Another approach to deletion is to preserve the file until all references to it are deleted. To implement this approach, we must have some mechanism for determining that the last reference to the file has been deleted. We could keep a list of all references to a file (directory entries or symbolic links). When a link or a copy of the directory entry is established, a new entry is added to the file-reference list. When a link or directory entry is deleted, we remove its entry on the list. The file is deleted when its file-reference list is empty.

The trouble with this approach is the variable and potentially large size of the file-reference list. However, we really do not need to keep the entire list - we need to keep only a count of the number of references. A new link or directory entry increments the reference counts; deleting a link or entry decrements the count. When the count is 0, the file can be deleted; there are no remaining references to it. The UNIX operating system uses this approach for non-symbolic links, or hard links, keeping a reference count in the file information block or inode). By effectively prohibiting multiple references to directories, we maintain an acyclic-graph structure.

To avoid these problems, some systems do not allow shared directories link. For example, in MS-DOS, the directory structure is a tree structure, rather than an acyclic graph, thereby avoiding the problems associated with file deletion in an acyclic-graph directory structure.

7.7.5 General Graph Directory

One serious problem with using an acyclic graph structure is ensuring that there are no cycles. If we start with a two-level directory and allow users to create subdirectories, a tree-structured directory results. It should be fairly easy to see that simply adding new files and subdirectories to an existing tree-structured preserves the tree-structured nature. However, when we add links to an existing tree-structured directory, the tree structure is destroyed, resulting in a simple graph structure.

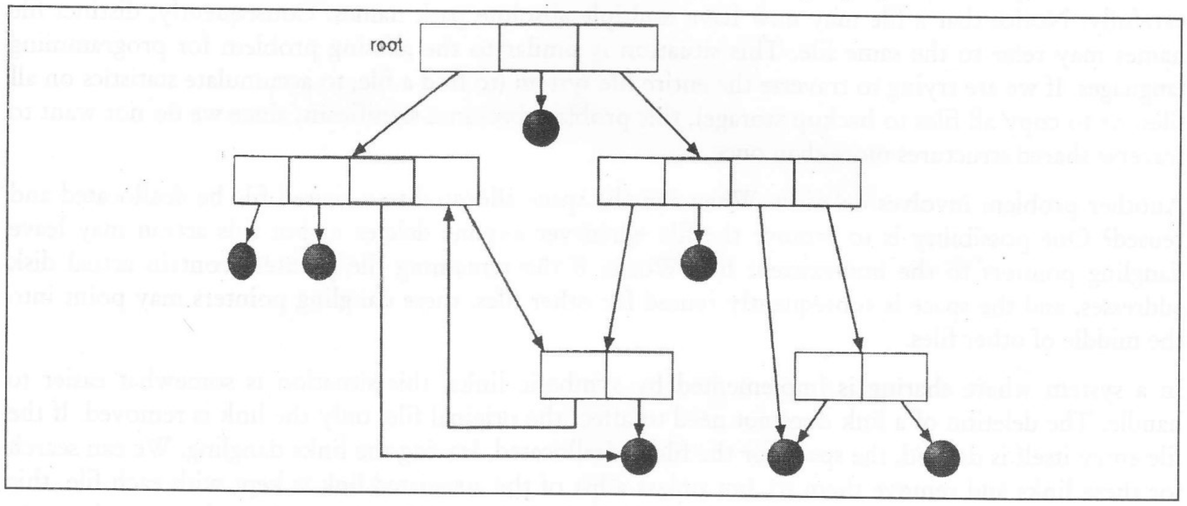

Figure 7.6: General Graph Directory

The primary advantage of an acyclic graph is the relative simplicity of the algorithms to traverse the graph and to determine when there are no more references to a file. We want to avoid traversing shared sections of an acyclic graph twice, mainly for performance reasons. If we have just searched a major shared subdirectory for a particular file, without finding that file, we want to avoid searching that subdirectory again; the second search would be wastage of time.

If cycles are allowed to exist in the directory, we likewise want to avoid searching any component twice, for reasons of correctness as well as performance. A poorly designed algorithm might result in an infinite loop continually searching through the cycle and never terminating. One solution is to arbitrarily limit the number of directories, which will be accessed during a search.

A similar problem exists when we are trying to determine when a file can be deleted. As with acyclic-graph directory structures, a value zero in the reference count means that there are no more references to the file or directory, and the file can be deleted. However, it is also possible, when cycles exist, that the reference count may be non-zero, even when it is no longer possible to refer to a directory or file. This anomaly results from the possibility of self-referencing (a cycle) in the directory structure. In this case, it is generally necessary to use a garbage collection scheme to determine when the last reference has been deleted and the disk space can be reallocated.

Garbage collection involves traversing the entire file system, marking everything that can be accessed. Then, a second pass collects everything that is not marked onto a list of free space. (A similar marking procedure can be used to ensure that a traversal or search will cover everything in the file system once and only once.) Garbage collection for a disk based file system, however, is extremely time-consuming and is thus seldom attempted. Garbage collection is necessary only because of possible cycles in the graph. Thus, an acyclic-graph structure is much easier to work with. The difficulty is to avoid cycles, as new links are added to the structure. How do we know when a new link will complete a cycle? There are algorithms to detect cycles in graphs however, they are computationally xpensive, especially when the graph is on disk storage. Generally, tree directory structures are more common than are acyclic-graph structures.

7.8 FILE ALLOCATIONS

The direct-access nature of disks allows flexibility in the implementation of files. In almost every case, many files will be stored on the same disk. The main problem is how to allocate space to these files so that disk space is utilized effectively and files can be accessed quickly. Three major methods of allocating disk space are in wide use: contiguous, linked, and indexed. Each method has its advantages and disadvantages. Accordingly, some systems (such as Data General's RDOS for its Nova line of computers) support all three. More common, a system will use one particular method for all files.

7.8.1 Contiguous space Allocation

The contiguous allocation method requires each file to occupy a set of contiguous blocks on the disk. Disk addresses define a linear ordering on the disk. With this ordering, assuming that only one job is accessing the disk, accessing block *b+1* after block b normally requires no head movement. When head movement is needed (from the last sector of one cylinder to the first sector of the next cylinder), it is only one track movement. Thus, the number of disk seeks required for accessing contiguously allocated files is minimal.

Contiguous allocation of the file is defined by the disk address and length (in block units) of the first block. If the file is n blocks long, and starts at location b, then it occupies block *b, b + 1, b + 2, ..., b + n-1.* The directory entry for each file indicates the address of the starting block and the length of the area allocated for this file.

Accessing a file that has been allocated contiguously is easy. For sequential access, the file system remembers the disk address of the last block referenced and, when necessary, reads the next block. For direct access to block i of a file that starts at block b, we can immediately access block $b + i$. Thus, both sequential and direct access can be supported by contiguous allocation.

One difficulty with contiguous allocation is finding space for a new file. The contiguous disk-space-allocation problem can be seen to be particular application of the general *dynamic storage-allocation* problem, which is how to satisfy a request of size n from a list of free holes. First-fit and best-fit are the most common strategies used to select a free hole from the set of available holes. Simulations have shown that both first-fit and best-fit are more efficient than worst-fit in terms of both time and storage utilization. Neither first-fit nor best-fit is clearly best in terms of storage utilization, but first-fit is generally faster.

These algorithms suffer from the problem of *external fragmentation.* As files are allocated and deleted, the free disk space is broken into little pieces. External fragmentation exists whenever free space is broken into chunks. It becomes a problem when the largest contiguous chunk is insufficient for a request; storage is fragmented into a number of holes, no one of which is large enough to store the data. Depending on the total amount of disk storage and the average file size, external fragmentation may be either a minor or a major problem.

Some older microcomputer systems used contiguous allocation on floppy disks. To prevent loss of significant amounts of disk space to external fragmentation, the user had to run a re-packing routine that copied the entire file system onto another floppy disk or onto a tape. The original floppy disk was then freed completely, creating one large contiguous free space. The routine then copied the files back onto the floppy disk by allocating contiguous space from this one large hole. The scheme effectively *compacts* all free space into one contiguous space, solving the fragmentation problem. The cost of this compaction is time. The time cost is particularly severe for large hard disks that use contiguous allocation, where compacting all the space may take hours and may be necessary on a weekly basis. During this *down time*, normal system operation generally cannot be permitted, so such compaction is avoided at all costs on production machines.

This is not all, there are other problems with contiguous allocation. A major problem is determining how much space is needed for a file. When the file is created, the total amount of space it will need must be found and allocated. How does the creator (program or person) know the size of the file to be created? In some cases, this determination may be fairly simple (copying an existing file, for example); in general, however, the size of an output file may be difficult to estimate.

If too little space is allocated to a file, it may be found that file cannot be extended. Especially with a best-fit allocation strategy, the space on both sides of the file may be in use. Hence, we cannot make the file larger in place. Two possibilities then exist. First, the user program can be terminated, with an appropriate error message. The user must then allocate more space and run the program again. These repeated runs may be costly. To prevent them, the user will normally over-estimate the amount of space needed, resulting in considerable wasted space.

The other possibility is to find a larger hole, to copy the contents of the file to the new space, and to release the previous space. This series of actions may be repeated as long as space exists, although it can also be time-consuming. Notice, however, that in this case the user never needs to be informed explicitly about what is happening; the system continues despite the problem, although more and more slowly.

Even if the total amount of space needed for a file is known in advance, pre-allocation may be inefficient. A file that grows slowly over a long period (months or years) must be allocated enough space for its final size, even though much of that space may be unused for a long time. The file, therefore, has a large amount of internal fragmentation.

To avoid several of these drawbacks, some operating systems use a modified contiguous allocation scheme, in which a contiguous chunk of space is allocated initially, and then, when that amount is not large enough, another chunk of contiguous space, called an extent, is added to the initial allocation. The location of a file's blocks is then recorded as a location and a block count, plus a link to the first block of the next extent. On some systems, the owner of the file can set the extent size, but this setting results in inefficiencies if the owner is incorrect. Internal fragmentation can still be a problem if the extents are too large, and external fragmentation can be a problem as extents of varying sizes are allocated and deallocated in turn.

7.8.2 Linked Allocation

Linked allocation solves all problems of contiguous allocation. With linked allocation, each file is a linked list of disk blocks; the disk blocks may be scattered anywhere on the disk. The directory contains a pointer to the first and last blocks of the time. For example, a file of five blocks might start at block 9, continue at block 16, then block 1, block 10, and finally block 25. Each block contains a pointer to the next block. These pointers are not made available to the user. Thus, if each block is 512 bytes, and a disk address (the pointer) requires 4 bytes, then the user sees blocks of 508 bytes.

To create a new file, we simply create a new entry in the directory. With linked allocation, each directory entry has a pointer to the first disk block of the file. This pointer is initialized to nil (the end-of-list pointer value) to signify an empty file. The size field is also set to 0. A write to the file causes a free block to be found via the free-space management system, and this new block is then written to, and is linked to the end of the file. To read a file, we simply read blocks by following the pointers from block to block.

There is no external fragmentation with linked allocation, and any free block on the free-space list can be used to satisfy a request. Notice also that there is no need to declare the size of a file when that file is created. A file can continue to grow as long as there are free blocks. Consequently, it is never necessary to compact disk space.

Linked allocation suffers from some disadvantages, however. The major problem is that it can be used effectively for only sequential-access files. To find the ith block of a file, we must start at the beginning of that file, and follow the pointers until we get to the ith block. Each access to a pointer requires a disk read, and sometimes a disk seek. Consequently, it is inefficient to support a direct-access capability for linked allocation files.

Space required for the pointers is another disadvantage to linked allocation. If a pointer requires 4 bytes out of a 512-byte block, then $((4 / 512) * 100 = 0.78)\%$ of the disk is being used for pointers, rather than for information. Each file requires slightly more space than it otherwise would.

The usual solution to this problem is to collect blocks into multiples, called clusters, and to allocate the clusters rather than blocks. For instance, the file system may define a cluster as 4 blocks, and operate on the disk in only cluster units. Pointers then use a much smaller percentage of the file's disk space. This method allows the logical-to-physical block mapping to remain simple, but improves disk throughput (fewer disk head-seeks) and decreases the space needed for block allocation and free-list management. The cost of this approach is an increase in internal fragmentation, because more space is wasted if a cluster is partially full than when a block is partially full. Clusters can be used to improve the disk access time for many other algorithms, so they are used in most operating systems.

Yet another problem is reliability. Since the files are linked together by pointers scattered all over the disk, consider what would happen if a pointer were lost or damaged. A bug in the operating-system software or a disk hardware failure might result in picking up the wrong pointer. This error could result in linking into the free-space list or into another file. Partial solutions are to use doubly linked lists or, to store the file name and relative block number in each block; however, these schemes require even more overhead for each file.

An important variation on the linked allocation method is the use of a file-allocation table (FAT). This simple but efficient method of disk-space allocation is used by the MSDOS and OS/2 operating systems. A section of disk at the beginning of each partition is set aside to contain the table. The table has one entry for each disk block, and is indexed by block number. The FAT is used much as is a linked list. The directory entry contains the block number of the first block of the file. The table entry indexed by that block number then contains the block number of the next block in the file. This chain continues until the last block, which has a special end-of-file value as the table entry. Unused blocks are indicated by a 0 table value. Allocating a new block to a file is a simple matter of finding the first 0-valued table entry, and replacing the previous end-of-file value.

Note that the FAT allocation scheme can result in a significant number of disk head seeks, unless the FAT is cached. The disk head must move to the start of the partition to read the FAT and find the location of the block in question, then move to the location of the block itself. In the worst case, both moves occur for each of the blocks. A benefit is that random access time is improved, because the disk head can find the location of any block by reading the information in the FAT.

7.8.3 Indexed Allocation

Linked allocation solves the external-fragmentation and size-declaration problems of contiguous allocation. However, in the absence of a FAT, linked allocation cannot support efficient direct access, since the pointers to the blocks are scattered with the blocks themselves all over the disk and need to be retrieved in order. Indexed allocation solves this problem by bringing all the pointers together into one location: the index block.

Each file has its own index block, which is an array of disk-block addresses. The ith entry in the index block points to the ith block of the file. The directory contains the address of the index block (Figure 7.7). To read the ith block, we use the pointer in the ith index-block entry to find and read the desired block.

When the file is created, all pointers in the index block are set to nil. When the ith block is first written, a block is obtained from the free-space manager, and its address is put in the ith index-block entry.

Indexed allocation supports direct access, without suffering from external fragmentation, because any free block on the disk may satisfy a request for more space.

Indexed allocation does suffer from wasted space. The pointer overhead of the index block is generally greater than the pointer overhead of linked allocation. Consider a common case in which we have a file of only one or two blocks. With linked allocation, we lose the space of only one pointer per block (one or two pointers).

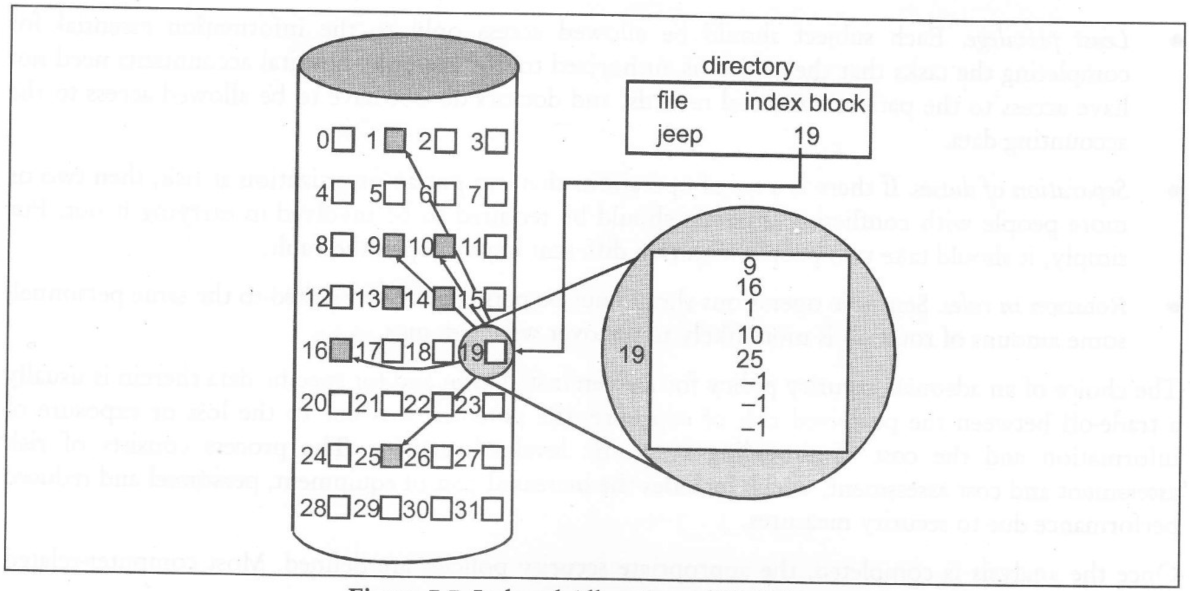

Figure 7.7: Indexed Allocation of Disk Space

With indexed allocation, an entire index block must be allocated, even if only one or two pointers will be non-nil.

This point raises the question of how large the index block should be. Every file must have an index block, so we want the index block to be as small as possible. If the index block is too small, however, it will not be able to hold enough pointers for a large file, and a mechanism will have to be available to deal with this issue.

7.9 SECURITY POLICIES AND MECHANISMS

Security policies specify what is desired in terms of protection and security. Security mechanisms specify how to affect the security policies and enforce them in a given system.

The primary objective of operating systems and of the systems software is to provide a flexible and functionally complete set of security mechanisms in order to enable users and owners of information to enforce security policies as they see it.

7.9.1 Security Policies

Security policies have probably been around since the accumulation of the first valuables that needed guarding. They usually encompass procedures and processes that specify:

1. How can information enter and exit the system?
2. Who is authorized to access what information and under what conditions?
3. What are the permissible flows of information within the system?

Additional limitations, such as restricting database queries about too large or too small sets, can be imposed to reduce the danger of deducing data by statistical inference. Security policies are often guided by the age-old principles of:

- *Least privilege.* Each subject should be allowed access only to the information essential for completing the tasks that the subject is authorized to, for example, hospital accountants need not have access to the patient's medical records, and doctors do not have to be allowed access to the accounting data.
- *Separation of duties.* If there is a set of operations that can put an organization at risk, then two or more people with conflicting interests should be required to be involved in carrying it out. Put simply, it should take two people with two different keys to open the vault.
- *Rotation in roles.* Sensitive operations should not be permanently entrusted to the same personnel; some amount of rotation is more likely to uncover wrong doings.

The choice of an adequate security policy for a given installation and for specific data therein is usually a trade-off between the perceived risk of exposure, the potential loss due to the loss or exposure of information and the cost of providing a specific level of security. The process consists of risk assessment and cost assessment, which includes the increased cost of equipment, personnel and reduced performance due to security measures.

Once the analysis is completed, the appropriate security policies are defined. Most computer-related security policies belong to one of the two basic categories;

- *Discretionary Access Control* (DAC). Policies are usually defined by the owner of data, who may pass access rights to other users. Usually, the creator of a file can specify the access rights of other users. This form of access control is common in file systems. It is vulnerable to the Trojan-horse attack, where intruders pass themselves off as legitimate users.
- *Mandatory Access Control* (MAC). Mandatory access restrictions are not subject to user discretion and thus limit the damage that the trojan horse can cause. In this scheme, users are classified according to level of authority or clearance. Data are classified into security classes according to level of confidentiality, and strict rules are defined regarding which level of user clearance is required for accessing the data of a specific security class. For example, military documents are categorized as unclassified, confidential, secret, and top secret. The user is required to have clearance equal to or above that of the document in order to access it. MAC also appears in other systems in perhaps less obvious forms. For example, university administrators cannot pass the right to access grade records to students. Security policies that address both external and internal threats are very important in environments that handle sensitive data, because most misuses are originated by insiders.

7.9.2 Security Mechanisms and Design Principles

In general, security measures include control and monitoring of physical access to the computer premises as well as the internal computer-system security. External or physical security includes the standard techniques of fencing, surveillance, authentication and attendance monitoring. Additional access restrictions may be imposed in special areas, such as the computer center and storage areas for backup volumes. Physical security may also include measures for disaster recovery, which often amount to replication of critical data and/or equipment at geographically dispersed locations to minimize exposure to the consequences of disasters such as fire or flood.

In this chapter, we concentrate on the issues of primary concern to operating system designers, that is, on the internal security mechanisms that provide the foundation for implementation of security policies. Saltzer and Schroeder (1975) have identified the following general design principles for protection mechanisms:

- *Least privilege:* Every subject should use the least set of privileges necessary to complete its task. This principle limits the damage from Trojan-horse attacks. It effectively advocates support for small protection domains and switching of domains when the access needs change.
- *Separation of privilege:* When possible, access to objects should depend on satisfying more than one condition (i.e., two keys to open the vault).
- *Least common mechanism:* This approach advocates minimizing the amount of mechanism common to and depended upon by multiple users. Design implications include the incorporation of techniques for separating users, such as logical separation via virtual machines and physical separation on different machines in distributed systems.
- *Economy of mechanism:* Keeping the design as simple as possible facilitates verification and correct implementations.
- *Complete mediation:* Every access request for every object should be checked for authorization. The checking mechanism should be efficient because it has a profound influence on system performance.
- *Fail-safe default:* Access rights should be acquired by explicit permission only, and the default should be lack of access.
- *Open design:* The design of the security mechanism should not be secret, and it should not depend on the ignorance of attackers. This implies the use of cryptographic systems where the algorithms are known but the keys are secret.
- *User acceptability:* The mechanism should provide ease of use so that it is applied correctly and not circumvented by users. Computer-system security mechanisms include authentication, access control, flow control, auditing and cryptography. They are presented in the remainder of this chapter.

7.10 PROTECTION AND ACCESS CONTROL

The original motivation for protection mechanisms came with the advent of multiprogramming. The intent was to confine each user's program to its assigned area of memory and thus, prevent programs from trespassing and harming each other. With the increased desire for sharing of objects in primary and secondary memory, more complex mechanisms for access control were devised.

7.10.1 Protection in Computer System

Protection in primary storage is usually adjunct to address translation. Its objective is to allow concurrent and potentially mutually suspicious resident process to share the common physical address space, primary memory. In systems with contiguous allocation of memory, protection is usually accomplished with the aid of some sort of limit registers. When the program is loaded, the limit or the bound registers are set to delineate the extent of its legitimate address space. At run time, each memory reference is prechecked to verify that it is within the bounds. Otherwise, access to memory is denied, and an exception is raised to activate the protection mechanism. Protection is ensured by making modification of the limit registers, a privileged operation that can be executed only when the machine is running in the privileged, supervisor state. The supervisor state is usually reserved for the operating system and for trusted system programs. User programs, by default, run in the less privileged user mode. In paging systems, a page-map table lists all pages that the related program can access. In addition, the table stores access rights - such as read, write, or execute- for each individual page. Each process has a separate page-map table. At run-time, the hardware address translation mechanism translates virtual addresses to physical addresses. Before allowing access to memory, the hardware verifies that (1) the target page exists in the program's address space and (2) that the intended mode of access is permitted. Any discrepancy causes an exception that invokes the protection mechanism. Loading and modification of page-map tables are privileged operations. The page-map tables themselves are usually kept in the operating system's private address space. Virtual-memory systems based on paging operate in much the same way, with the additional provision of handling legitimate references to pages that are not resident in main memory.

Systems based on segmentation use the segment descriptor tables for address translation and protection. There is one segment-map table per process. Each entry of the table defines the base address, the length (size), and the access rights to the related segment. For each memory reference, the run-time address translation mechanism verifies that (1) the segment is within the program's address space, (2) the offset is valid, and (3) the intended mode of access is permitted.

As discussed earlier, protection in secondary storage is usually effected by means of user-defined access rights that are associated with files and managed by the file system. Typically, the file owner- which is usually its creator- has the discretion to designate the access rights for all users of the file. The owner may subsequently modify the access rights in lists consisting of user IDs and their specific rights. The access list is usually stored in association with the file. For efficiency, some systems use abbreviated access lists.

7.10.2 Access-Matrix Model of Protection

The use of seemingly quite different protection mechanisms for primary and secondary memory can sometimes obscure the basic underlying issues and principles. This section introduces the access-matrix model of protection, which serves as a useful abstraction for reasoning about protection mechanisms in computer systems.

A computer system may be viewed as consisting of a set of subjects, such as processes, that operate on and manipulate a set of objects. Objects include both hardware, such as peripheral devices and memory segments, and software objects, such as files and arrays.

From the software point of view, each object is an abstract data type. Operations on an object amount to applications of functions that may transform the state of the object. In principle, the specific subset of functions that can be meaningfully applied to an individual object is object-specific. The protection mechanism should ensure that (1) no process is allowed to apply a function inappropriate to a given object type and (2) each process is permitted to apply only those functions that it is explicitly authorized for a specific object. For any given object, the latter set is a subset of the object-specific legitimate operations. The authority to execute an operation on an object is often called the access right.

Some of these relationships may be expressed by means of an abstraction called protection domain, which specifies a set of objects and the types of operations that may be performed on each object. A protection domain is a collection of access rights, each of which is a pair <object identifier, rights

set>. In general, domains need not be static; their elements can change as objects are deleted or created and the access rights are modified. Domains may overlap; a single object can participate in multiple domains, possibly with different access rights defined therein.

A simple illustration of the protection domain concept is provided by the dual, user/supervisor mode of operation found in many computer systems. A more elaborate example, provided by the IBM/360 type of hardware, uses 4-bit memory protection keys and thus, supports up to 15 user domains. In multiuser systems, each user typically has a protected set of programs and files, which amounts to as many protection domains as there are users.

A process executes in a protection domain at a given point in time. This binding is not static, and a process may switch between different protection domains in the course of its execution. In a flexible protection system, not all parts and phases of a program need be given equal and unrestricted access to all objects that the program has access rights to. For example, a procedure may have private data that it wants to have exclusive access rights to. The need to control access rights is especially pronounced in situations, where some common utilities, such as editors and compilers, are shared. In order for a process to use a shared utility, some of the user's access rights must be conveyed to it. For example, the compiler must be granted at least read access to the user's source file and, optionally, may have create and write-file access to the user program's home directory for object and listing files. However, it is unwise and dangerous to effect this transfer of rights by allowing the shared utility to assume all of the invoking user's access rights. Such promiscuous behavior, not unusual in real systems, provides a fertile ground for planning of Trojan horses and for spreading of computer viruses.

These relationships may be represented by means of an access matrix, which is a representation of all access rights of all subjects to all objects in a computer system. It is usually depicted as a two-dimensional matrix, with protection domains as rows and system objects as columns. Both hardware and software objects are included in the access matrix. Figure 7.8 illustrates a small access matrix. Blank entries indicate no access rights. Thus, for example, a process executing in domain D2 can access only one object-File 2, in read-only mode. File 3 is presumably, a shared utility that is maintained by domain D3 and is also executable in domain D1.

Object Domain	File 1	File 2	File 3	Printer
D1	Read Write		Execute	Output
D2		Read		
D3			Read Write Execute Copy	Output

Figure 7.8: Access Matrix

Although a useful model, access matrices are inefficient for storage of access rights in a computer system because they tend to be large and sparse. The actual forms of representation of access rights, captured and expressed by the access matrix, differ in practice in accordance with the access-control mechanism in use. The common access-control mechanisms are:

- Access hierarchies, such as levels of execution privilege and block-structured programming languages.
- Access lists of all subjects having access rights to a particular object.
- Capabilities or tickets for objects that specify all access rights of a particular subject.

These are discussed in greater detail:

Access Hierarchies

A simple form of access hierarchy is provided by the dual, user/supervisor, mode of operation found in many computer systems. In that model, a restricted range of operations is available in the user mode, which is a default for program execution. The supervisor mode is a superset that, in addition to user-mode instructions, allows execution of instructions that can adversely affect the system's integrity. These include certain I/O functions, halting of the machine, and updating of the address translation tables. The supervisor mode is reserved for the operating system and for trusted programs, usually various system utilities. Thus, user programs execute in the user domain, and the operating system executes in the supervisor domain. Instruction-level domain switching is allowed only in the privileged mode.

When a user program needs to perform an operation outside its protection domain, it calls the operating system. At the control-transfer-point, such as the supervisor-call instruction, the operating system can check the user's authority and grant or deny execution accordingly.

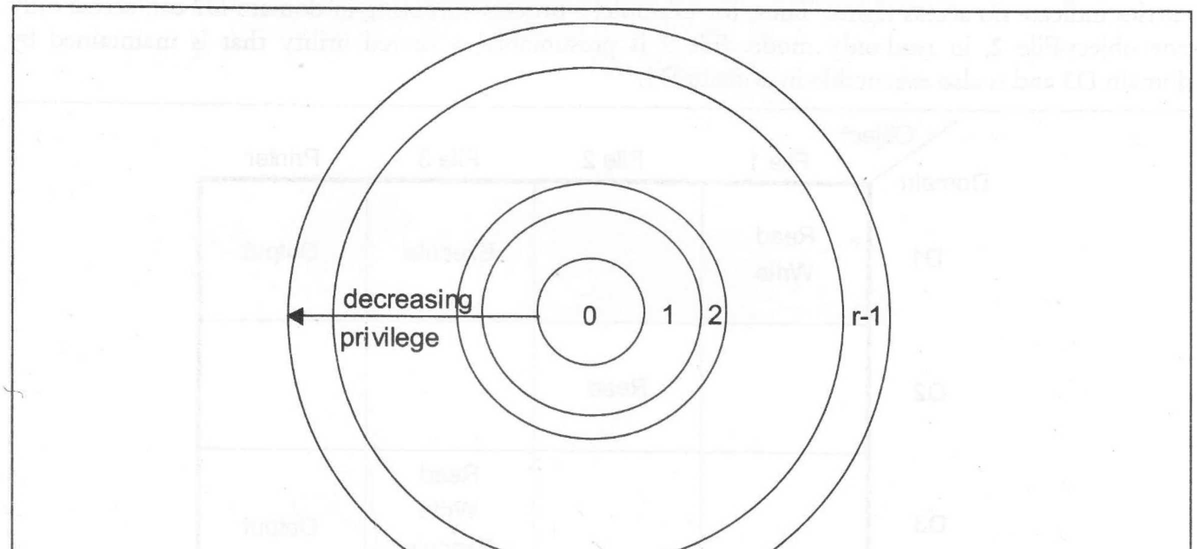

Figure 7.9: Protection Rings in Multics

Some systems extend this mode of operation to multiple levels of protection. For example, some DEC minicomputers have three modes: kernel (most privileged), supervisor and user. The kernel mode is used in some designs to run the security kernel, supervisor for the rest of the operating system, and user mode is for application programs.

The protection rings, introduced in Multics, are a generalization of the concept of a supervisor state. Each protection ring defines a domain of access. At any given time, each process runs in a specific protection ring, whose number is specified in the processor-status word as an integer in the range [0, r-1]. The access privileges of ring j are a subset of those for ring i, for all $0 \leq i \leq j \leq r-1$.

Protection rings are illustrated in Figure 7.9. Inner rings (lower numbers) have higher access rights. Protection barriers, in the form of call gates, are invoked by hardware when a lesser-privileged outer ring needs to call on a service running in an inner, more privileged, ring. Intel's 80286 and higher-numberd processors in that family, implement a reduced, four-ring version of the multics ring-protection scheme.

The concept of access hierarchy is not unique to hardware. It can also be used in software. For instance, the scope rules of block-structured programming languages, such as Pascal and C, represent a hierarchy of access domains. In that approach, the scope of an identifier encompasses the block x in which it is declared, and all blocks defined in x. As illustrated in Figure 7.10, identifiers declared in block A (outermost, level 0) are accessible in all of A's nested blocks. A statement contained in inner block D (level 2) may legally reference all identifiers declared in D's outer blocks - blocks A and B in the example - but not the identifiers declared in the disjoint block C. However, outer blocks cannot reference identifiers declared in their enclosed, inner-level blocks. For example, statements in block A do not have access to variables declared in blocks B and D, and variables declared in block D cannot be accessed from block B.

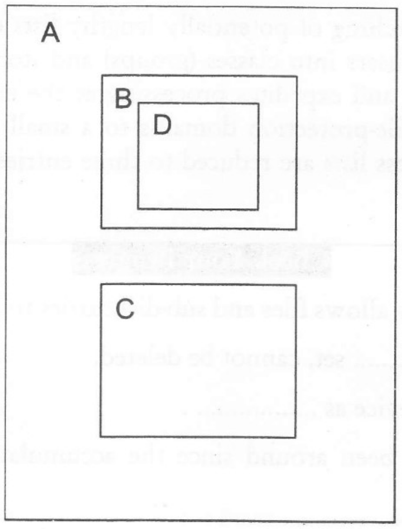

Figure 7.10: Scope in a Block-structured Language

In general, access hierarchies violate the design principle of least privilege. They usually grant too many access rights to privileged programs. For example, a process running at ring 0 has full access to the whole system. A bug or a Trojan horse in such a program can easily corrupt the entire system. Moreover, the linearity of the ring-based protection mechanism imposes too strict ordering of objects and access-right classes. This makes it difficult or impossible to represent arbitrary constraints, such as a cyclic graph.

Access Lists

Access lists are one way of recording access rights in a computer system. They are frequently used in file systems. In principle, an access list is an exhaustive enumeration of the specific access rights of all entities (domains or subjects) that are authorized access to a given object. In effect, an access list for a specific object is a list that contains all nonempty cells of a column of the access matrix associated with a given object.

In systems that employ access lists, a separate list is maintained for each object. Usually, the owner has the exclusive right to define and modify the related access list. The owner of the object can revoke the access rights granted to a particular subject or a domain by simply modifying or deleting the related entry in the access list.

Many variations of the access-list scheme are used to store access information in file systems. Typically, the access list or a pointer to it is stored in the file directory. Access lists may be combined with other schemes to strengthen protection. In multics, for example, access lists are combined with a ring-based protection scheme to control access to segments that reside on secondary storage.

The primary drawback of access lists is the search overhead, imposed by the need to verify the authority of a subject to access a requested object. According to the principle of complete mediation, every request to access a file should be checked. In order to improve efficiency, some systems check the requestor's authority only when the file is opened. This weakens protection by opening the door for penetration after the file is opened and by making revocations of privilege ineffective as long as the user has the file open-which may be indefinitely in some systems.

In order to avoid storage and searching of potentially lengthy lists of authorized users, especially for public files, some systems divide users into classes (groups) and store only the aggregate group access rights. This scheme saves storage and expedites processing at the expense of reducing flexibility and limiting the number of distinct file-protection domains to a small number of available distinct user classes. In Unix, for example, access lists are reduced to three entries per file, one each for the owner, group, and all other users (world).

Check Your Progress

1. An acyclic graph directory allows files and sub-directories to be shared. (True/False)
2. Files with attribute .... set, cannot be deleted.
3. UNIX treats every I/O device as ....
4. .... have probably been around since the accumulation of the first valuables that needed guarding.

7.11 LET US SUM UP

A file is a collection of letters, numbers and special characters: it may be a program, a database, a dissertation, a reading list, a simple letter etc. Sometimes you may import a file from elsewhere, for example from another computer. If you want to enter your own text or data, you will start by creating a file. A file system is a method for storing and organizing computer files and the data they contain to make it easy to find and access them. Disks provide the bulk of secondary storage on which a file system is maintained. To improve I/O efficiency, I/O transfer between memory and disk are performed in units of blocks. Each block is one or more sectors. Depending on the disk drive, sectors vary from 32 bytes to 4096 bytes; usually, they are 512 bytes. The file system provides the mechanism for online storage and access to both data and programs. The file system resides permanently on secondary storage, which has the main requirement that it must be able to hold a large amount of data, permanently. In a multi-user environment, a file is required to be shared among more than one users. There are several techniques and approaches to affect this operation. The file systems of computers can be extensive. Some systems store thousands of files on hundreds of gigabytes of disk. The direct-access nature of disks allows flexibility in the implementation of files. In almost every case, many files will be stored on the same disk. Security policies specify what is desired in terms of protection and security. Security mechanisms specify how to affect the security policies and enforce them in a given system.

7.12 KEYWORDS

Complete Mediation: Every access request for every object should be checked for authorization. The checking mechanism should be efficient because it has a profound influence on system performance.

Fail-safe Default: Access rights should be acquired by explicit permission only, and the default should be lack of access.

Disk File Systems: A disk file system is a file system designed for the storage of files on a data storage device, most commonly a disk drive, which might be directly or indirectly connected to the computer.

Flash File Systems: A flash file system is a file system designed for storing files on flash memory devices.

Access Hierarchy: A simple form of access hierarchy is provided by the dual, user/supervisor, mode of operation found in many computer systems.

Access Lists: Access lists are one way of recording access rights in a computer system.

7.13 QUESTIONS FOR DISCUSSION

1. Explain the file system architecture and functions.
2. What is file sharing and file allocation?
3. Briefly explain the access matrix model of protection.
4. What does the general graph directory contains?

Check Your Progress: Model Answers

1. True
2. Read Only
3. File
4. Security Policies

7.14 SUGGESTED READINGS

Andrew S. Tanenbaum, *Modern Operating System,* Published By Prentice Hall

Silberschatz Galvin, *Operating System Concepts,* Published By Addison Wesley

Andrew M. Lister, *Fundamentals of Operating Systems,* Published By Wiley

Colin Ritchie, *Operating Systems,* Published By BPB Publications

UNIT V

LESSON

8

THE UNIX OPERATING SYSTEM

CONTENTS

8.0 AIMS AND OBJECTIVES

After studying this lesson, you will be able to:

- Study the case of UNIX operating system
- Describe the command language of UNIX
- Explain the system call of UNIX

8.1 INTRODUCTION

Simply stated, UNIX is an operating system. It is by and large the most popular operating system existing today. The features and flexibility of UNIX is so immense that it has become a standard for great many operating systems. It is multi-user system, which means that more than one user can work at the same computer system at the same time. UNIX also supports multi-tasking. Multitasking means that more than one program can be made to run at the same time. For example, you can initiate a program and leave it by itself to go on and in the meantime you can work on some other program. Multi-tasking and multi-user are the two most important characteristics of UNIX, which have helped it gain widespread acceptance among a large variety of users.

The UNIX OS files consume 40 MB of the 80 MB disk space. Another 10-20 MB of disk space is eaten up as swap space. This swap space is used at that point of time when UNIX falls short of memory. So, the contents that are not immediately required are stored in the swap space. Any time when the program needs these contents, they are read from the swap space.

8.2 CASE STUDY: THE UNIX OPERATING SYSTEM

Case Study: Multinational Financial Services Corporation.

A multinational financial services organization comprised of seven separate operating companies has primary headquarters located in North America, Europe, Asia Minor, and Southeast Asia. Over 50 major regional offices provide a complete range of financial services (investment and personal banking, asset management and insurance). Each operating company is an autonomous business unit; however, at the local level, each company might share offices with one or more operating companies.

This company operates under the strict regulatory scrutiny of many countries and regions and under their respective statutes regarding financial privacy, trading, and IT functionality and security. As a result, maintaining secure and stable systems at both the network operating system level and the desktop operating system level is required.

Existing IT Environment

There is no central IT group for all operating companies, so there are no comprehensive IT standards for the entire organization. Each operating company has created its own standards; therefore, each company has its own IT infrastructure. In some locations, operating companies share one common network. In other locations, the number of networks matches the number of operating companies sharing that office location. Local offices, especially the consumer and retail locations, maintain their own file and print servers, although regional offices usually have domain controller's Regional offices are otherwise limited in their IT functions.

Some financial services applications require the UNIX operating system. Currently, all infrastructure services such as Dynamic Host Configuration Protocol (DHCP) and DNS are managed in a UNIX environment. Windows 2000 DNS dynamic update protocol will be used while the company researches the possibility of migrating the custom applications running on UNIX servers to Windows 2000.

Their current network operating system environment runs 95 percent on Windows NT Server 4.0 and five percent on Novell NetWare Bindery. The current client operating systems in use at each operating company include 80 percent Windows NT Workstation 4.0, approximately 15 percent Windows NT

Workstation 3.51, and about 5 percent Windows 95. Some financial services professionals use both UNIX and Windows NT 4.0 clients.

Source: http://technet.microsoft.com/en-us/library/cc960330.aspx

8.3 COMMAND LANGUAGE OF UNIX

8.3.1 UNIX Command Structure

There are a few of UNIX commands, that you can type them stand alone. For example, is, date, pwd, logout and so on. But UNIX commands generally require some additional options and/or arguments to be supplied in order to extract more information. Let us find out the basic UNIX command structure.

The UNIX commands follow the following format:

Command [Options] [Arguments]

The options/arguments are specified within square brackets if they are optional. The options are normally specified by a '-' (hyphen) followed by letter, one letter per option.

8.3.2 Some Commonly Used UNIX Commands

There are many commands you will use regularly. Let us discuss some of these commands, but please make a note that the options that are specified for the commands may not necessarily work on all look-alike UNIX. There might be some variations, (which can be ignored) here and there. The commands covered in this unit are:

- banner – To display information.
- cal – To display calendar on the screen.
- date – To display and set the current system date and time.
- passwd – To install or change the password on the system.
- who – To determine the currently logged users on the system.
- finger – Gives specific information about a user.

The Banner Command

This command displays information. It displays its argument exploded to a bigger size, onto the standard output. The banner command splits up the long arguments on individual word boundaries. It displays the argument up to 10 characters in large letters.

Options: None

Example: $ banner VICKY

Output:

The CAL Command

The *cal* command creates a calendar of the specified month for the specified year. If you do not specify the month, it creates a calendar for the entire year. By default, this command shows the calendar for the current month based on the system date. The cal writes its output to the standard output.

Syntax: cal [[mm] yy]

where mm is the month, an integer between 1 and 12 and yy is the year, an integer between 1 and 9999. For current years, a 4-digit number must be used, '98' will not produce a calendar of 1998.

Options: None

Examples:

❖ *$ cal*

Output:

```
May 1999

      S   M   T   W   TH F   S
                             1

      2   3   4   5   6   7   8

      9   10  11  12  13  14  15

      16  17  18  19  20  21  22

      23  24  25  26  27  28  29

      30  31
```

❖ *$ cal 1998:* The above command displays the calendar for entire year, 1998. The entire year will whip month by month.

❖ $ cal 1998/lpr : *The above command prints the calendar for the entire year onto the printer.*

The Date Command

It shows or sets the system date and time. If no argument is specified, it displays the current date and the current time.

Syntax: date[+options]

Options:

- %d – displays date as mm/dd/yy
- %a – displays abbreviated weekday (Sun to Sat)
- %t – displays time as HH:MM:SS
- %r – displays time as HH:MM:SS(A.M/P.M.)
- %d – displays only dd
- %m – displays only mm

Examples:

- $date: It will display date as – Mon June 9 04: 50:24 EDT 1998.
- $ date +%d: It will display date as – 11/12/98
- $date +%r: It will display time as – 07: 20:50 PM
- If you are working in the superuser mode, you can set the date as shown below:

```
$  date       MMddhhmm [yy]
   where        MM    =   Month (1-12)
                dd    =   day (1-31)
                hh    =   hour (1-23)
                mm    =   minutes (1-59)
                yy    =   Year
```

- It sets the system date and time to the value specified by the argument.

The Passwd Command

The *passwd* command allows the user to set or change the password. Passwords are set to prevent unauthorized users from accessing your account.

Syntax : passwd [user-name]

Options

- –d: Deletes your password
- –x days: This sets the maximum number of days that the password will be active. After the specified number of days you will be required to give a new password.
- –n days: This sets the minimum number of days the password has to be active, before it can be changed.
- –s: This gives you the status of the user's password.

The above options can be used only by the super user.

Example:

- $ passwd -x 40 bobby: The above command will set the password of the user as 'bobby' which will be active for a maximum of 40 days.

Also note, that passswd program will prompt you twice to enter your new password. If you don't type the same thing both the times, it will give you one more chance to set your password.

- *$passwd:* bobby

```
Old password:
New password:
Re-enter new password:
```

The Who Command

The who command lists the users that are currently logged into the system.

Syntax: who [options]

Options:

- –u : lists the currently logged-in users.
- –t : gives the status of all logged-in users.
- am i: this lists login-id and terminal of the user invoking this command.

Examples:

❖ $who –t

Output:

```
Shefali    +    ttyo1   Jan   12   9:50
Bobby      –    ttyo2   Jan   12   10:10
```

The second column here shows whether the user has write permission or not.

❖ $who -u

Output:

```
Shefali    ttyo1   Jan   12   9:50    1235
Bobby      ttyo2   Jan   12   10:10   2401
```

The last column here denotes the process-id.

❖ $who am i

Output:

```
Shefali    ttyp6   Jan   12   14:34
```

❖ This command shows the account name, where and when I logged in. It also shows the computer terminal being used.

The Finger Command

In larger system, you may get a big list of users shown on the screen. The finger command with an argument gives you more information about the user. The finger command followed by an argument can give a complete information for a user who is not logged onto the system.

Syntax: finger [user-name]

Options: none

Examples:

$ finger shefali

This command will give more information about Shefali's identity as shown below:

```
Login name: shefali
(512) 222-4444
Directory: /home/shefali
Last login Fri May 16 12: 14: 40 on ttyo1
```

Project: X window programming

Shefali ttyol May 18 20:05

If you want to know about everyone currently logged onto the system, give the following command:

```
$finger
```

8.4 SYSTEM CALLS OF UNIX

Now that you have learnt sufficiently about operating systems in general, let us try to learn and appreciate how UNIX operating system performs its designated functions:

- *Process management functions:* Creating, destroying and manipulating processes.
- *Memory management functions:* Allocating, de-allocating and manipulating memory.
- *Input/Output functions:* Communicating and controlling I/O device and file system.
- *Miscellaneous functions:* Network functions etc.

The UNIX System offers somewhere around 64 system calls. However, only 32 system calls are frequently used. These system calls carry very simple options with them. So, it becomes easy to make use of these system calls. The body of the Kernel is formed by the set of system calls and the internal algorithms that implement them. Thus, the Kernel provides all the services to the application programs in the UNIX system. In the UNIX system, the programs don't have any knowledge of the internal format in which the Kernel stores file data.

8.4.1 Process Management Functions

The behavior of a UNIX process is defined by its text segment, data segment and stack segment as shown in Figure 8.1.

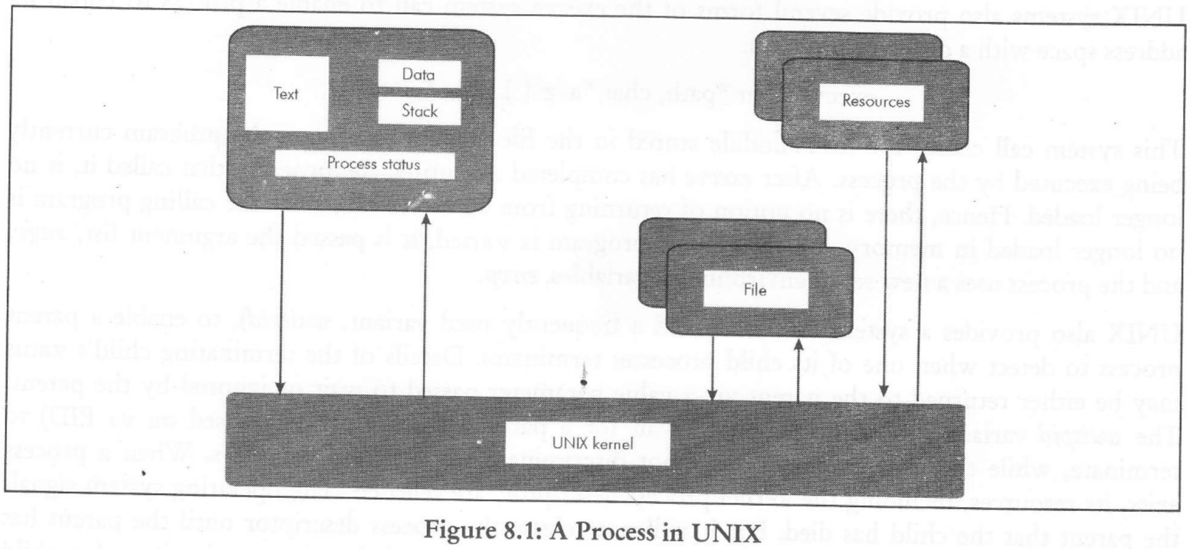

Figure 8.1: A Process in UNIX

The text segment contains the compiled object instructions, the data segment contains static variables, and the stack segment holds the runtime stack used to store temporary variables. A set of source file that is compiled and linked into an executable form is stored in a file with the default name of a. out (of course, the file can be explicitly given any name by the programmer). If the program references statically define data, such as C static variables, a template for the data segment is maintained in the executable file. The data segment will be created and initialized to contain values and space for variables when the executable file is loaded and executed. The stack segment is used to allocate storage for dynamic elements of the program, such as automatic C variables that are created when they come into scope and are destroyed when pass out of scope.

The executable file is created by the compiler and linker. These utilities do not define a process; they define only the program text and a template for the data component that the process will use when it executes the program. When the loader loads a program into the computer's memory, the system creates appropriate data and stack segments, called a process.

A process has a unique process identifier, a PID that is essentially a pointer–an integer into a table of process descriptors used by the UNIX OS kernel to reference the process's descriptor. Whenever one process references another process in a system call, it provides the PID of the target process. The UNIX pa command lists each process associated with the user executing the command. The PID of each process appears as a field in the descriptor of each process. The next time you are using UNIX, try the ps-aux command to observe the PID value identifying each process in the system.

The UNIX command for creating a new process is the *fork* system call. Whenever a (parent) process calls *fork*, a child process is created with its descriptor, including its own copies of the parent's program text, data, and segments, and access to all open file descriptors (in the kernel). The child and percent processes execute in their own separate address spaces. This means that even though they have access to the same information, both the child and its parent each reference their own copy of the data. No part of the address space of either process is shared. Hence, the parent and child cannot communicate by referencing variables stored at the same address in their respective address space. In UNIX, the only thing the two processes can reference in common is a file.

UNIX systems also provide several forms of the execve system call to enable a process to reload its address space with a different program:

execve (char *path, char *avgv[ ], char *envp[ ]);

This system call causes the load module stored in the file at path to replace the program currently being executed by the process. After *execve* has completed executing, the program that called it, is no longer loaded. Hence, there is no notion of returning from an *execve* call, since the calling program is no longer loaded in memory. When the new program is started, it is passed the argument list, *angv*, and the process uses a new set of environment variables, *envp*.

UNIX also provides a system call, *wait* (and a frequently used variant, *waitpid*), to enable a parent process to detect when one of its child processes terminates. Details of the terminating child's status may be either returned to the parent via a value parameter passed to wait or ignored by the parent. The *waitpid* variant allows the parent to wait for a particular child process (based on its PID) to terminate, while the wait command does not discriminate among child processes. When a process exits, its resources, including the kernel process descriptor, are released. The operating system signals the parent that the child has died, but it will not release the process descriptor until the parent has received the signal. The parent executes the wait call to acknowledge the termination of a child process.