Chapter 1

INSIDE THE UNIX KERNEL

Despite the simplicity of the UNIX operating system at its inception, UNIX has evolved into a very complex, versatile, and scalable operating system. This complexity is due to the fact that the kernel manages a wide variety of services including Network File Systems (NFS), Input/Output (I/O), sockets, and process and memory management. The UNIX operating system is highly tunable and supports many different types of configuration. This chapter is not meant to be an introduction to the UNIX operating system. On the contrary, this chapter requires that you have a good working knowledge of the UNIX operating system. If you find this material overwhelming or difficult to understand, please consider reading a book on the fundamentals of the UNIX operating system (see the References section).

1.1 UNIX INTERNALS

The UNIX kernel is the central core of the operating system. It provides an interface to the hardware devices as well as to process, memory, and I/O management. The kernel manages requests from users via system calls that switch the process from user space to kernel space (see Figure 1.1).

Each time a user process makes a system call such as `read()`, `fork()`, `exec()`, `open()`, and so on, the user process experiences a *context switch*. A context switch is a mechanism by which a process switches from one state to another. The process may be either suspended until the system call is completed (*blocking*), or the process may continue and later be notified of the system call completion via a signal (*nonblocking*). Figure 1.2 shows an example of a context switch.

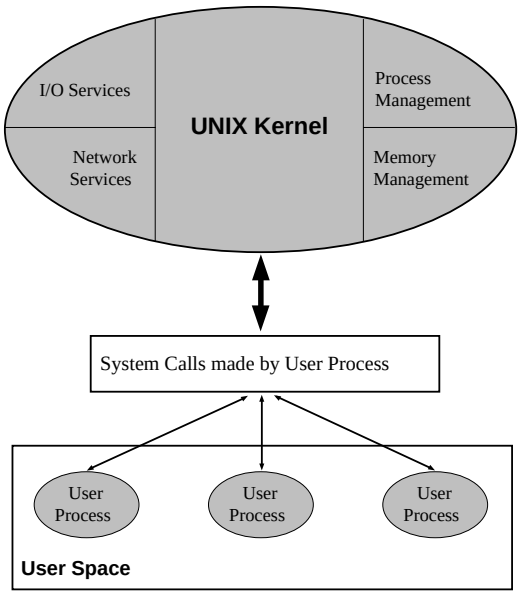

FIGURE 1.1 System calls and the UNIX kernel.

Figure 1.2 demonstrates a user process issuing a system call, in this case the `read()`[1] system call. The `read()` system call takes a file descriptor, buffer, and the number of bytes to be read as arguments. The `read()` system call forces the user process to block until the completion or timeout of the `read()` system call. Most UNIX operating system vendors provide a library for performing nonblocking (*asynchronous*) I/O calls since the traditional UNIX I/O calls of `read()`and `write()` are blocking (*synchronous*). See Chapter 3 for a more detailed discussion on the UNIX I/O model.

The UNIX kernel provides services to different system resources including I/O, memory management, process management, and network services. A particular application or user process accesses the system resources and services via system calls. The performance of the application is highly dependent on the type of system calls used and the number of system calls per application. Kernel resources are expensive and should be regarded as a valuable and limited set of resources. In order to maximize performance, the application should minimize the amount of system calls used, thereby reducing kernel space overhead and maintaining the user process in user space for most of the time. However, certain applications, such as a database management system heavily dependent on I/O, cannot avoid system calls. The goal in this situation is to tune the UNIX kernel and the system to respond as quickly and efficiently as possible so that the kernel resources can be freed quickly to service other requests. In other words, the goal of the application should be to minimize the amount of time spent in kernel space while maximizing throughput in the user space.

1. Refer to the manual pages on the `read()` system call.

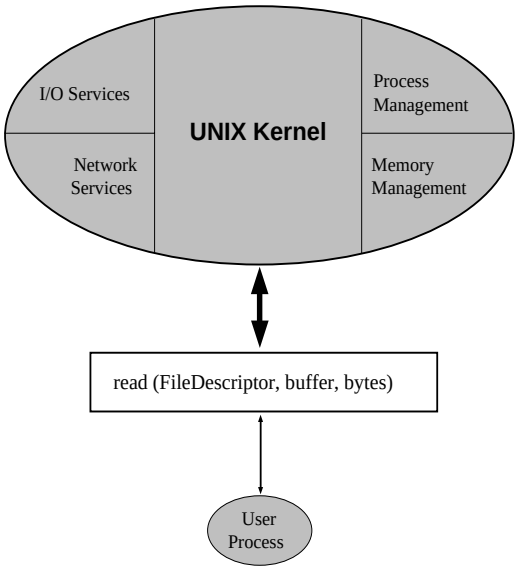

FIGURE 1.2 User process issues `read()` system call.

One of the major system calls central to the UNIX operating system is the `fork()` system call.[2] The `fork()` system call creates another process known as the *child* process, while the process that issued the `fork()` system call becomes the *parent* process. The `fork()` system call duplicates the entire process structure and the process address space of the parent process. The child process inherits the job class characteristics and environment of the parent process. The child process also inherits open file descriptors from the parent process. You can reference the parent process `ID` of a process using the `ps -ef` command.[3] The `fork()` system call enables one process to create another process either synchronously by waiting on the child process, or asynchronously by continuing the execution of the parent process. The `fork()` system call takes no arguments and returns an integer. The return value can represent one of three distinct values.

1. 0, meaning the `fork()` call successfully created a child process, and 0 refers to the child process within the parent process;
2. –1, meaning the system was unable to create another process. In this case, you can use the `perror()`[4] system call to output the exact error message that caused the `fork()` call to fail; or

2. Refer to the manual pages on `fork()` (i.e., man fork) for a detailed description of the `fork()` system call.
3. Refer to the manual pages on `ps` for platform-specific options.
4. Refer to the manual pages on `perror()` for a complete description.

3. A default positive integer greater than 0 that is returned to the parent process which represents the process `ID` of the child process. The parent process can then wait on this child process, thereby blocking until termination of the child process, or the parent process can continue execution without waiting on the child process. This would be an asynchronous event, and the parent process can be signaled when the child process terminates. The parent process would need to establish a signal handler code segment to trap and interpret the signal from the child process.

The `fork()` system call is not only used within application programs to create subprocesses and subtasks, but also within the kernel itself to create subprocesses. For example, consider the output in Table 1.1 from the `ps -ef` command.

TABLE 1.1 Output from the `ps -ef` command

UID	PID	PPID	TIME CMD
root	0	0	0:01 sched
root	1	0	0:01 /etc/init -
root	2	0	0:00 pageout
root	3	0	3:27 fsflush
root	131	1	0:01 /usr/sbin/inetd -s
root	289	1	0:00 /usr/lib/saf/sac -t 300
root	112	1	0:02 /usr/sbin/rpcbind
root	186	1	0:00 /usr/lib/lpsched
root	104	1	0:12 /usr/sbin/in.routed -q
root	114	1	0:01 /usr/sbin/keyserv
root	122	1	0:00 /usr/sbin/kerbd
root	120	1	0:00 /usr/sbin/nis_cachemgr
root	134	1	0:00 /usr/lib/nfs/statd
root	136	1	0:00 /usr/lib/nfs/lockd
root	155	1	0:00 /usr/lib/autofs/automountd
root	195	1	0:00 /usr/lib/sendmail -bd -q1h
root	159	1	0:00 /usr/sbin/syslogd
root	176	1	0:07 /usr/sbin/nscd
root	169	1	0:13 /usr/sbin/cron

In Table 1.1, the `sched` process has the Process `ID` (PID) 0 and a Parent Process `ID` (PPID) of 0, meaning the `sched` process is the base process. The `init` process has a process `ID` of 1 and a parent process `ID` of 0, meaning that the `init` process was inherited by the `sched` process. The `sched` process is the scheduler process that is responsible for scheduling processes on the run queue. The `inetd` daemon has a process `ID` of 131 and a parent process of 1, meaning the `inetd` daemon was inherited by the `init` process. The `init` process is the system initializer process responsible for spawning and initializing processes with certain defaults.

Therefore, it is apparent that the `fork()` system call plays a large role in the UNIX operating system and is used frequently in both user applications and system applications. Consider the following program segment that summarizes the `fork()` system call and its use.

File: sample1.c

```
#define FORK_ERROR    -1

#include <stdio.h>
#include <unistd.h>

main ()
{

   int lpid;

   lpid=fork();  /* fork() issued, return code is checked below */
   switch(lpid)  /* for success or failure. */
   {

       case 0:  /* Child Process section - fork() succeeded, call
                   execl(). */
         execl ("sample2",arg*0,arg*1,....,arg*n, NULL);

        case -1:  /* Unable to create process */
             perror ("Unable to create process");
             exit (FORK_ERROR);

        default:  /* Return to Parent Process */
             /* Either continue or wait on child process. */
          /* Value returned here is process id of child process. */

  }

}
```

The process flow of this program would appear as in Figure 1.3 if compiled and executed.

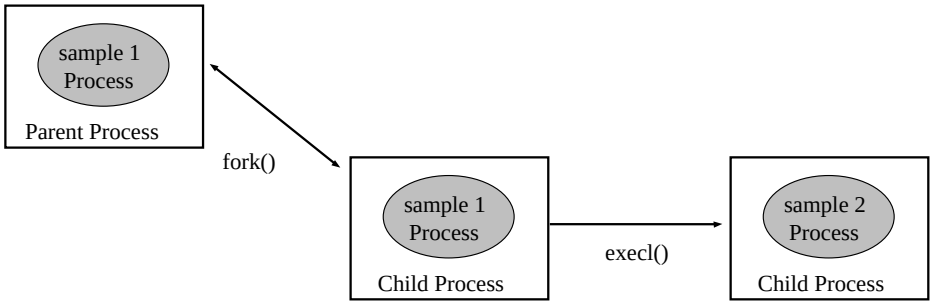

FIGURE 1.3 Process flow.

1.2 UNIX ARCHITECTURE

The UNIX operating system consists of many different layers that manage different resources and services. The basic architecture of the UNIX operating system (OS) consists of two main layers: the system or kernel layer, and the user layer.

Figure 1.4 illustrates the different layers of the UNIX operating system.

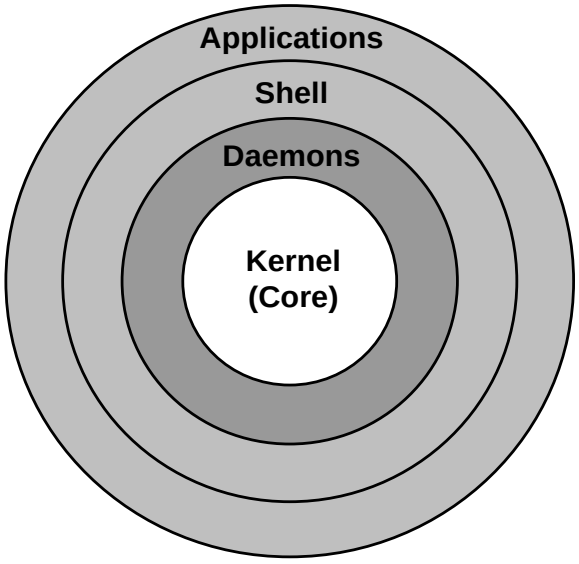

FIGURE 1.4 Layers of the UNIX operating system.

1.2.1 KERNEL LAYER

The kernel layer is the core of the operating system and provides services such as process and memory management, I/O services, device drivers, job scheduling, and low-level hardware interface routines such as test and set instructions. The kernel layer consists of many different internal routines that process the user requests. Access to the kernel is indirect via system calls. You cannot call the internal kernel routines directly. The system calls provide a mechanism by which a user process via the system call can request a kernel resource or service. The kernel itself is relatively small in terms of its disk and memory storage sizes. On Sequent systems, the kernel is located in the root directory and consists of a single file named unix (`/unix`). The size of the UNIX kernel on Sequent systems varies depending on the system configuration and the number of device drivers configured. A typical size may range from 1 MB–5 MB. On Solaris 7 (2.7) systems, the file path for the UNIX kernel is /platform/<platform_type>`/kernel/unix`. The <platform_type> indicates the type of system architecture such as sun4m or sun4u. It is important to note that when the operating system is installed on the machine, you must allocate enough space for the root file system. If you do not allocate sufficient space, you may not be able to rebuild the kernel because rebuilding the kernel requires a temporary staging area to hold both the new kernel and the previous kernel file. In addition, you may also not be able to install operating system patches if sufficient free space does not exist in the root file system. I recommend that you always leave at least 500 MB free in the root file system. Although this may seem high, rebuilding the root file system to increase its space is not an easy task. It is better to have extra space than be short a few megabytes.

The kernel also provides services such as signal handling, synchronization, interprocess communication, file system services, network services, and hardware monitoring. Each time a process is started, the kernel has to initialize the process, assign the process a priority, allocate memory and resources for the process, and schedule the process to run. When the process terminates, the kernel frees any memory and/or resources held by the process. The UNIX process model will be discussed in greater detail later in section 1.2.5.

1.2.2 FILE SYSTEMS

The UNIX file system is a hierarchical file system that begins with the `root (/)` file system. Each file system is mounted at a mount-point. For example, the `/(root)` file system is mounted on the / mount-point. The `/usr` file system is mounted on the `/usr` mount-point. Figure 1.5 shows the hierarchical directory structure of the UNIX file system.

In Figure 1.5, the `/usr`, `/var`, and `/opt` directories are subdirectories of the `/(root)` file system. The `/usr`, `/var`, and `/opt` subdirectories may each be separate file systems from the root file system; however, the hierarchical file structure still applies. In fact, it is recommended that the `/(root)`, `/var`, `/opt`, and `/usr` subdirectories all be separate file systems. This helps prevent the root file system from being filled up with patches, system log files, E-mail, and user software packages. This also increases the performance of the root file system since the overhead of maintaining a larger file system is reduced by separating the operating system (OS) binaries and executables from user software packages and system log files.

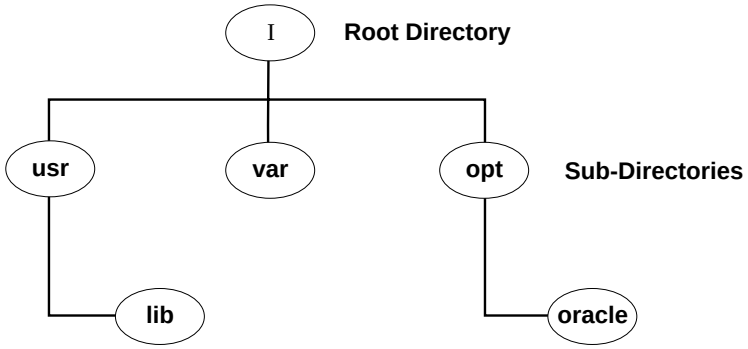

FIGURE 1.5 Directory structure of the UNIX file system.

The UNIX file system has many different file system types such as `ufs` (UNIX file system), `NFS` (Network file system), `vxfs` (Veritas file system), and `cdfs` (CD-ROM file system). The `ufs` file system type is the most standard UNIX file system consisting of super-blocks, inodes, offsets, and storage blocks. Reads and writes to a `ufs` file system are done in blocks depending on the size of the file system block size. Block sizes can range from 1 KB to 8 KB depending on the file system type selected. `NFS` is a remote file system that is exported to other client systems. These client systems may mount the remote file system from the server that is exporting the file system. There are system daemons on both sides (`NFS` server daemons and `NFS` client daemons) that communicate with each other via RPC (Remote Procedure Call). `NFS` packets are IP (Internet Protocol) packets that utilize the UDP (User Datagram Protocol) datagram.[5] The `vxfs` file system type is a file system based on the Veritas Volume Management software.[6] The `cdfs` file system type is a CD-ROM–based file system, also referred to as `hsfs` (High Sierra file system) and ISO 9660 CD-ROM file system.

1.2.2.1 `ufs`

The `ufs` file system is the most common UNIX file system type and consists of a block-based file system that contains offsets, super-blocks, cylinder groups' maps, inodes, and storage blocks. Figure 1.6 shows the `ufs` file system type structure.

Each ufs file system consists of multiple cylinder groups with each cylinder group consisting of the structure diagrammed in Figure 1.6. The boot block is contained only in the first cylinder group of the file system (Cylinder Group 0). The boot block is 8 KB in size and contains the boot strap used during the system boot process. If the file system is not used for booting (i.e., other than the root file system), the boot block is left blank.

5. Solaris 2.5 changed the NFS code (NFS Version 3) to utilize TCP/IP instead of UDP.
6. Veritas provides disk volume management software for UNIX platforms.

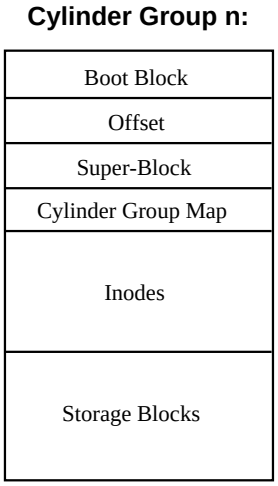

FIGURE 1.6 The `ufs` file system type structure.

The super-block consists of information about the file system, such as

- ❑ Status of the file system (used by `fsck`)
- ❑ File system label (name)
- ❑ File system size in logical blocks
- ❑ Date and time stamp of the last update
- ❑ Cylinder group size
- ❑ Number of data blocks per cylinder group
- ❑ Summary data block

The cylinder group map is a specific feature of the `ufs` file system type and consists of a block of data per cylinder group that maintains block usage within the cylinder. The cylinder group map also maintains the free list of unused blocks, as well as monitors disk fragmentation.

The inode layer is one of the most important layers within the file system. The inode layer contains most of the information about a file with the exception of the file's name. The name of the file is maintained in a directory. An inode is approximately 128 bytes long. By default, an inode is created for every 2 KB of storage in the file system. The number of inodes can be specified during the file system creation phase by using the `newfs` or `mkfs` command.[7] See Chapter 3 for more details on tuning file systems. An inode in the `ufs` file system contains the following information.

7. Refer to the manual pages on `newfs` and `mkfs` for specific parameters and options.

- ❑ Mode and type of the file
- ❑ Number of hard links to the file
- ❑ User `ID` of the owner of the file
- ❑ Group `ID` of the group to which the file belongs
- ❑ Number of bytes in the file
- ❑ Two arrays comprising a total of 15 disk block addresses
- ❑ Date and time stamp of last access
- ❑ Date and time stamp of last modification
- ❑ Date and time stamp of file creation

The core of an inode is two arrays that together consist of 15 disk block addresses. The first array is comprised of 12 direct addresses. These direct addresses map directly to the first 12 storage blocks of the file. If the file size exceeds 12 blocks, the first address of the second array refers to an indirect block. This indirect block consists of direct addresses as opposed to file contents. The second address refers to a double indirect block containing addresses of indirect blocks. The third address references a triple indirect block containing addresses of indirect blocks. Figure 1.7 portrays the `ufs` addressing scheme.[8]

Table 1.2 shows the maximum number of bytes addressable by each level of indirection.[9]

TABLE 1.2 Maximum file addressable size

Logical Block Size	Direct Blocks	Single Indirect Blocks	Double Indirect Blocks	Triple Indirect Blocks
2 KB	24 KB	1 MB	512 MB	256 GB
4 KB	48 KB	4 MB	4 GB	4 TB
8 KB	96 KB	16 MB	32 GB	64 TB

Note: TB (Terabyte)

The maximum file size in a `ufs` file system is 2 GB, which is addressable through triple indirection. Triple indirection is a signed 32-bit field, hence the 2 GB file limit as most UNIX operating systems are 32-bit.[10] Most UNIX vendors are currently working to convert to a 64-bit–based operating system. The 64-bit operating system will eliminate the 2 GB file limit, thus allowing terabyte size files. Although the logical file system block size ranges from 2 KB to 8 KB, `ufs` provides a smaller subset of a block known as a fragment or chunk. The fragment or chunk is usually 1 KB in size but can be set to a multiple of the operating system block size of 512 bytes. Fragments are used to accommodate smaller files in order to avoid a wastage of space. The maximum length of a `ufs` filename is 255 bytes.

8. Kulihin, Julia, Fox, Mary L., Nester, Joan. *System Administration*, Vol. 2. UNIX SVR4.2 MP, pp. 3–18.
9. Kulihin, Julia, Fox, Mary L., Nester, Joan. *System Administration*, Vol. 2. UNIX SVR4.2 MP, pp. 3–18.
10. Digital has the DEC-Alpha UNIX Servers, which are based on Digital's 64-bit operating system.

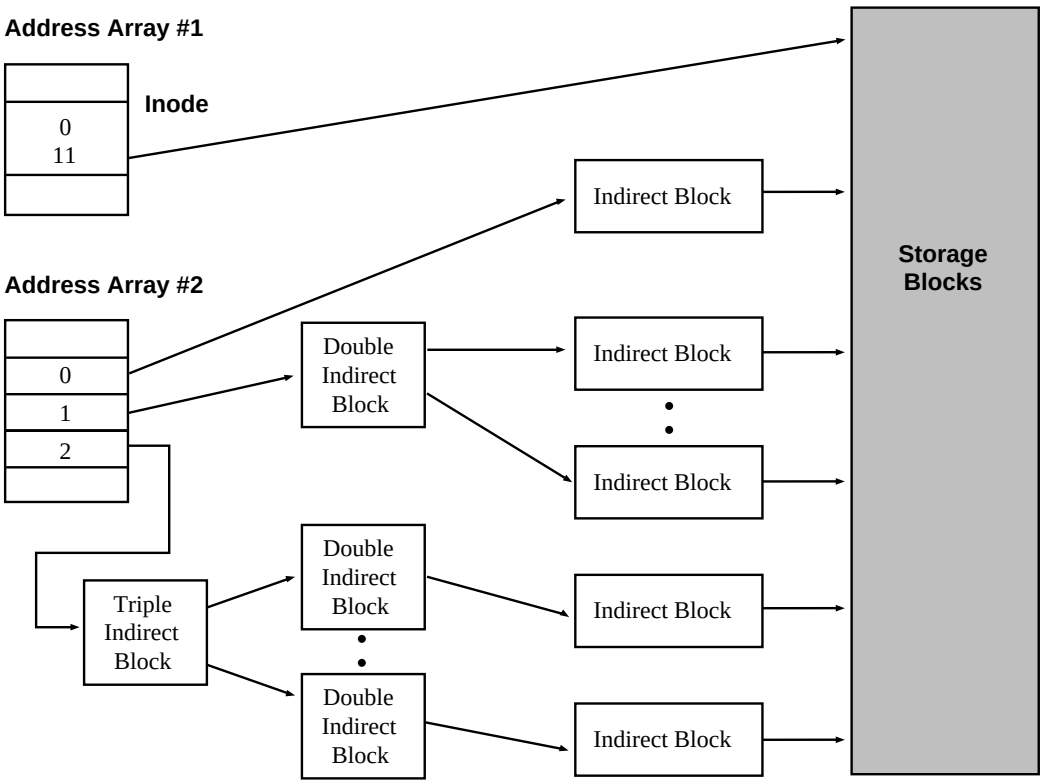

FIGURE 1.7 The `ufs` addressing scheme.

Each time a `ufs` file system is mounted, the state of the file system is checked. If the file system state is stable or clean, the file system is mounted and made available for use. If the file system state is unstable, an `fsck` is performed on the file system. The `fsck` utility scans the file system and corrects any discrepancies found between the inode information and data storage blocks. The `fsck` utility may cause a loss of data if certain inconsistent files are truncated. For large file systems, fsck may take a long time because `fsck` is a synchronous process and must scan the entire file system. For this reason, you may choose to use a journaled file system such as `vxfs`.

1.2.2.2 `vxfs`

The `vxfs` file system is based on the Veritas Volume Management software. The `vxfs` file system offers several enhancements over the `ufs`-based file system. It provides extent level allocation and a journal feature. The extent level allocation reduces the allocation workload by allocating an extent (many blocks) at once, as opposed to the `ufs`-based file system which allocates in increments of blocks. The journal feature is similar to the

concept of the Oracle redo log. As changes are made to the file system, those changes are recorded in the journal. Therefore, if the system were to crash or panic, an `fsck` would not be necessary since the journals need only to be applied to restore the file system's state of consistency. This significantly reduces the length of recovery time following a system crash or panic. Figure 1.8 details the structure of the `vxfs` file system.[11]

1.2.2.3 NFS

`NFS` (Network file system) is a remote file system that is accessed by client systems over the network. `NFS` file system types are extremely common and popular in the UNIX environment. `NFS` allows multiple client systems access to a common file system from an `NFS` server. It also provides heterogeneous systems the ability to share files and data. This centralizes system administration since only the `NFS` server needs to be backed up. This of course also leads to a single point of failure. `NFS` file systems typically are used for development, home directories, and software distribution. Figure 1.9 shows how an `NFS` file system works.

In Figure 1.9, the `NFS` server—that is, the system exporting the file system for remote mounting (access)—is running the `NFS` server daemons which service `NFS` client requests. On the client side, the client—a workstation or another server—mounts the file system as an `NFS` file system and accesses the `NFS` file system through the `NFS` client daemons. The `NFS` client daemons communicate with the `NFS` server daemons via RPC (Remote Procedure Call). Remote procedure calls enable applications to call procedures (functions) on a

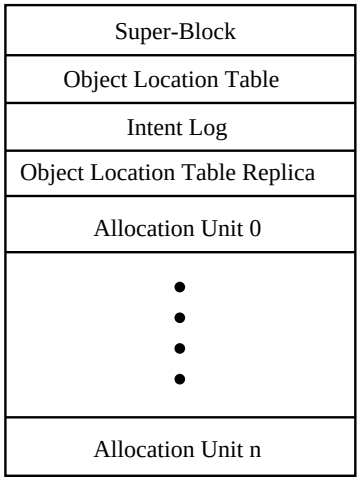

FIGURE 1.8 Structure of the `vxfs` file system.

11. Kulihin, Julia, Fox, Mary L., Nester, Joan. *System Administration*, Vol. 2. UNIX SVR4.2 MP, pp. 3–33.

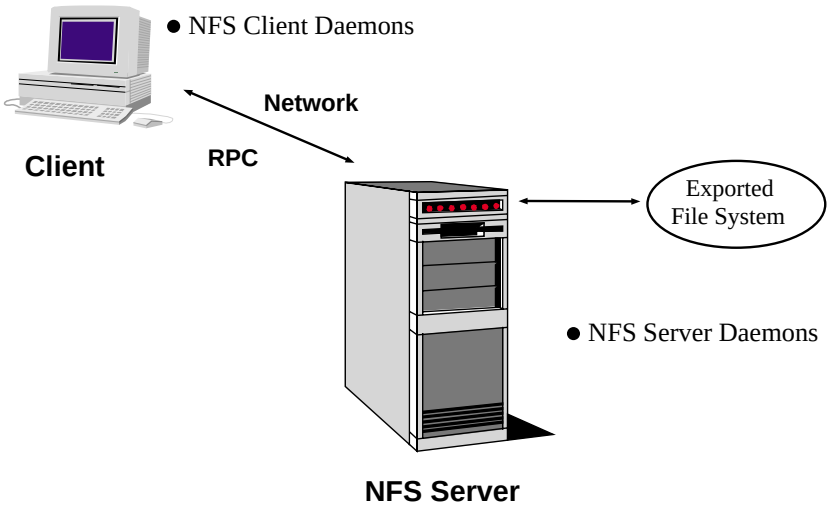

FIGURE 1.9 The NFS file system concept.

remote system. RPC uses the XDR (External Data Representation) protocol for data exchange. For more information on XDR or RPC, refer to the respective manual pages, or refer to a programmer's guide on RPC. A heavily hit `NFS` server can significantly increase the network traffic. This can affect network and `NFS` performance. There are many performance issues to consider when using `NFS` file systems. Performance issues pertaining to `NFS` are discussed in Chapter 3.

1.2.2.4 `cdfs`

The `cdfs` file system is a CD-ROM–based file system that is used to mount CDs for the purposes of installing software and/or data. These file systems are read-only and provide specific options during mounting[12] that control the format of the files on the CD.

1.2.2.5 The vnode

Internally, UNIX manages file systems using *vnodes*. Vnodes are known as virtual nodes and act as a higher-level object to the inode. The UNIX operating system uses the inode object to manage files and file systems. Vnodes are virtual nodes which then map to the specific object type depending on the use of the vnode. For example, for the `ufs` file system, a vnode consists of an inode. For an NFS file system, the vnode is an rnode (remote node). For more information on the vnode structure, you can browse the vnode header file

12. Refer to the manual pages on the `mount` command.

/usr/include/sys/vnode.h. The vnode structure allows for a generic file system implementation regardless of the specific type of file system.

1.2.2.6 Large Files

In the race to provide support for large files (files larger than 2 GB in the 32-bit world), operating system vendors have recently provided APIs allowing applications to manage large files. Solaris 2.6, HP-UX 10.2, and Dynix ptx 4.4 provide APIs that support 64-bit file offsets. Many vendors such as Sun, HP, Sequent, Oracle, and Veritas participated in the Large File Summit. The Large File Summit (LFS) is an industry initiative to develop a generic specification for the support of files larger than 2 GB. The LFS specification extends the current interfaces to behave differently when dealing with large files. For instance, the `open()` system call will set errno to `EOVERFLOW` if the file is greater than or equal to 2 GB.

The LFS draft defines a set of additional 64-bit API functions that provide support for regular size and large size files. In essence, the LFS draft defines a new set of functions named `xxx64()` corresponding to the existing set of functions named `xxx()`. For example, `open64()`, `creat64()`, `lstat64()`, and `lseek64()` are a few of the 64-bit functional interfaces added by the LFS specification. There is also a 64-bit file offset data type, namely `off64_t`.

In Solaris 2.6, setting `_FILE_OFFSET_BITS` to 64 in your application source before including system headers enables the 64-bit interfaces. Setting `_FILE_OFFSET_BITS` to 64 results in `off_t` being type-defined as a `long long` 64-bit data type. In addition, the existing file I/O function calls `-xxx()-` will be mapped to their 64-bit counterpart `-xxx64()`. In order to utilize the large file functionality, refer to your platform-specific documentation. Each platform may have a different set of APIs and commands for dealing with large files.

Solaris 2.6 also provides a new mount option which enables or disables large file support. If the file system is mounted with the `largefiles` option, then files larger than 2 GB can be created in the file system. The `nolargefiles` mount option disables support for large files. The system administrator can use these mount options on a per file system basis in order to manage large file support.

Large files are useful because they improve system administration and performance. System and database administrators now have the ability to reduce the number of files by utilizing large files. This helps reduce the complexity of backups and restores in dealing with a large number of files. However, large files also reduce the amount of file I/O parallelism. For example, two 2 GB files may be backed up or restored concurrently as opposed to a single 4 GB file. Large files may also improve performance by allowing wider stripes. The 2 GB file size restriction limited the sizes of disk partitions and thus the width of the stripe. By using the large file feature, more disks can be added to the stripe. For instance, instead of creating a 2 GB stripe consisting of three disks, you can create a 4 GB stripe consisting of six disks. The large file option has its advantages and disadvantages, but for the most part it is a useful feature enabling UNIX applications to manage large files.

1.2.2.7 UNIX Buffer Cache

UNIX employs a file system buffer cache to cache file system information such as directory name lookups, inodes, and file control block data. The actual file data blocks are part of the virtual memory pages, not the buffer cache. The concept of the UNIX file system cache is simple: data from reads and writes is maintained in the cache in the event that subsequent reads may request data that is already in the cache. This significantly increases performance since access to the cache is in the time order of nanoseconds (memory access speeds), and access to the disk is in the time order of milliseconds (disk access speeds). This is an order of magnitude 100,000 times greater since milliseconds have a base of 10^{-3}, and nanoseconds have a base of 10^{-9}. When a user process issues a read, the kernel first scans through the UNIX file system cache to determine if the data is already in the cache (*cache hit*). A cache hit means that the read request found the data in the cache, thereby avoiding physical disk I/O. If the data is in the cache, the kernel copies the data from the file system cache into the user's workspace. Although this is a memory-to-memory copy, system performance can suffer during periods of high file system activity. This can also lead to increased levels of paging. See Chapter 3 for a discussion of the performance tuning tips for the UNIX file system cache. If the data was not found in the cache (*cache miss*), the kernel issues a physical read to the disk drive(s) containing the data and brings the data into the UNIX file system cache. A cache miss means that the data was not found in the cache, and physical I/O was necessary to bring the data into the cache. The kernel then copies the data from the UNIX file system cache into the user's workspace. Figure 1.10 shows a cache hit and Figure 1.11 shows a cache miss. Each time a user process issues a write to a file system–based file, the data is written to the file system cache and later flushed to disk. Depending on the flags used to open[13] the file, physical writes to the file may be deferred. If the writes are deferred, the data is written only to the cache and is later flushed out to disk by the file system flush daemon. If the file was opened with the flag that requests writes to not be deferred, writes to the file are physically written to the disk. However, all file system reads are always read from the UNIX file system cache, unless specific calls are used to bypass the UNIX file system cache. The size of the UNIX file system cache is controlled by kernel-tunable parameters. On Solaris, the `bufhwm` kernel parameter controls the maximum size of the file system buffer cache. Another related parameter, `npbuf`, controls the number of physical I/O buffers allocated each time more buffers are needed (provided the total amount does not exceed `bufhwm`). These parameters can be set in the `/etc/system` file.

The UNIX file system cache uses a least recently used algorithm to age data out of the cache. The file system cache is maintained by a system daemon that is both time-driven and space-driven. The daemon wakes up periodically and scans for *dirty* pages to be written out. A dirty page is a page that has changed and has been marked as dirty, meaning that the page should be written out to disk upon the next write-out scan. The daemon also ensures that there is sufficient space for reads and writes by flushing dirty disk pages to disk or aging out least recently used data. The file system daemon is tunable, and performance gains can be achieved by tuning the file system daemon appropriately.

13. Refer to the manual pages on the `open()` file system call.

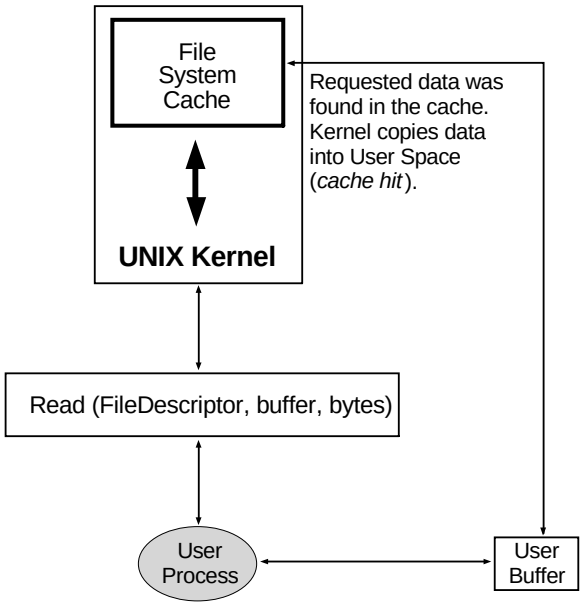

FIGURE 1.10 A cache hit.

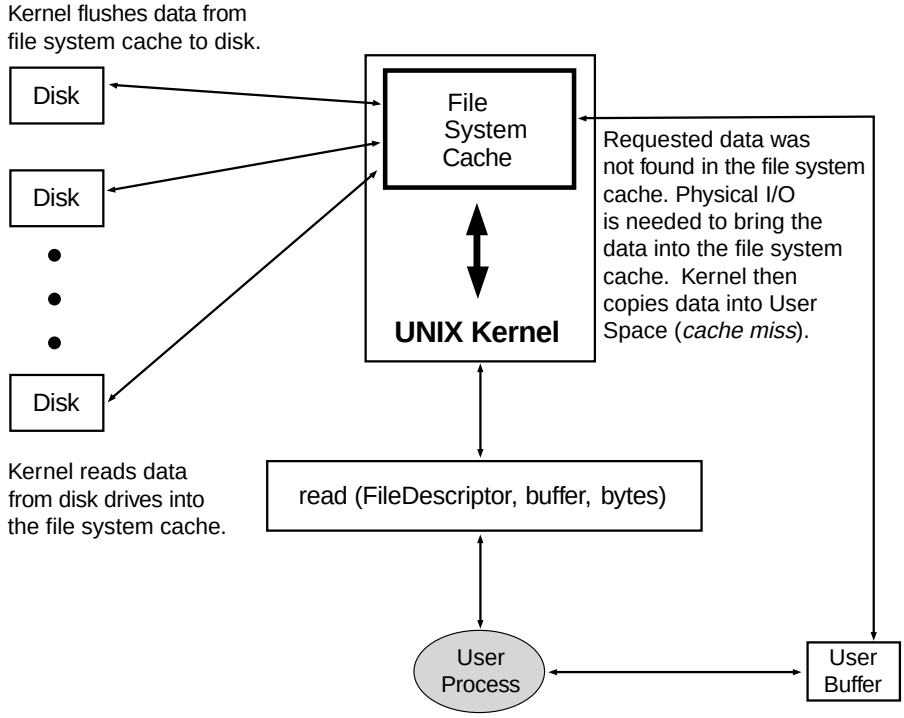

FIGURE 1.11 A cache miss.

1.2.3 VIRTUAL MEMORY

The UNIX operating system is based on the virtual memory model. The virtual memory model allows a process or a system to address more memory than physically exists. The UNIX virtual memory model consists of a set of memory pages, usually 4 KB each. Older versions of UNIX had page sizes of 8 KB. You can determine the page size on your platform by using the `sysconf()`[14] system call. The UNIX operating system uses a swap device in addition to physical memory to manage the allocation and deallocation of memory. For example, a machine with 64 MB of physical memory and a 192 MB swap device supports a virtual memory size of 256 MB. Earlier versions of UNIX did not provide a finer level of memory management than swapping. Swapping is the process by which the system no longer has enough free physical memory available, and processes are completely swapped out (written) to the swap device. Once the process has been completely written out to the swap device (i.e., swapped out), the physical memory occupied by the process can be freed. The swap device can be a single file or series of files, and each swap file can be no larger than 2 GB.[15] Therefore, the UNIX kernel virtual memory is based on both the physical memory and swap device. Figure 1.12 illustrates the virtual memory and swap device.

Because the UNIX operating system is based on the virtual memory model, a translation layer is needed between virtual memory addresses and physical memory addresses. This translation layer is part of the kernel and is usually written in machine language (Assembly) to achieve optimal performance when translating and mapping addresses. Figure 1.13 illustrates the virtual to physical translation layer within the kernel.

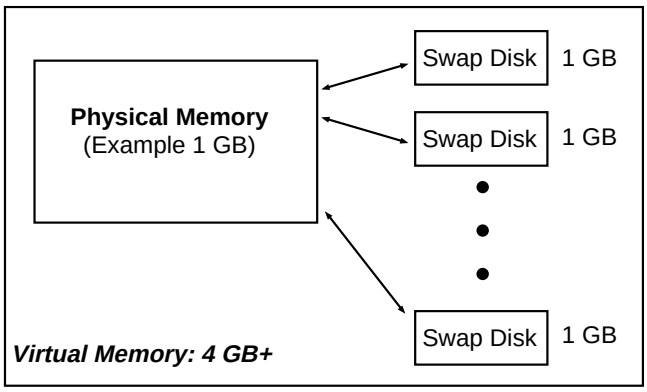

FIGURE 1.12 Virtual memory model.

14. Refer to the manual pages on the `sysconf()` system call.
15. This is due to the limitation that a single file on 32-bit UNIX platforms cannot exceed 2 GB. Most OS vendors currently provide large file support.

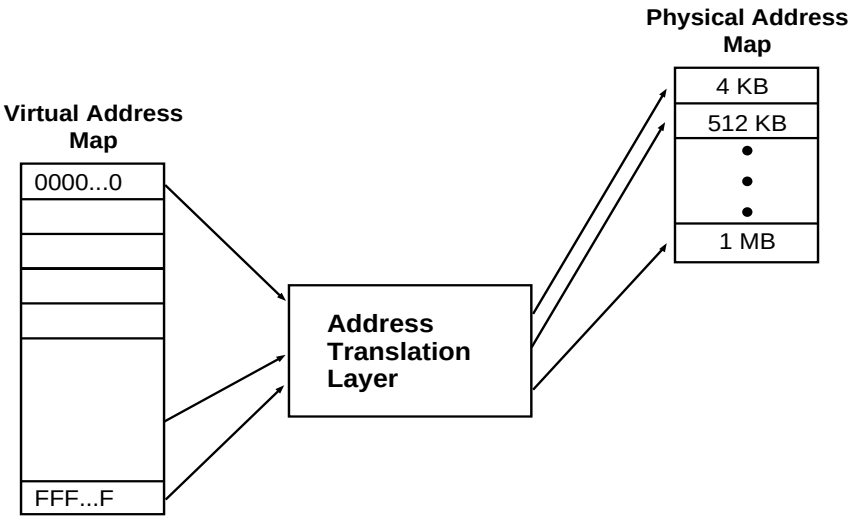

FIGURE 1.13 Virtual to physical memory address translation layer.

In Figure 1.13, each time a user process accesses memory, the kernel uses the address translation layer to map the virtual memory address into a physical memory address. Virtual addressing makes it possible to address larger amounts of memory than physically exist. As long as the virtual memory address scheme supports large memory configurations, adding additional physical memory is straightforward. Most 32-bit UNIX kernels support a maximum of 4 GB of direct addressable memory per process, and extensions are provided to address memory beyond 4 GB.

Current versions of UNIX provide a more granular approach to swapping known as *paging*. Paging swaps out various different pages in memory to the swap device rather than the entire process as in swapping. This not only increases performance by minimizing the time needed to load a process into memory, it can also provide more physical memory to other processes. In certain instances, a process might allocate a large section of memory upon process invocation. However, most of this memory may remain unused during the life of the process. For example, declaring a large System Global Area (SGA) when only a small section of the SGA is actively used could cause the unused portions of the SGA to be paged back to the free list. This allows other processes access to these physical memory pages. Hence, paging not only minimizes the chances of a complete swapout, it also provides higher availability to memory by utilizing idle memory pages. The UNIX kernel maintains a free list of memory pages and uses a `page` daemon to periodically scan the memory pages for active and idle pages. The `page` daemon uses a clocklike algorithm to maintain pages in memory. Figure 1.14, shows the paging algorithm.[16]

16. SunService. *SunOS 5.X Internals Guide SP-365*, Revision B.1, September 1993, pp. 5–6.

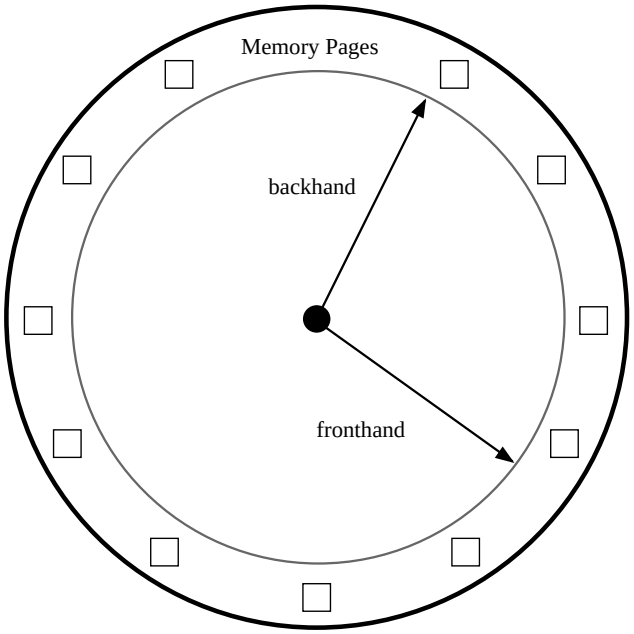

FIGURE 1.14 The paging algorithm.

In Figure 1.14, the `page` daemon using the minute hand (fronthand) scans the page list and examines the reference bit. If the reference bit is active, the page's reference bit is set to idle. If the reference bit is marked idle (meaning that no references have been made to this page), the page becomes a candidate for the page free list. During the second phase, the hour hand (backhand) scans the page list. If the reference bit is still idle and was set to idle during the previous scan, the page becomes a candidate for the free list. If the page reference bit is set (meaning that the page is active and has been referenced), the page is left alone. This algorithm's goal is twofold: to ensure that the system has plenty of free memory, and to minimize paging and swapping. There are system thresholds that control the frequency of the `page` daemon. As the available free memory closely approaches these thresholds, the `page` daemon will increase the frequency of its scans in order to increase the amount of free memory. The thresholds include limits such as the amount of memory that is always left free, the desired or ideal amount of free memory, and the amount of free memory before swapping occurs. All of these limits control the activity of the `page` daemon. Higher limits cause an active `page` daemon. Lower limits reduce the activity of the `page` daemon. Tuning the `page` daemon is discussed in Chapter 2.

Paging and swapping can severely impact the performance of your system, and efforts must be made to ensure that swapping never occurs and that paging activity is minimized. It should be noted that swapping is an act of desperation by the UNIX kernel to free memory immediately. This act of desperation causes processes to be completely swapped out in order to free memory. Thrashing occurs when even swapping was not successful in freeing memory, and memory is needed to service critical processes. This usually results in

a system panic, as this is the last action that the kernel can take to free memory. Swapping needs to be avoided at all costs as this is evidence of insufficient memory. You may need to tune your system and/or application to use less memory or you may need to add additional physical memory to accommodate the workload.

1.2.4 SHARED MEMORY

The UNIX System V Release 4 operating system standard provides many different mechanisms for interprocess communication. Interprocess communication is required when multiple distinct processes need to communicate with each other either by sharing data or resources or both. Interprocess communication is also used when processes are dependent on one another. For example, one process may be responsible for updating files while another process is responsible for receiving the updates. Once received and verified, the process that receives the updates may forward the updates on to the other process to process the updates. These two processes need a method to share data. Shared memory is one of the methods of UNIX System V Release 4 interprocess communication. Shared memory, as the term suggests, enables multiple processes to "share" the same memory, in other words, multiple processes mapping into the same virtual address map. Shared memory avoids expensive memory-to-memory copies when processes need to share data in memory. Shared memory also increases application portability because most platforms support shared memory. Figure 1.15 illustrates the concept of shared memory.

Many applications use shared memory as a technique for interprocess communication. Oracle uses shared memory to hold the SGA. This enables Oracle processes and Oracle user connections to attach to the same SGA without having to copy the SGA between each process. Having to copy the SGA between processes not only would be time consuming, but memory requirements would also grow at an explosive rate.

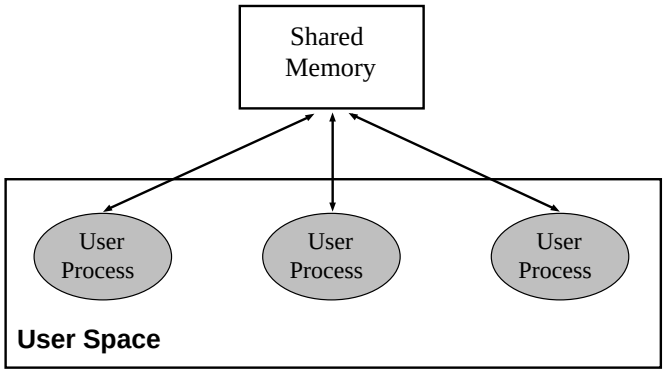

FIGURE 1.15 Shared memory concept.

1.2.5 PROCESS MODEL

The UNIX operating system uses a priority-based round-robin process scheduling architecture. Processes are dispatched to the run queue based on priority and are placed on the sleep queue based either on the process' completion of its time slice or by waiting on an event or resource. Processes can sometimes result in a deadlock when waiting on a resource. Hence, user applications need to provide deadlock detection to ensure that user processes do not deadlock on resources.

```
machine1% myprogram
```

When a user issues the command `myprogram` at the UNIX prompt, the kernel places the request for myprogram on the dispatch queue with a certain level of priority. The default priority for the particular job class is assigned assuming that no specific priority has been requested either through the `nice`[17] command or `priocntl()`[18] system call. The internal kernel routines scan the dispatch queue and the sleep queue to ensure that processes are not starved from the CPU and that processes do not exceed their time slice. Do not confuse internal kernel routines with system calls. Internal kernel routines are routines that can be used and called only by the kernel. System calls are the user interface to the kernel system services. For example, `fork()` and `read()` are examples of user system calls, while `getpage()` and `putpage()` are examples of internal memory management routines. Figure 1.16 illustrates the process scheduling cycle of the UNIX OS.

Figure 1.16 shows both the dispatch (i.e., run) queue and the sleep queue as containing processes. The dispatch queue contains processes marked as ready to run. Once a process is marked as runable, the process will begin execution on a CPU as soon as a CPU becomes available. In Figure 1.16, three processes are shown on the dispatch queue that then begin execution on the three available CPUs. Once the time slice (time-share job class) of the process is exceeded, the process is migrated to the sleep queue in favor of other processes which have been sleeping and are now marked as ready to run. The process scheduling algorithm in the UNIX kernel tries to balance CPU time collectively with other processes in order to avoid CPU starvation and CPU saturation. Once on the sleep queue, the process with the highest priority will be scheduled to run. The same holds true of the dispatch queue. The kernel may also *preempt* processes by reducing their priority. Preemption is a technique used by the UNIX kernel to stop a currently running process or set of processes and schedule other processes to run. A process can be preempted under many different circumstances, such as when the process has exceeded its time slice, a higher priority process needs to be scheduled, a system event or interrupt occurred that needs to be serviced, or the process appears to be waiting on a resource that is busy. The kernel maintains information about processes including the current process state. Using the `ps` command, you can obtain the process state of a particular process or list of processes. For example, Table 1.3 shows the output listing from the `ps` command.

17. The `nice` command enables a user to change the priority of a particular user process. Only the superuser or root account can increase the priority of a process using `nice`.
18. The `priocntl()` system call provides a system call interface to user process priority management.

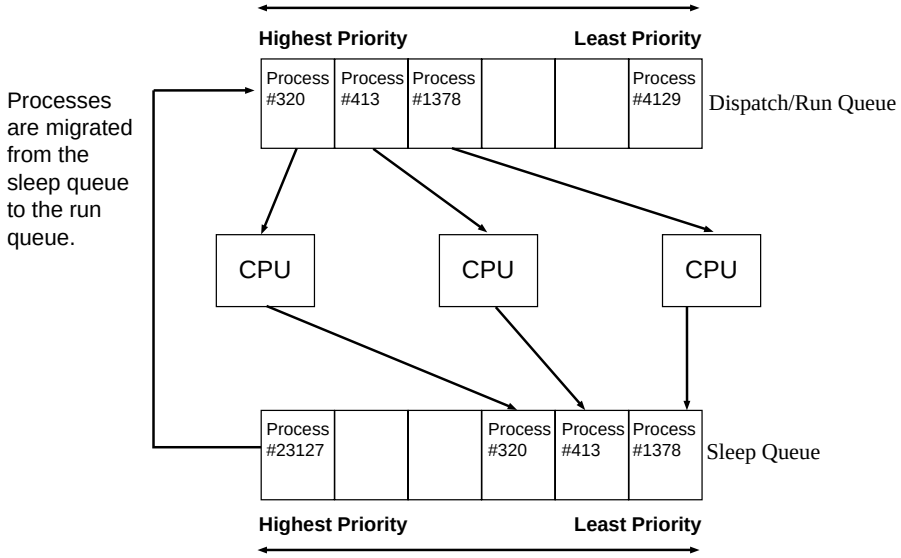

FIGURE 1.16 Process scheduling cycle of the UNIX OS.

TABLE 1.3 Output listing from the `ps` command

ST	UID	PID	PPID	TIME	CMD
S	root	0	0	0:01	sched
S	root	1	0	0:01	/etc/init -
O	root	2	0	0:00	pageout
R	test	281	280	0:01	xterm
Z	test	282			<defunct>

The process state can be classified as one of the following:

`S` — Process is currently sleeping.
`O` — Process is currently running on a CPU.
`R` — Process is ready to run.
`I` — Process is idle; currently being created.
`T` — Process is being traced.
`X` — Process is awaiting memory.
`Z` — Process is a zombie.

In a process state of S, the process is sleeping and is waiting on an event. The event could be waiting on an I/O request completion, waiting for a locked resource, or waiting on some data that is not yet available. A swapped process will also have a state of S (sleeping). Putting a process to sleep and later awakening the process is an expensive operation and can lead to performance problems—especially when a high number of process sleeps occur. A process state of O means the process is active and is running on a CPU. The process state R indicates that the process is ready to run, and as soon as a processor becomes available, the process will run (assuming it is not preempted in favor of a higher priority process). A process state of I indicates that the process is idle and is being created or initialized. A process state of T indicates that the process is being traced. This is common during debugging mode when parent processes trace child processes to detect problems. X refers to the state when the process is awaiting additional memory. Lastly, Z refers to the zombie process state. The zombie process state has two main meanings: the first is that the process (child process) has terminated and has a parent process. The second is that the process is marked to be killed by the UNIX kernel. Each process has a complete process structure associated with it when it is created. This process structure has a link to the address space of the process. During the zombie state, the process structure and address space are removed and freed back to the system. The only information remaining in the kernel for a zombie process is an entry in the process table. The kernel is responsible for cleaning up the process table and removing zombie entries. Figure 1.17 shows the process map within the kernel.

The process structure contains many different fields, including process information and process statistics. Each time a process is created, a separate process structure for the process is created. To learn more about the process structure, I encourage you to read the `<proc.h>` header file located in the `/usr/include/sys` directory. This header file is platform-specific, and it provides a complete listing of the process structure.

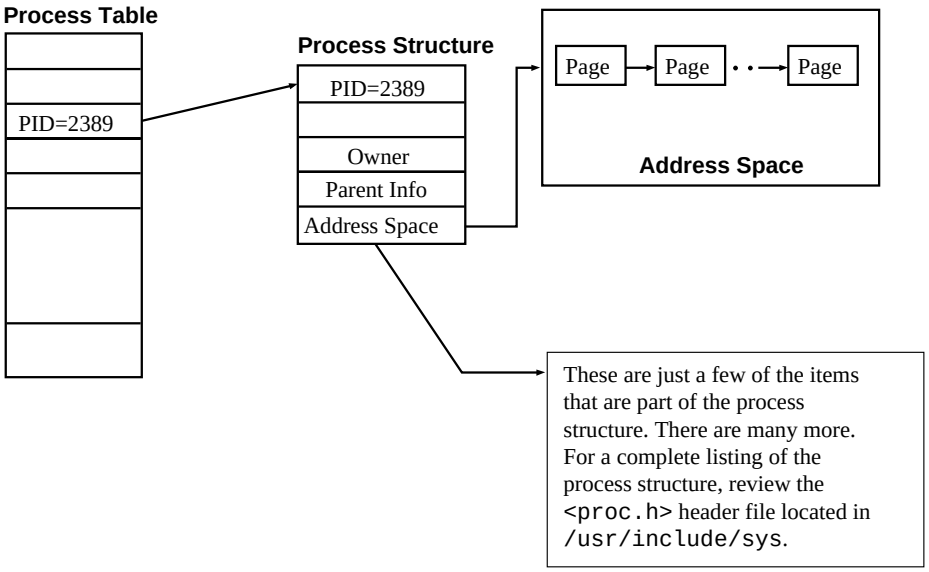

FIGURE 1.17 Process structure.

1.2.6 THE JOB SCHEDULER

The UNIX operating system job scheduler provides three different types of job classes as listed in Table 1.4. Each class listed in Table 1.4 has a discrete set of global priorities that control the scheduling order of the tasks (processes) within the particular job class. The job classes listed in Table 1.4 are listed in order of increasing priority. It is critical to understand the purpose as well as the properties of each job class. The understanding of the different job classes will enable you to further understand the kernel's job scheduling policy. I must emphasize that extreme care must be taken before altering the job class and/or priority of any particular process. Doing so without a full understanding of the possible effects could cause system instability, possibly forcing the system to panic. Please consult with your hardware and OS vendor before altering job-class parameters. Figure 1.18 summarizes the UNIX job scheduling classes.[19]

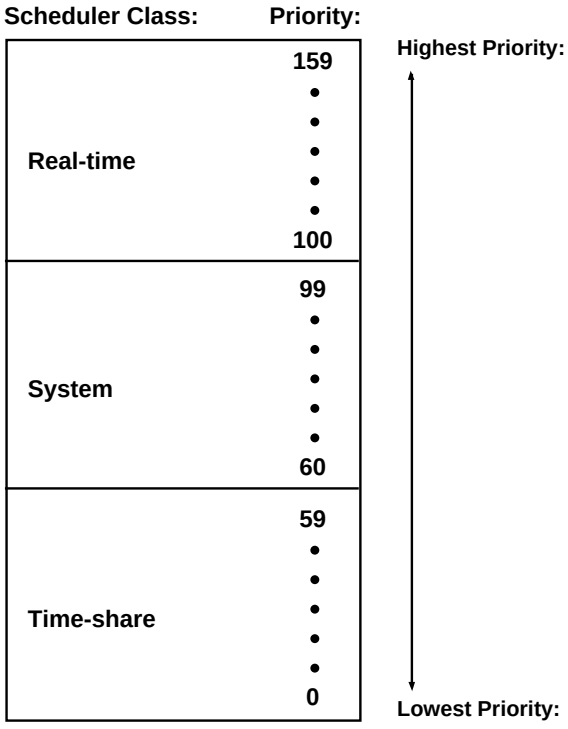

FIGURE 1.18 UNIX job scheduling classes.

19. SunService. *SunOS 5.X Internals Guide SP-365*, Revision B.1. September 1993, pp. 6–7.

TABLE 1.4 Job classes provided by the UNIX operating system job scheduler

JOB CLASS	PRIORITY RANGE
Time-share	0 through 59 (59 being the highest); default job class
System	60 through 99; reserved for system daemon processes
Real-time	100 through 159; highest priority job class

1.2.6.1 Time-Share Job Class

The time-share job class is the default job class and has priority 0 through 59, with 59 being the highest priority. In this job class, each process is assigned a time slice (or quantum). The time slice specifies the number of CPU clock ticks that a particular task (or process) can occupy the CPU. Once the process finishes its time slice, the process priority usually is decreased and the process is placed on the sleep queue. Other processes waiting for CPU time may have their priority increased, thereby increasing their likelihood to run. Unless specified through the `dispadmin`[20] or `priocntl` command, any time a new process is created, it will be assigned to the time-share job class with a default priority. You can use the `ps -c` command to list the job class and priority of a process.

For example,

```
myhost%> ps -c
  PID   CLS   PRI   COMD
 2403   TS    59    csh
15231   TS    48    ps
```

In this example, the output from the `ps -c` command shows the Process `ID` (`PID`) of the current shell as well as the `ps` command itself. Both processes are in the Time-Share (TS) job class, and have a priority of 59 or less.

1.2.6.2 System Job Class

The system job class is reserved for system daemon processes such as the pageout daemon or the file system daemon. The priority range for the system job class is between 60 and 99. The system job class has a higher priority than the time-share job class. This is to ensure that the system can provide services to the time-share job class processes when system services such as memory, and/or file I/O are requested. Although it is possible to alter the job class of a time-share process to real-time, it is not recommended because the process would run at a higher priority than the system daemons. Use extreme care before altering the job class or priority of processes.

20. The `dispadmin` command is the Solaris utility used for process scheduler administration while the system is running. You can use this command to change job classes. Refer to the manual pages on `dispadmin` for a complete listing. On Sequent, use the `priocntl` command. On HP, use the `rtsched` to schedule a real-time job.

1.2.6.3 Real-Time Job Class

The real-time job class is the highest priority job class and has no time slice. Real-time process priorities range between 100 and 159. The fixed-priority job class (real-time) ensures that critical processes always acquire the CPU as soon as the process is scheduled to run—even if processes in other classes are currently running or scheduled to run. Again, use caution when switching priorities or job classes of processes. Consult with your vendor before switching processes into different job classes. There may be some operating system–specific issues when the job class or priority of a process is altered. Because there is no time slice with real-time processes, the number of context switches is reduced significantly, which can increase performance. More discussion on process management as it pertains to performance follows in Chapter 2.

1.2.7 THREADS MODEL

Several UNIX operating system vendors support the threads model either through a vendor-supplied library, or by supporting and supplying the POSIX[21] threads library. Sun Solaris 2.X provides a Solaris threads library as well as the POSIX threads library. HP-UX 10.X provides user threads and HP-UX 11.0 provides both user and kernel threads. The threads model consists of an LWP (Light-Weight Process) structure. A thread is a unit of control or execution within a process. A thread can be thought of as a subtask or subprocess. Unlike the process model, which creates a separate process structure for each process, the threads model has substantially less overhead than a process. A thread is part of the process address space; therefore, it is not necessary to duplicate the process address space of the process when a new thread is created. The `fork()` system call creates a new process and duplicates the process address space of the parent process. However, when a thread is created, it is part of its process address space. This significantly reduces memory consumption as well as creation time for the thread. There is an order of magnitude of difference between creating a process versus creating a thread. It takes approximately 1000 times longer to create a full process than it does to create a thread.

Each thread can be bound to a CPU, thereby enabling parallel processing. There are various routines such as `thr_create()`, `thr_suspend()`, `thr_join()`, and `thr_continue()` that create a thread, suspend a thread, block until thread termination, and continue execution of a thread, respectively.[22] Threads are extremely efficient but also are extremely complex. There are many issues such as signals, stacks, and synchronization that have to be considered carefully when using the threads model as opposed to the process model. The threads model calls for high-throughput low-overhead asynchronous processing. By default, threads are asynchronous and careful thought has to be given to the design of a threads application with respect to synchronization. While the concepts of threads and

21. POSIX is the Portable Operating System for the UNIX Standards Committee.

22. For more information on these Solaris thread system calls, refer to the manual pages on either `thr_create()`, `thr_continue()`, `thr_join()`, or `thr_suspend()`. The POSIX thread calls start with `pthread`, such as `pthread_create()`.

threads programming are certainly beyond the scope of this book, I intend to discuss the basic concepts of threads as well as the performance gains associated with threads. I encourage you to obtain a more detailed text on threads and threads programming.

Prior to the availability of threads, parallelism and asynchronous processing were accomplished by using the `fork()` system call. Using the `fork()` system call, a process created another process allowing both the parent and child process to execute simultaneously (assuming enough CPUs were available to handle multiple processes). The fork technique generates tremendous system overhead because separate process structures must be created and additional memory must be allocated. Solaris 2.X is a multithreaded operating system and most of the system daemons use threads to service requests.

1.2.8 SIGNALS AND INTERRUPTS

Fundamental to any operating system are the concepts of signals and interrupts. Signals are software events and interrupts are hardware events. Signals can be asynchronous or synchronous. Asynchronous signals can occur at any point. Asynchronous signals are things like the `QUIT` signal (`CTRL-\`), software interrupt signal (`CTRL-C`), or hang-up signal. Synchronous signals are caused by an invalid operation such as a floating-point exception, a divide-by-zero error, a segmentation violation, or a bus error. Signals are commonly used within applications as a method of communication between processes. Applications also use signals to trap certain events such as a user issuing the kill command on a specific process. The application could trap the kill signal and then shut down cleanly, as opposed to not trapping the signal and immediately aborting due to the kill request. You can use the `signal()`, `sigset()`, or `sigaction()` system calls to establish a signal handler in your application. You may choose to take a certain action upon receiving a certain signal or choose to ignore the signal and allow your application to continue. Certain signals, such as fatal hardware events, cannot be ignored and usually result in a core dump with the application being terminated. You can refer to the manual pages on signals or refer to the `/usr/include/sys/signal.h` file for system-defined signals. You can also define your own application user-specific signals.

Interrupts are kernel events that are used to notify the kernel when an event has occurred. A user application may issue a series of I/O calls and the kernel will pass these I/O calls to the I/O queue. The I/O devices involved will then process the I/O requests and then send an interrupt to the kernel, notifying the kernel that the I/O requests have completed. The kernel may then send a signal to the user process notifying the process that its I/O requests have completed. The user process can then check the error code of the I/O system calls to determine if the I/O requests have completed successfully. The kernel also uses interrupts to preempt processes or take a processor off-line.

1.2.9 NETWORK SERVICES AND COMMUNICATION

The kernel provides various network services including remote login, file transfers, *connectionless* connections, and `NFS`. The kernel provides a system daemon—the `inetd` daemon,

to provide network services such as `rlogin`, `rsh`, `ftp`, and many other TCP/IP–based services. The default networking stack protocol for the UNIX kernel is TCP/IP. TCP/IP works in two parts: TCP (Transmission Control Protocol) is responsible for the virtual circuit layer that provides bidirectional data transmission; IP (Internet Protocol) is responsible for packet transmission between the user processes and the networking layer. Figure 1.19 shows the relationship between TCP and IP.

When the system is booted in multi-user mode, the `inetd` daemon is started. The `inetd` daemon is the Internet services daemon responsible for `telnet`, `rlogin`, `rsh`, and other types of Internet services. Upon a `login` or `telnet`, the `inetd` daemon forks and execs the appropriate daemon such as `in.telnetd` for a `telnet` session. The `/etc/inet/inetd.conf` file is the configuration file used to configure the `inetd` daemon. You can use this file to restrict certain Internet services such as remote shell (`rsh`) or finger. Upon an Internet service request such as `ftp`, the `inetd` daemon forks and execs the appropriate service daemon such as `in.ftpd` in the case of an `ftp` request. Using the `ps` command, you can verify this as there should be a process for the `ftp` session (`in.ftpd`), and the parent process `ID` should be the process `ID` of the `inetd` daemon. Most system administrators disable certain Internet services such as the remote shell and finger in order to increase the security of the system.

The raw IP layer consists of connectionless sockets. Sockets are a low-level method of communication using endpoints. With sockets, you can specify the type of protocol to use within the communication endpoint.

The UNIX kernel also supports other methods of communication such as pipes, named pipes, sockets, and messages. Messages are another method of interprocess communication. Messages typically are used to send special data to processes. A pipe is a unidirectional mechanism used to send streams of data to other processes. Named pipes are also another method of interprocess communication that provide a permanent path, as opposed to the traditional pipe. Sockets are bidirectional and are generally used for network communication between clients and servers. The client opens the communication endpoint via a socket, and the server listens on the socket to receive the communication.

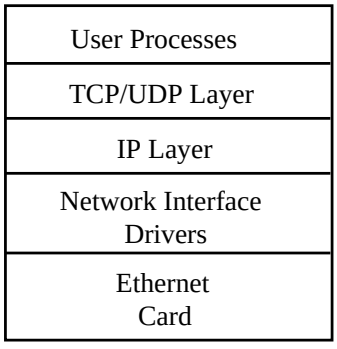

FIGURE 1.19 The networking layers.

1.2.10 I/O SERVICES

The UNIX kernel provides many different types of I/O including the standard synchronous I/O that consists of the `read()` and `write()` system calls, asynchronous I/O, streams I/O, and network I/O. Network I/O consists of the various different network services such as `ftp`, `telnet`, and other TCP/IP or other protocol-driven I/O services. Stream I/O is a mechanism for character-based I/O between the kernel and user processes. Examples of a stream are network connections, tape media, and other character devices. Asynchronous I/O is an enhancement to the standard synchronous I/O that allows nonblocking reads and writes. The kernel performs the asynchronous reads and writes and returns control to the calling program immediately following the asynchronous I/O system call. In the synchronous model, calling the system call `read()` or `write()` would not return control to the calling program until the completion of the `read()` or `write()` call. With asynchronous I/O, calls to `aioread()` or `aiowrite()` return immediately following the call.[23] If a signal handler has been established, the user process is later signaled when an outstanding asynchronous I/O call completes. Solaris 2.X provides a user library for writing applications using asynchronous I/O. The full path of the library is `/usr/lib/libaio.so` (shared library). Sequent also provides the library `/usr/lib/libseq.a` that contains the asynchronous I/O system calls.

The Sun Solaris library provides the `aioread()` and `aiowrite()` system calls for initiating asynchronous reads and writes, respectively. Sequent's library, `libseq.a`, provides `DIO_Read()` and `DIO_Write()` which, in addition to asynchronous I/O, provide direct I/O to file systems. Direct I/O bypasses the file system cache and accesses the disks directly.

1.2.11 SYNCHRONIZATION

The UNIX kernel provides several different methods of synchronization. Synchronization is the process of locking a resource to ensure consistency so that two or more processes do not invalidate each other's modifications. Although multiple processes can execute concurrently, locks are needed when a process wants to protect a resource such as a file, data block, and/or section of memory. For instance, consider the standard banking transaction: An individual is withdrawing money from his or her account, while another relative is depositing funds into the same account simultaneously. Without synchronization (locking), the end balance may not be reflected correctly.

The Solaris kernel provides the following methods of synchronization:[24]

1. mutex locks
2. condition variables

---

23. The `aioread()` and `aiowrite()` calls are provided by the Solaris asynchronous I/O library (`libaio.so`). For more information on the calls, you can refer to the manual pages on `aioread()` and `aiowrite()`.

24. SunService. *SunOS 5.X Internals Guide SP-365*, Revision B.1. September 1993, pp. 2–31. Refer to the manual pages on mutexes, condition variables, semaphores, or reader/writer locks for detailed information on the related system calls.

3. semaphores
4. multiple-reader single-writer locks

Mutex locks are mutual exclusion locks and can be used at the process level or at the individual thread level. Mutex locks are simple and efficient low-level locks. Condition variables are used in conjunction with mutex locks to verify or wait on a particular condition. Semaphores are a standard method of synchronization and are part of the System V Release 4 interprocess communication kit. A semaphore is a nonnegative integer count of available resources. The integer count is either incremented or decremented as resources are freed or obtained, respectively. A semaphore count of zero indicates that no more resources are available. Multiple-reader single-writer locks are more complex than the other three types of synchronization as they provide multiple-reader locks as well as a writer lock. Multiple-reader single-writer locks are primarily used for data that is queried more often than updated. No reader locks can be obtained while a writer lock is held. Writer lock requests are favored over readers.

1.2.12 SYSTEM DAEMON LAYER

The concept of daemons is fundamental to the UNIX OS. A daemon is simply, as the term suggests, a task acting on behalf of another task or entity. In other words, a daemon is nothing more than a background job or process responsible for a certain task or set of tasks. The UNIX kernel consists of numerous system daemons that are responsible for memory management, file system management, printer jobs, network connections, and several other services. For example, consider the output shown in Table 1.5 from the ps command.

The listing in Table 1.5 from the `ps` command shows some of the system daemons that are running.[25] The `pageout` daemon (in Table 1.5 with `process ID=2`) is responsible for writing modified (i.e., dirty) pages to the file system. The `page` daemon, under certain conditions, will scan the memory page list and determine if the page is currently in use or has been recently referenced. Depending on the paging algorithm used,[26] the `page` daemon either will skip the memory page or return the page to the free list. The main goal of the `page` daemon is to ensure that the system does not run out of memory. After the page has been flushed out to disk by the `pageout` daemon, the page is placed on the page free list. The `pageout` daemon continues to run as long as there are pages to be written out to disk.

Another system daemon is the `fsflush` daemon. This daemon is responsible for flushing dirty disk pages from memory to disk. The UNIX kernel employs a UNIX file system cache for managing file systems. All writes to a file system–based file are first written to the file system cache, assuming the writes do not explicitly bypass the UNIX file system cache. The goal of the `fsflush` daemon is to ensure that dirty file system pages are written to disk.

25. The preceding output from `ps` was generated from a Sun Server running Solaris 2.5.
26. Paging algorithms are discussed in detail in Chapter 2.

TABLE 1.5 Output from the `ps` command

UID	PID	PPID	TIME	CMD
root	0	0	0:01	sched
root	1	0	0:01	/etc/init -
root	2	0	0:00	pageout
root	3	0	3:27	fsflush
root	131	1	0:01	/usr/sbin/inetd -s
root	289	1	0:00	/usr/lib/saf/sac -t 300
root	112	1	0:02	/usr/sbin/rpcbind
root	186	1	0:00	/usr/lib/lpsched
root	104	1	0:12	/usr/sbin/in.routed -q
root	114	1	0:01	/usr/sbin/keyserv
root	122	1	0:00	/usr/sbin/kerbd
root	120	1	0:00	/usr/sbin/nis_cachemgr
root	134	1	0:00	/usr/lib/nfs/statd
root	136	1	0:00	/usr/lib/nfs/lockd
root	155	1	0:00	/usr/lib/autofs/automountd
root	195	1	0:00	/usr/lib/sendmail -bd -q1h
root	159	1	0:00	/usr/sbin/syslogd
root	169	1	0:13	/usr/sbin/cron
root	194	186	0:00	lpNet

System daemons are critical to the operating system and service requests both from the kernel and from the user. The goal of these daemons is to satisfy these requests while maintaining the integrity of the system. System daemons generally close the standard files `stdin`, `stdout`, and `stderr`.[27] The file descriptor values for `stdin`, `stdout`, and `stderr` are 0, 1, and 2, respectively. Output from system daemons is usually directed to the system console or specific system administration files.

27. `Stdin`, `stdout`, and `stderr` are the default file descriptors associated with each process. `Stdin` refers to standard input. `Stdout` refers to standard output, and `stderr` refers to standard error.

1.2.13 SHELL LAYER

The UNIX operating system also provides different types of shells that act as a command interpreter and an interface to the OS. UNIX provides three standard types of shells: The Bourne shell (`sh`), the Korn shell (`ksh`), and the C-shell (`csh`). The Bourne shell and C-shell seem to be the most common. You may use the shell of your choice and customize it accordingly using the dot (.) files such as `.login`, `.cshrc`, and `.logout` for the C-shell. The `.cshrc` is the C-shell resource file invoked each time a C-shell is created. You can set objects such as environment variables, aliases, and file locations. You can also use your shell to invoke specific startup scripts or programs. Your default shell can be set by the UNIX system administrator in the `/etc/passwd` file or equivalent namespace in NIS or NIS+. Some UNIX vendors also provide a restricted shell (`/usr/lib/rsh`). This should not be confused with the standard remote shell (`rsh`), which is a remote login facility. The restricted shell is a special shell that restricts the user from changing environment variables and from changing directory locations. Under a restricted shell, the user may execute only commands found in the $PATH environment variable. This shell is often useful for secured accounts that need to run certain applications only. The system administrator can use a restricted shell to increase system security, permitting the user to run only certain commands and applications.

1.2.14 APPLICATION LAYER

The application layer within the UNIX architecture consists of user-level programs or scripts that are written in some sort of programming or script language. These programs or scripts tend to be executable or binary files used within an application. If a programming language such as C or C++ is used, the source code for the application is compiled using the C or C++ compiler. The compiler then generates object code specific to the platform and architecture. For example, compiling C code on a Sequent machine using DYNIX/ptx 4.4 generates object code based on the Intel architecture and the DYNIX/ptx operating system. Linking the object code produces a DYNIX/ptx format executable.

The application layer is an important layer within the UNIX architecture as it is responsible for running database software such as the Oracle8*i* Enterprise Edition, business-specific applications, graphics tools, and development and debugging tools. It is important that you understand this layer and all the layers beneath. This solid understanding will help you understand the Oracle architecture, thereby maximizing your ability to successfully tune your Oracle and UNIX environment.