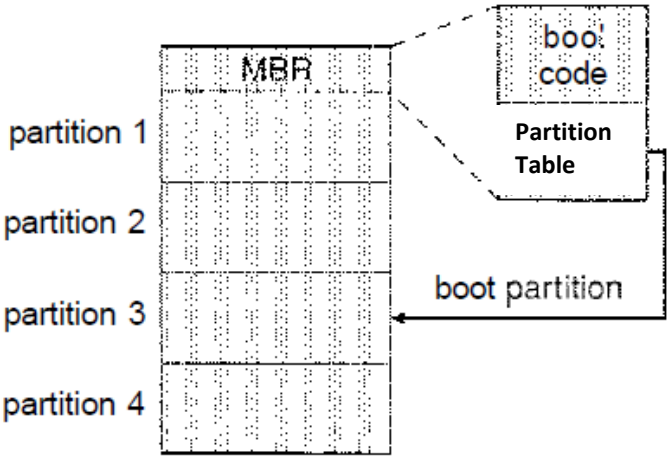
To increase efficiency, most file systems group blocks together into larger chunks, frequently called **clusters.**

Boot Block
When a computer is switched on or rebooted—it must have an initial program to run. This is called the *bootstrap* program. The bootstrap program –

- initializes the CPU registers, device controllers, main memory, and then starts the operating system.
- Locates and loads the operatingsystem from the disk
- jumps to beginning the operating-system execution.

The bootstrap is stored in read-only memory (ROM). Since ROM is read only, it cannot be infected by a computer virus. The problem is that changing this bootstrap code requires changing the ROM, hardware chips. So 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 can be changed easily: A new version is simply written onto the 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 Windows 2000 system places its boot code in the first sector on the hard disk (**master boot record,** or MBR). The 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.



Bad Blocks

Disk are prone to failure of sectors due to the fast movement of r/w head. Sometimes the whole disk will be changed. Such group of sectors that are defective are called as bad blocks.
Different ways to overcome bad blocks are -

- Some bad blocks are handled manually, eg. In MS-DOS.
- Some controllers replace each bad sector logically with one of the spare sectors(extra sectors). The schemes used are **sector sparing** or **forwarding** and **sector slipping**.

In MS-DOS format command, scans the disk to find bad blocks. If format finds a bad block, it writes a special value into the corresponding FAT entry to tell the allocation routines not to use that block.
In SCSI disks , bad blocks are found 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:

- The operating system tries to read logical block 87.
- The controller finds that the sector is bad. It reports this finding to the operating system.
- The next time the system is rebooted, a special, command is run to tell the SCSI controller to replace the bad sector with a spare.
- After that, whenever the system requests logical block 87, the request is translated into the replacement sector's (spare) address by the controller.

Some controllers replace bad blocks by **sector slipping**. Here is an example: Suppose that logical block 17 becomes defective and the first available spare follows sector 202. Then, sector slipping 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, and 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 sector 17 can be mapped to it.

SWAP SPACE MANAGEMENT

The amount of swap space needed on a system can vary 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. It can range from a few megabytes of disk space to gigabytes.

The swap space can overestimated or underestimated. It is safer to overestimate than to underestimate the amount of swap space required. If a system runs out of swap space due to underestimation of space, it may be forced to abort processes or may crash entirely. Overestimation wastes disk space that could otherwise be used for files, but it does no other harm.

Swap-Space Location

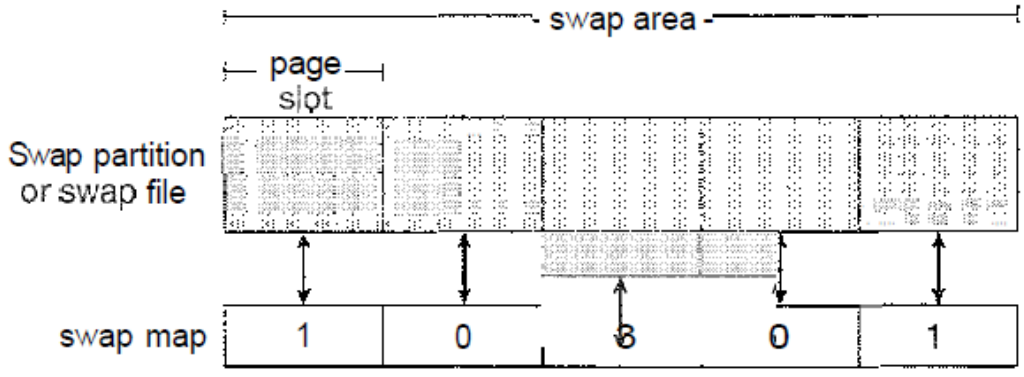
A swap space can reside in one of two places: It can be carved out of the normal **file system**, or it can be in a separate **disk partition**. If the swap space is simply a large file within the file system, normal file-system routines can be used to create it, name it, and allocate its space. External 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.

Alternatively, swap space can be created in a separate raw partition. A separate swap-space storage manager is used to allocate and deallocate the blocks from the raw partition.

Swap-Space Management: An Example

Solaris allocates swap space only when a page is forced out of physical memory, rather than when the virtual memory page is first created.

Linux is similar to Solaris in that swap space is only used for anonymous memory or for regions of memory shared by several processes. 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 raw 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. The data structures for swapping on Linux systems are shown in below figure.

PROTECTION:

GOALS OF PROTECTION

Protection is a mechanism for **controlling the access** of programs, processes, or users to the resources defined by a computer system. Protection ensures that only processes that have gained proper authorization from the operating system can operate on the files, memory segments, CPU, and other resources of a system.

Protection is required to prevent mischievous, intentional violation of an access restriction by a user.

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. An operating system provides mechanisms to enable privileges when they are needed and to disable them when they are not needed.

DOMAIN OF PROTECTION

A computer system is a collection of processes and objects. O*bjects* are 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 (resource) has a unique name that differentiates it from all other objects in the system.

The operations that are possible may depend on the object. For example, a CPU can only be executed on. Memory segments can be read and written, whereas a CD-ROM or DVD-ROM can only be read. Tape drives can be read, written, and rewound. Data files can be created, opened, read, written, closed, and deleted; program files can be read, written, executed, and deleted.

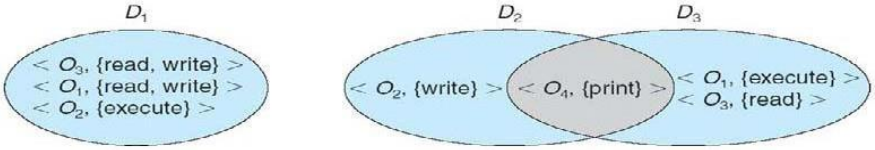
A process should be allowed to access only those resources

a) for which it has authorization
b) currently requires to complete process

Domain Structure

A domain is a set of objects and types of access to these objects. Each domain is an ordered pair of <object-name,rights-set>. Example, if domain D has the access right <*file F*, {read,write}>, then all process executing in domain D can both read and write file *F*, and cannot perform any other operation on that object.

Domains do not need to be disjoint; they may share access rights. For example, in below figure, we have three domains: D1 D2, and D3. The access right < *O4*, (print}> is shared by D2 and *D3,it* implies that a process executing in either of these two domains can print object O4.



A domain can be realized in different ways, it can be a user, process or a procedure.
ie. each user as a domain,
each process as a domain or
each procedure as a domain.

ACCESS MATRIX

Our model of protection can be viewed as a matrix, called an **access matrix.** It is a general model of protection that provides a mechanism for protection without imposing a particular protection policy. 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. The entry access(i,j) defines the set of operations that a process executing in domain *Di* can invoke on object *Oj*.

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	

In the above diagram, there are four domains and four objects—three files (F1, F2, F3) and one 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.
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 process executing in one domain and be **switched** to another domain. When we switch a process from one domain to another, we are executing an operation (**switch**) on an object (the domain). Domain switching from domain Di to domain *Dj* is allowed if and only if the access right switch € access(i,j). Thus, in the given 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 domain 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			

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
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	

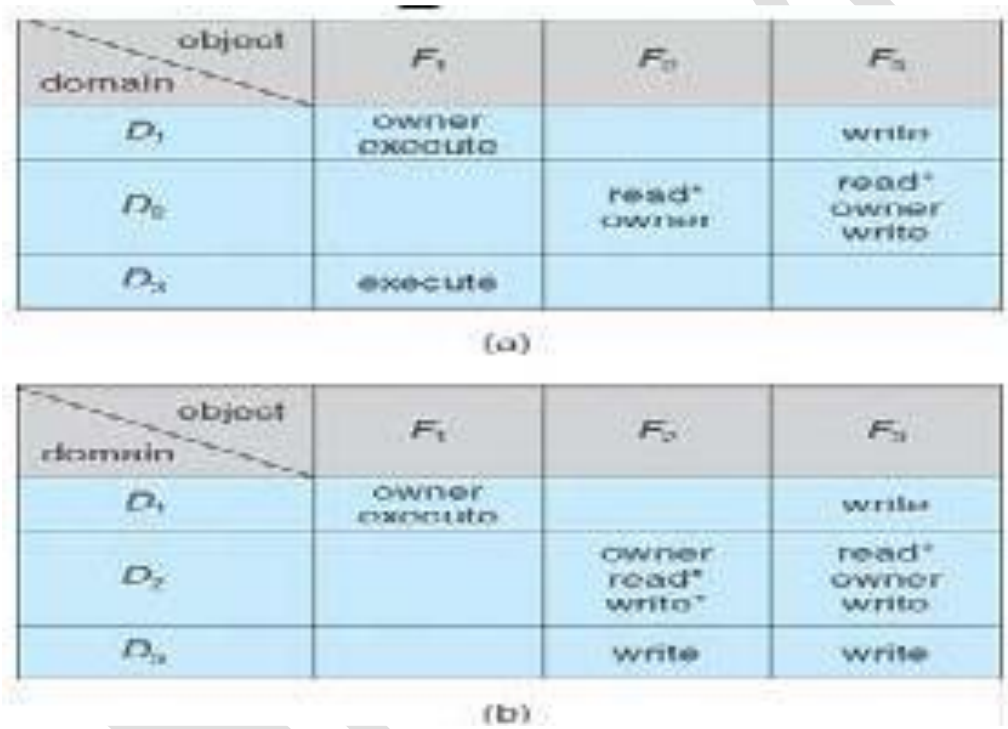
(b)

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 copying of the access right only within the column for which the right is defined. In the below figure, 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 shown in figure (b).

This scheme has two 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 *transfer* of a right, rather than a copy.
2) Propagation of the *copy* right- limited copy. Here, 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.*

We also need a mechanism to allow **addition** of new rights and **removal** of some rights. The ***owner*** right controls these operations. 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) as follows.



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)

A mechanism is also needed to change the entries in a row. If access(i,j) includes the ***control*** right, then a process executing in domain *Di,* can remove any access right from row j. For example, in figure, we include the *control* right in access(D3, D4). Then, a process executing in domain D3 can modify domain D4.

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					Control
D_4	read write		read write		switch			

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			

Implementation of Access Matrix

Different methods of implementing the access matrix (which is sparse).

- Global Table
- Access Lists for Objects
- Capability Lists for Domains
- Lock-Key Mechanism

Global Table

This is the simplest implementation of access matrix. A set of ordered triples <*domain, object*, *rights-set*> is maintained in a file. Whenever an operation *M* is executed on an object Oj, within domain *Di*, the table is searched for a triple <*Di*, Oj, *Rk*>. If this triple is found, the operation is allowed to continue; otherwise, an exception (or error) condition is raised.

Drawbacks -

The table is usually large and thus cannot be kept in main memory.

Additional I/O is needed

Access Lists for Objects

Each column in the access matrix can be implemented as an access list for one object. The empty entries are discarded. The resulting list for each object consists of ordered pairs *<domain, rights-set>*. It defines all domains access right for that object. When an operation M is executed on object Oj in Di, search the access list for object Oj, look for an entry <*Di*, Rj > with M e Kj. If the entry is found, we allow the operation; if it is not, we check the default set. If M is in the default set, we allow the access. Otherwise, access is denied, and an exception condition occurs. For efficiency, we may check the default set first and then search the access list.

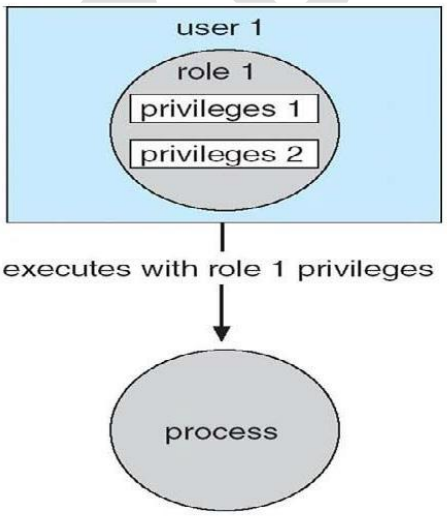
Capability Lists for Domains

A capability list for a domain is a list of objects together with the operations allowed on those objects. An object is often represented by its name or address, called a **capability**. To execute operation M on object Oj, the process executes the operation *M*, specifying the capability for object Oj as a parameter. Simple possession of the capability means that access is allowed.

Capabilities are usually distinguished from other data in one of two ways:
Each object has a tag to denote its type either as a capability or as accessible data.
Alternatively, the address space associated with a program can be split into two parts. One part is accessible to the program and contains the program's normal data and instructions. The other part, containing the capability list, is accessible only by the operating system.

A Lock-Key Mechanism

The lock-key scheme is a compromise between access lists and capability lists. Each object has a list of unique bit patterns, called locks. Similarly, each domain has a list of unique bit patterns, called keys. A process executing in a domain can access an object only if that domain has a key that matches one of the locks of the object.



Access Control

Each file and directory are assigned an owner, a group, or possibly a list of users, and for each of those entities, access-control information is assigned.

Solaris 10 advances the protection available in the Sun Microsystems operating system by explicitly adding the principle of least privilege via **role-based access control (RBAC).** This facility revolves around privileges. A privilege is the right to execute a system call or to use an option within that system call (such as opening a file with write access). Privileges can be assigned to processes, limiting them to exactly the access they need to

perform their work. Privileges and programs can also be assigned to **roles.** Users are assigned roles or can take roles based on passwords to the roles. In this way, a user can take a role that enables a privilege, allowing the user to run a program to accomplish a specific task, as depicted in Figure 14.8. This implementationof privileges decreases the security risk associated with superusers and setuid programs.

Revocation of Access Rights

Since the capabilities are distributed throughout the system, we must find them before we can revoke them. Schemes that implement revocation for capabilities include the following:

• **Reacquisition.** Periodically, all capabilities are deleted from each domain. If a process wants to use a capability, it may find that that capability has been deleted. The process may then try to reacquire the capability. If access has been revoked, the process will not be able to reacquire the capability.

• **Back-pointers.** A list of pointers is maintained with each object, pointing to all capabilities associated with that object. When revocation is required, we can follow these pointers, changing the capabilities as necessary.

• **Indirection**. The capabilities point indirectly to the objects. Each capability points to a unique entry in a global table, which in turn points to the object. We implement revocation by searching the global table for the desired entry and deleting it. Then, when an access is attempted, the capability is found to point to an illegal table entry.

• **Keys**. A key is a unique bit pattern that can be associated with a capability. This key is defined when the capability is created, and it can be neither modified nor inspected by the process owning the capability. A master key is associated with each object; it can be defined or replaced with the **set-key** operation. When a capability is created, the current value of the master key is associated with the capability. When the capability is exercised, its key is compared with the master key. If the keys match, the operation is allowed to continue; otherwise, an exception condition is raised.

In key-based schemes, the operations of defining keys, inserting them into lists, and deleting them from lists should not be available to all users.

CAPABILITY-BASED SYSTEM

Here, survey of two capability-based protection systems is done.

1) An Example: Hydra

Hydra is a capability-based protection system that provides considerable flexibility. A fixed set of possible access rights is known to and interpreted by the system. These rights include such basic forms of access as the right to read, write, or execute a memory segment. In addition, a user (of the protection system) can declare other rights.

Operations on objects are defined procedurally. The procedures that implement such operations are themselves a form of object, and they are accessed indirectly by capabilities. The names of user-defined procedures must be identified to the protection system if it is to deal with

objects of the user defined type. When the definition of an object is made known to Hydra, the names of operations on the type become auxiliary rights.

Hydra also provides rights amplification. This scheme allows a procedure to be certified as *trustworthy* to act on a formal parameter of a specified type on behalf of any process that holds a right to execute the procedure. The rights held by a trustworthy procedure are independent of, and may exceed, the rights held by the calling process.

When a user passes an object as an argument to a procedure, we may need to ensure that the procedure cannot modify the abject. We can implement this restriction readily by passing an access right that does not have the modification (write) right.

The procedure-call mechanism of Hydra was designed as a direct solution to the *problem of mutually suspicious subsystems.*

A Hydra subsystem is built on top of its protection kernel and may require protection of its own components. A subsystem interacts with the kernel through calls on a set of kernel-defined primitives that define access rights to resources defined by the subsystem.

2) An Example: Cambridge CAP System

A different approach to capability-based protection has been taken in the design of the Cambridge CAP system. CAP's capability system is simpler and superficially less powerful than that of Hydra. It can be used to provide secure protection of user-defined objects. CAP has two kinds of capabilities.

The ordinary kind is called a **data capability. It** can be used to provide access to objects, but the only rights provided are the standard read, write, and execute of the individual storage segments associated with the object.

The second kind of capability is the **software capability,** which is protected, but not interpreted, by the CAP microcode. It is interpreted by a *protected* (that is, a privileged) procedure, which may be written by an application programmer as part of a subsystem. A particular kind of rights amplification is associated with a protected procedure.

CASE STUDY: THE LINUX OPERATING SYSTEM

History

- Linux is a modem, free operating system based on UNIX standards.
- First developed as a small but self-contained kernel in 1991 by Linus Torvalds, with the major design goal of UNIX compatibility.
- Its history has been one of collaboration by many users from all around the world, corresponding almost exclusively over the Internet.
- It has been designed to run efficiently and reliably on common PC hardware, but also runs on a variety of other platforms.
- The core Linux operating system kernel is entirely original, but it can run much existing free UNIX software, resulting in an entire UNIX-compatible operating system free from proprietary code.

The Linux Kernel

- Version 0.01 (May 1991) had no networking, ran only on 80386-compatible Intel processors and on PC hardware, had extremely limited device-drive support, and supported only the Minix file system.
- Linux 1.0 (March 1994) included these new features:
  - Support for UNIX's standard TCP/IP networking protocols
  - BSD-compatible socket interface for networking programming
  - Device-driver support for running IP over an Ethernet
  - Enhanced file system
  - Support for a range of SCSI controllers for high-performance disk access
  - Extra hardware support
- Version 1.2 (March 1995) was the final PC-only Linux kernel.

Linux 2.0

- Released in June 1996, 2.0 added two major new capabilities:
  - Support for multiple architectures, including a fully 64-bit native Alpha port.

  - Support for multiprocessor architectures
- Other new features included:
  - Improved memory-management code
  - Improved TCP/IP performance
  - Support for internal kernel threads, for handling dependencies between loadable modules, and for automatic loading of modules on demand.
  - Standardized configuration interface
- Available for Motorola 68000-series processors, Sun Sparc systems, and for PC and PowerMac systems.

The Linux System

- Linux uses many tools developed as part of Berkeley's BSD operating system, MIT's X Window System, and the Free Software Foundation's GNU project.
- The min system libraries were started by the GNU project, with improvements provided by the Linux community.
- Linux networking-administration tools were derived from 4.3BSD code; recent BSD derivatives such as Free BSD have borrowed code from Linux in return.
- The Linux system is maintained by a loose network of developers collaborating over the Internet, with a small number of public ftp sites acting as de facto standard repositories.

Linux Distributions

- Standard, precompiled sets of packages, or *distributions*, include the basic Linux system, system installation and management utilities, and ready-to-install packages of common UNIX tools.
- The first distributions managed these packages by simply providing a means of unpacking all the files into the appropriate places; modern distributions include advanced package management.
- Early distributions included SLS and Slackware. *Red Hat* and *Debian* are popular distributions from commercial and noncommercial sources, respectively.

- The RPM Package file format permits compatibility among the various Linux distributions.

Linux Licensing

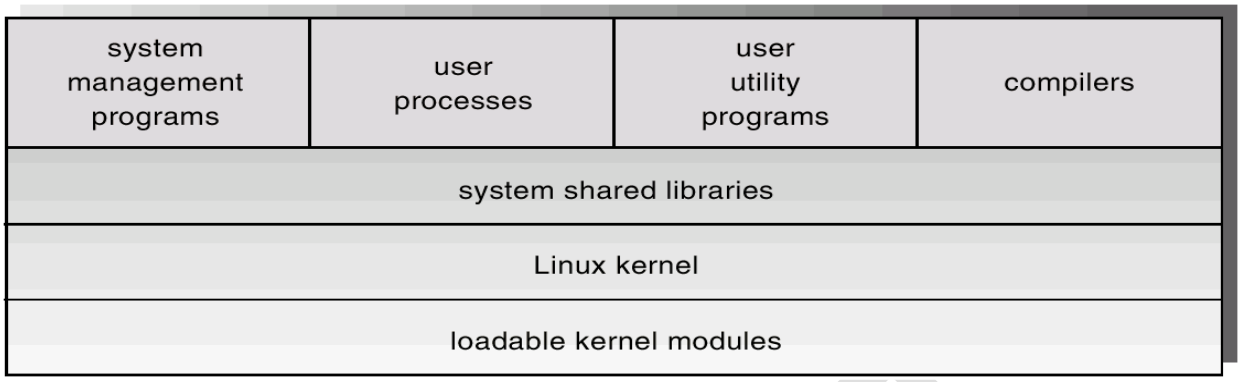
- The Linux kernel is distributed under the GNU General Public License (GPL), the terms of which are set out by the Free Software Foundation.
- Anyone using Linux, or creating their own derivative of Linux, may not make the derived product proprietary; software released under the GPL may not be redistributed as a binary-only product.

Design Principles

- Linux is a multiuser, multitasking system with a full set of UNIX-compatible tools..
- Its file system adheres to traditional UNIX semantics, and it fully implements the standard UNIX networking model.
- Main design goals are speed, efficiency, and standardization.
- Linux is designed to be compliant with the relevant POSIX documents; at least two Linux distributions have achieved official POSIX certification.
- The Linux programming interface adheres to the SVR4 UNIX semantics, rather than to BSD behavior

Components of a Linux System

- Like most UNIX implementations, Linux is composed of three main bodies of code; the most important distinction between the kernel and all other components.
- The **kernel** is responsible for maintaining the important abstractions of the operating system.
    - Kernel code executes in *kernel mode* with full access to all the physical resources of the computer.
    - All kernel code and data structures are kept in the same single address space.



- The **system libraries** define a standard set of functions through which applications interact with the kernel, and which implement much of the operating-system functionality that does not need the full privileges of kernel code.
- The **system utilities** perform individual specialized management tasks.

Kernel Modules

- Sections of kernel code that can be compiled, loaded, and unloaded independent of the rest of the kernel.
- A kernel module may typically implement a device driver, a file system, or a networking protocol.
- The module interface allows third parties to write and distribute, on their own terms, device drivers or file systems that could not be distributed under the GPL.
- Kernel modules allow a Linux system to be set up with a standard, minimal kernel, without any extra device drivers built in.
- Three components to Linux module support:
  - module management
  - driver registration
  - conflict resolution

Module Management

- Supports loading modules into memory and letting them talk to the rest of the kernel.
- Module loading is split into two separate sections:

  - Managing sections of module code in kernel memory
  - Handling symbols that modules are allowed to reference
- The module requestor manages loading requested, but currently unloaded, modules; it also regularly queries the kernel to see whether a dynamically loaded module is still in use, and will unload it when it is no longer actively needed.

Driver Registration

- Allows modules to tell the rest of the kernel that a new driver has become available.
- The kernel maintains dynamic tables of all known drivers, and provides a set of routines to allow drivers to be added to or removed from these tables at any time.
- Registration tables include the following items:
  - Device drivers
  - File systems
  - Network protocols
  - Binary format

Conflict Resolution

- A mechanism that allows different device drivers to reserve hardware resources and to protect those resources from accidental use by another driver
- The conflict resolution module aims to:
  - Prevent modules from clashing over access to hardware resources
  - Prevent *autoprobes* from interfering with existing device drivers
  - Resolve conflicts with multiple drivers trying to access the same hardware

Process Management

- UNIX process management separates the creation of processes and the running of a new program into two distinct operations.
  - The **fork** system call creates a new process.

- A new program is run after a call to **execve**.

- Under UNIX, a process encompasses all the information that the operating system must maintain *t* track the context of a single execution of a single program.
- Under Linux, process properties fall into three groups: the process's identity, environment, and context.

Process Identity

_ **Process ID** (PID). The unique identifier for the process; used to specify processes to the operating system when an application makes a system call to signal, modify, or wait for another process.

_ **Credentials**. Each process must have an associated user ID and one or more group IDs that determine the process's rights to access system resources and files.

_ **Personality**. Not traditionally found on UNIX systems, but under Linux each process has an associated personality identifier that can slightly modify the semantics of certain system calls. Used primarily by emulation libraries to request that system calls be compatible with certain specific flavors of UNIX.

Process Environment

- The process's environment is inherited from its parent, and is composed of two null-terminated vectors:
  - The argument vector lists the command-line arguments used to invoke the running program; conventionally starts with the name of the program itself
  - The environment vector is a list of "NAME=VALUE" pairs that associates named environment variables with arbitrary textual values.
- Passing environment variables among processes and inheriting variables by a process's children are flexible means of passing information to components of the usermode system software.

- The environment-variable mechanism provides a customization of the operating system that can be set on a per-process basis, rather than being configured for the system as a whole.

Process Context

- The (constantly changing) state of a running program at any point in time.
- The **scheduling context** is the most important part of the process context; it is the information that the scheduler needs to suspend and restart the process.
- The kernel maintains **accounting** information about the resources currently being consumed by each process, and the total resources consumed by the process in its lifetime so far.
- The **file table** is an array of pointers to kernel file structures. When making file I/O system calls, processes refer to files by their index into this table.
- Whereas the file table lists the existing open files, the **file-system context** applies to requests to open new files. The current root and default directories to be used for new file searches are stored here.
- The **signal-handler table** defines the routine in the process's address space to be called when specific signals arrive.
- The **virtual-memory context** of a process describes the full contents of the its private address space.

Processes and Threads

- Linux uses the same internal representation for processes and threads; a thread is simply a new process that happens to share the same address space as its parent.
- A distinction is only made when a new thread is created by the **clone** system call.
  - **fork** creates a new process with its own entirely new process context
  - **clone** creates a new process with its own identity, but that is allowed to share the data structures of its parent

- Using **clone** gives an application fine-grained control over exactly what is shared between two threads.

Scheduling

- The job of allocating CPU time to different tasks within an operating system.
- While scheduling is normally thought of as the running and interrupting of processes, in Linux, scheduling also includes the running of the various kernel tasks.
- Running kernel tasks encompasses both tasks that are requested by a running process and tasks that execute internally on behalf of a device driver.

Kernel Synchronization

- A request for kernel-mode execution can occur in two ways:
  - A running program may request an operating system service, either explicitly via a system call, or implicitly, for example, when a page fault occurs.
  - A device driver may deliver a hardware interrupt that causes the CPU to start executing a kernel-defined handler for that interrupt.
- Kernel synchronization requires a framework that will allow the kernel's critical sections to run without interruption by another critical section
- Linux uses two techniques to protect critical sections:

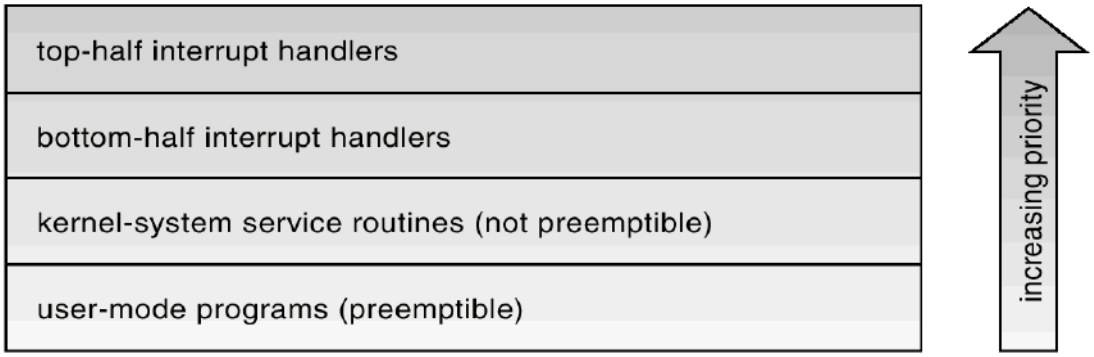
  1. Normal kernel code is nonpreemptible
     - ✓ when a time interrupt is received while a process is executing a kernel system service routine, the kernel's **need_resched** flag is set so that the scheduler will run once the system call has completed and control is about to be returned to user mode.

  2. The second technique applies to critical sections that occur in an interrupt service routines.
- By using the processor's interrupt control hardware to disable interrupts during a critical section, the kernel guarantees that it can proceed without the risk of concurrent access of shared data structures.

- To avoid performance penalties, Linux's kernel uses a synchronization architecture that allows long critical sections to run without having interrupts disabled for the critical section's entire duration.
- Interrupt service routines are separated into a *top half* and a *bottom half.*
  - The top half is a normal interrupt service routine, and runs with recursive interrupts disabled.
  - The bottom half is run, with all interrupts enabled, by a miniature scheduler that ensures that bottom halves never interrupt themselves.
  - This architecture is completed by a mechanism for disabling selected bottom halves while executing normal, foreground kernel code.

Interrupt Protection Levels

- Each level may be interrupted by code running at a higher level, but will never be interrupted by code running at the same or a lower level.
- User processes can always be preempted by another process when a time-sharing scheduling interrupt occurs.



Process Scheduling

- Linux uses two process-scheduling algorithms:
  - A time-sharing algorithm for fair preemptive scheduling between multiple processes
  - A real-time algorithm for tasks where absolute priorities are more important than fairness
- A process's scheduling class defines which algorithm to apply.

- For time-sharing processes, Linux uses a prioritized, credit based algorithm.
  - The crediting rule factors in both the process's history and its priority.
  - This crediting system automatically prioritizes interactive or I/O-bound processes.
- Linux implements the FIFO and round-robin real-time scheduling classes; in both cases, each process has a priority in addition to its scheduling class.
  - The scheduler runs the process with the highest priority; for equal-priority processes, it runs the longest-waiting one
  - FIFO processes continue to run until they either exit or block
  - A round-robin process will be preempted after a while and moved to the end of the scheduling queue, so that round robing processes of equal priority automatically time-share between themselves.

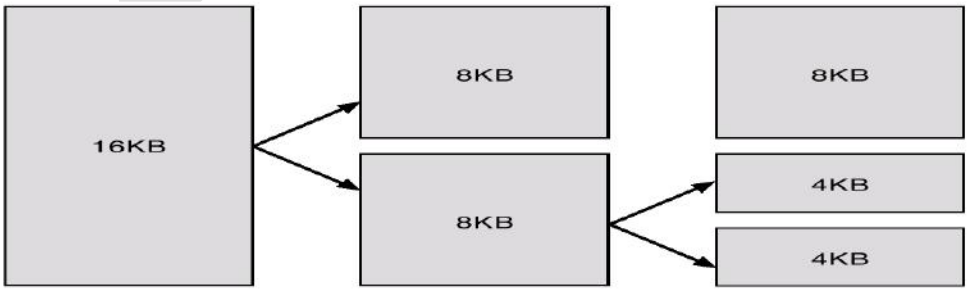
Symmetric Multiprocessing

- Linux 2.0 was the first Linux kernel to support SMP hardware; separate processes or threads can execute in parallel on separate processors.
- To preserve the kernel's nonpreemptible synchronization requirements, SMP imposes the restriction, via a single kernel spinlock, that only one processor at a time may execute kernel-mode code.

Memory Management

- Linux's physical memory-management system deals with allocating and freeing pages, groups of pages, and small blocks of memory.
- It has additional mechanisms for handling virtual memory, memory mapped into the address space of running processes.

Splitting of Memory in a Buddy Heap



Managing Physical Memory

- The page allocator allocates and frees all physical pages; it can allocate ranges of physically-contiguous pages on request.
- The allocator uses a *buddy-heap* algorithm to keep track of available physical pages.
  - Each allocatable memory region is paired with an adjacent partner.
  - Whenever two allocated partner regions are both freed up they are combined to form a larger region.
  - If a small memory request cannot be satisfied by allocating an existing small free region, then a larger free region will be subdivided into two partners to satisfy the request.
- Memory allocations in the Linux kernel occur either statically (drivers reserve a contiguous area of memory during system boot time) or dynamically (via the page allocator).

Virtual Memory

- The VM system maintains the address space visible to each process: It creates pages of virtual memory on demand, and manages the loading of those pages from disk or their swapping back out to disk as required.
- The VM manager maintains two separate views of a process's address space:
  - A logical view describing instructions concerning the layout of the address space. The address space consists of a set of nonoverlapping regions, each representing a continuous, page-aligned subset of the address space.
  - A physical view of each address space which is stored in the hardware page tables for the process.
- Virtual memory regions are characterized by:
  - The backing store, which describes from where the pages for a region come; regions are usually backed by a file or by nothing (*demand-zero* memory)
  - The region's reaction to writes (page sharing or copy-onwrite).

- The kernel creates a new virtual address space

  1. When a process runs a new program with the **exec** system call

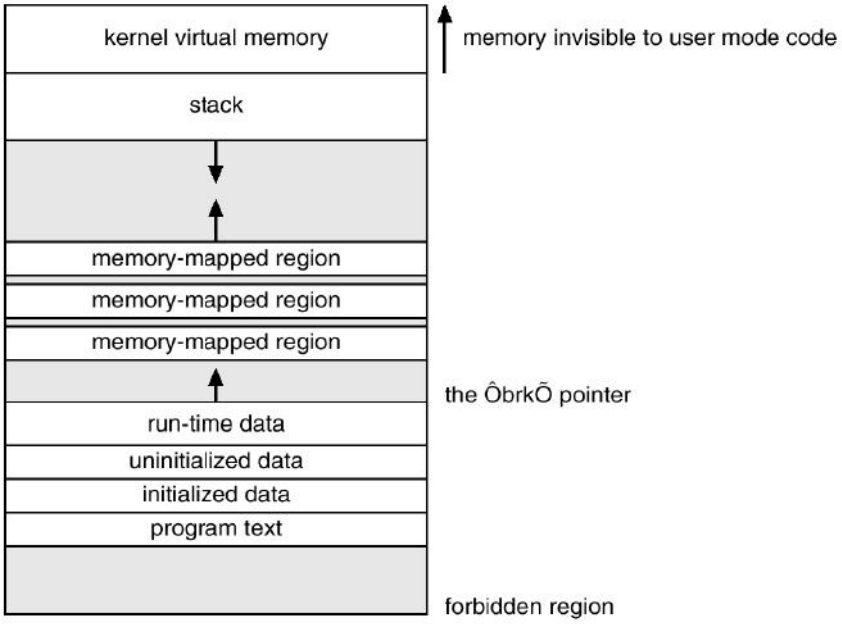
  2. Upon creation of a new process by the **fork** system call

- On executing a new program, the process is given a new, completely empty virtual-address space; the program loading routines populate the address space with virtual memory regions.
- Creating a new process with **fork** involves creating a complete copy of the existing process's virtual address space.
  - The kernel copies the parent process's VMA descriptors, then creates a new set of page tables for the child.
  - The parent's page tables are copies directly into the child's, with the reference count of each page covered being incremented.
  - After the fork, the parent and child share the same physical pages of memory in their address spaces.
- The VM paging system relocates pages of memory from physical memory out to disk when the memory is needed for something else.
- The VM paging system can be divided into two sections:
  - The pageout-policy algorithm decides which pages to write out to disk, and when.
  - The paging mechanism actually carries out the transfer, and pages data back into physical memory as needed.
- The Linux kernel reserves a constant, architecture dependent region of the virtual address space of every process for its own internal use.
- This kernel virtual-memory area contains two regions:
  - A static area that contains page table references to every available physical page of memory in the system, so that there is a simple translation from physical to virtual addresses when running kernel code.
  - The reminder of the reserved section is not reserved for any specific purpose; its page-table entries can be modified to point to any other areas of memory.

Executing and Loading User Programs

- Linux maintains a table of functions for loading programs; it gives each function the opportunity to try loading the given file when an exec system call is made.
- The registration of multiple loader routines allows Linux to support both the ELF and a.out binary formats.
- Initially, binary-file pages are mapped into virtual memory; only when a program tries to access a given page will a page fault result in that page being loaded into physical memory.
- An ELF-format binary file consists of a header followed by several page-aligned sections; the ELF loader works by reading the header and mapping the sections of the file into separate regions of virtual memory.

Memory Layout for ELF **Programs**



Static and Dynamic Linking

- A program whose necessary library functions are embedded directly in the program's executable binary file is *statically* linked to its libraries.
- The main disadvantage of static linkage is that every program generated must contain copies of exactly the same common system library functions.

- *Dynamic* linking is more efficient in terms of both physical memory and disk-space usage because it loads the system libraries into memory only once.

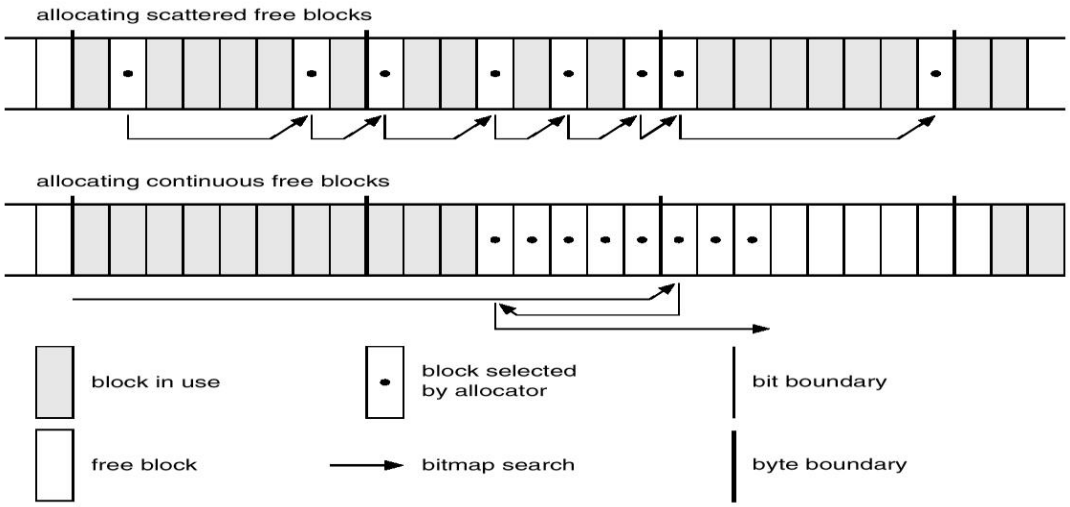
File Systems

- To the user, Linux's file system appears as a hierarchical directory tree obeying UNIX semantics.
- Internally, the kernel hides implementation details and manages the multiple different file systems via an abstraction layer, that is, the *virtual file system (VFS)*.
- The Linux VFS is designed around object-oriented principles and is composed of two components:
  - A set of definitions that define what a file object is allowed to look like
    - ✔ The *inode-object* and the *file-object* structures represent individual files
    - ✔ the *file system object* represents an entire file system
  - A layer of software to manipulate those objects.

The Linux Ext2fs File System

- Ext2fs uses a mechanism similar to that of BSD Fast File System (ffs) for locating data blocks belonging to a specific file.
- The main differences between ext2fs and ffs concern their disk allocation policies.
  - In ffs, the disk is allocated to files in blocks of 8Kb, with blocks being subdivided into fragments of 1Kb to store small files or partially filled blocks at the end of a file.
  - Ext2fs does not use fragments; it performs its allocations in smaller units. The default block size on ext2fs is 1Kb, although 2Kb and 4Kb blocks are also supported.
  - Ext2fs uses allocation policies designed to place logically adjacent blocks of a file into physically adjacent blocks on disk, so that it can submit an I/O request for several disk blocks as a single operation.

Ext2fs Block-Allocation Policies



The Linux Proc File System

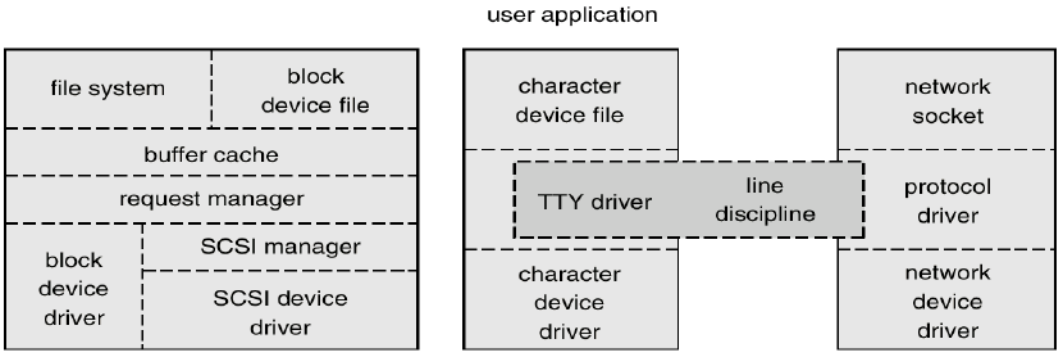
- The **proc** file system does not store data, rather, its contents are computed on demand according to user file I/O requests.
- **proc** must implement a directory structure, and the file contents within; it must then define a unique and persistent inode number for each directory and files it contains.
  - It uses this inode number to identify just what operation is required when a user tries to read from a particular file inode or perform a lookup in a particular directory inode.
  - When data is read from one of these files, **proc** collects the appropriate information, formats it into text form and places it into the requesting process's read buffer.

Input and Output

- The Linux device-oriented file system accesses disk storage through two caches:
  - Data is cached in the page cache, which is unified with the virtual memory system
  - Metadata is cached in the buffer cache, a separate cache indexed by the physical disk block.
- Linux splits all devices into three classes:
  - *block devices* allow random access to completely independent, fixed size blocks of data

- *character devices* include most other devices; they don't need to support the functionality of regular files.
- *network devices* are interfaced via the kernel's networking Subsystem

Device-driver Block Structure



Block Devices

- Provide the main interface to all disk devices in a system.
- The *block buffer* cache serves two main purposes:
  - it acts as a pool of buffers for active I/O
  - it serves as a cache for completed I/O
- The *request manager* manages the reading and writing of buffer contents to and from a block device driver.

Character Devices

- A device driver which does not offer random access to fixed blocks of data.
- A character device driver must register a set of functions which implement the driver's various file I/O operations.
- The kernel performs almost no preprocessing of a file read or write request to a character device, but simply passes on the request to the device.
- The main exception to this rule is the special subset of character device drivers which implement terminal devices, for which the kernel maintains a standard interface.

Interprocess Communication

Like UNIX, Linux informs processes that an event has occurred via signals.

- There is a limited number of signals, and they cannot carry information: Only the fact that a signal occurred is available to a process.

- The Linux kernel does not use signals to communicate with processes with are running in kernel mode, rather, communication within the kernel is accomplished via scheduling states and **wait.queue** structures.

Passing Data Between Processes

- The pipe mechanism allows a child process to inherit a communication channel to its parent, data written to one end of the pipe can be read a the other.
- Shared memory offers an extremely fast way of communicating; any data written by one process to a shared memory region can be read immediately by any other process that has mapped that region into its address space.
- To obtain synchronization, however, shared memory must be used in conjunction with another Interprocess communication mechanism.

Shared Memory Object

- The shared-memory object acts as a backing store for shared-memory regions in the same way as a file can act as backing store for a memory-mapped memory region.
- Shared-memory mappings direct page faults to map in pages from a persistent shared-memory object.
- Shared-memory objects remember their contents even if no processes are currently mapping them into virtual memory.

Network Structure

- Networking is a key area of functionality for Linux.
  - It supports the standard Internet protocols for UNIX to UNIX communications.
  - It also implements protocols native to nonUNIX operating systems, in particular, protocols used on PC networks, such as Appletalk and IPX.
- Internally, networking in the Linux kernel is implemented by three layers of software:
  - The socket interface
  - Protocol drivers
  - Network device drivers

- The most important set of protocols in the Linux networking system is the internet protocol suite.
  - It implements routing between different hosts anywhere on the network.
  - On top of the routing protocol are built the UDP, TCP and ICMP protocols

Security

- The *pluggable authentication modules (PAM)* system is available under Linux.
- PAM is based on a shared library that can be used by any system component that needs to authenticate users.
- Access control under UNIX systems, including Linux, is performed through the use of unique numeric identifiers (**uid** and **gid**).
- Access control is performed by assigning objects a *protections mask*, which specifies which access modes—read, write, or execute—are to be granted to processes with owner, group, or world access.
- Linux augments the standard UNIX **setuid** mechanism in two ways:
  - It implements the POSIX specification's saved *user-id* mechanism, which allows a process to repeatedly drop and reacquire its effective uid.
  - It has added a process characteristic that grants just a subset of the rights of the effective uid.
- Linux provides another mechanism that allows a client to selectively pass access to a single file to some server process without granting it any other privileges.

Device-driver Block Structure