UNIT V
CASE STUDY

LINUX interfaces and Shell.

Interfaces to Linux

A Linux system can be regarded as a kind of pyramid, as illustrated in Fig. 10-1. at the bottom is the hardware, consisting of the CPU, memory, disks, a monitor and keyboard, and other devices. Running on the bare hardware is the operating system. Its function is to control the hardware and provide a system call interface to all the programs. These system calls allow user programs to create and manage processes, files, and other resources.

Programs make system calls by putting thearguments in registersand issuing trap instructions to switch from user mode to kernel mode. Since there is no way to write a trap instruction in C, a library is provided, with one procedure per system call. These procedures are written in assembly language, but can be called from C. Each one first puts its arguments in the proper place, then executes the trap instruction. Thus to execute the read system call, a C program can call the read library procedure. As an aside, it is the library interface, and not the system call interface, that is specified by POSIX. In other words, POSIX tells which library procedures a conformant system must supply, what their parameters are, what they must do, and what results they must return. It does not even mention the actual system calls.

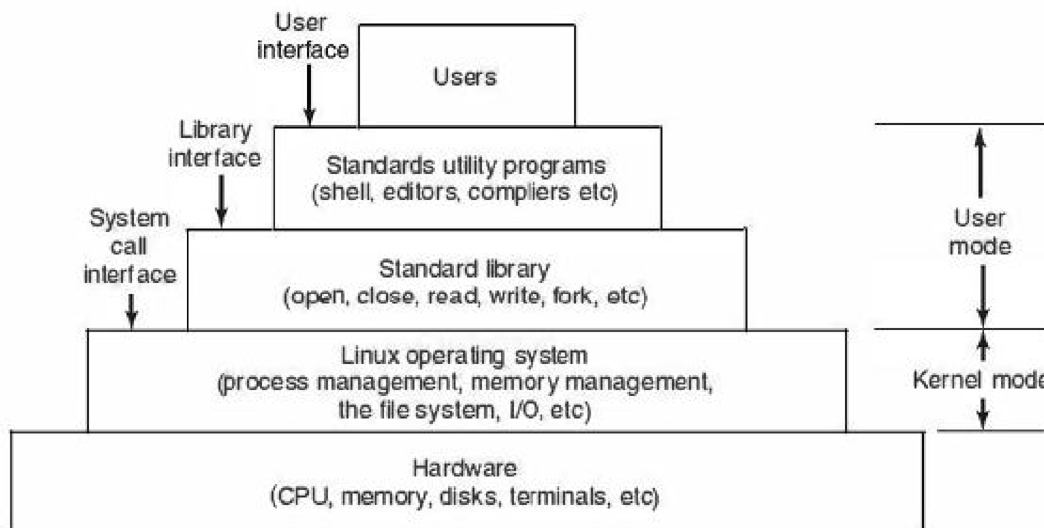

Figure 10-1. The layers in a Linux system.

In addition to the operating system and system call library, all versions of Linux supply a large number of standard programs, some of which are specified by the POSIX 1003.2 standard, and some of which differ between Linux versions.

These include the command processor (shell), compilers, editors, text processing programs, and file manipulation utilities. It is these programs that a user at the keyboard invokes. Thus we can speak of three different interfaces to Linux: the true system call interface, the library interface, and the interface formed by the set of standard utility programs.

Most personal computer distributions of Linux have replaced this keyboard oriented user interface with a mouse-oriented graphical user interface, without changing the operating system itself at all. It is precisely this flexibility that makes Linux so popular and has allowed it to survive numerous changes in the underlying technology so well.

The GUI creates a desktop environment, a familiar metaphor with windows, icons, folders, toolbars, and drag-and-drop capabilities. A full desktop environment contains a window manager, which controls the placement and appearance of windows, as well as various applications, and provides a consistent graphical interface. GUis on Linux are supported by the X Windowing System, or commonly Xll or just X, which defines communication and display protocols for manipulating windows on bitmap displays for UNIX and UNIX-like systems. The X server is the main component which controls devices such as keyboards, mouse, screen and is responsible for redirecting input to or accepting output from client programs. The actual GUI environment is typically built on top of a low-level library, xlib, which contains the functionality to interact with the X server. The graphical interface extends the basic functionality of Xll by emitting the window view, providing buttons, menus, icons, and other options.
When working on Linux systems through a graphical interface, users may use mouse clicks to run applications or open flies, drag and drop to copy files from one location to another, and so on. In addition, users may invoke a terminal emulator program, or xterm, which provides them with the basic command-line interface to the operating system.

The Shell

Although Linux systems have a graphical user interface, most programmers and sophisticated users still prefer a command-line interface, called the shell.
The shell command-line interface is much faster to use, more powerful, easily extensible, and does not give the user RSI from having to use a mouse all the time. Below we will briefly describe the bash shell (bash). It is heavily based on the original UNIX shell, Bourne shell, and in fact its name is an acronym for Bourne Again Shell. Many other shells are also in use (ksh, csh, etc.), but, bash is the default shell in most Linux systems.

When the shell starts up, it initializes itself, then types a prompt character, often a percent or dollar sign, on the screen and waits for the user to type a command line.

When the user types a command line, the shell extracts the first word from it, assumes it is the name of a program to be run, searches for this program, and if it finds it, runs the program. The shell then suspends itself until the program terminates, at which time it tries to read the next command. What is important here is simply the observation that the shell is an ordinary user program. All it needs is the ability to read from the keyboard and write tothe monitor and the power to execute other programs.

Commands may take arguments, which are passed to the called program as character strings. For example, the command line

cpsrc dest

Invokes the cp program with two arguments, src and dest. This program interprets the first one to be the name of an existing file. It makes a copy of this file and calls the copy dest.

Not all arguments are file names. In head -20 file the first argument, -20, tells head to print the first 20 lines of file, instead of the default number of lines, 10. Arguments thatcontrol the operation of a command or specify an optional value are called flags, and by convention are indicated with a dash. The dash is required to avoid ambiguity, because the command head 20 file is perfectly legal, and tells head to first print the initial 10 lines of a file called 20, and then print the initial 10 lines of a second file called file. Most Linux commands accept multiple flags and arguments.

To make it easy to specify multiple file names, the shell accepts magic characters, sometimes called wild cards. An asterisk, for example, matches all possible strings, so

Is *.C

tellsls to list all the files whose name ends in .c. If files named x.c, y.c, and z.c all exist, the above command is equivalent to typing

Is x.cy.cz.c

Another wild card is the question mark, which matches any one character. A list of characters insidesquare brackets selects any of them, so

Is [ape]*

Listsall files beginning with «a", "p", or "e".

A program like the shell does not have to open the terminal in order to read from it or write to it. Instead, when it starts up, it automatically has access to a file called standard input (for reading), a file called standard output (for writing normal output), and a file called standard error (for writing error messages). Normally, all three default

to the terminal, so that reads from standard input come from the keyboard and writes to standard output or standard error go to the screen. Many Linux programs read from standard input and write to standard output as the default. For example,

sort

invokes the sort program, which reads lines from the terminal (until the user types a CTRL-D, to indicate end of file), sorts them alphabetically, and writes the result to the screen. It is also possible to redirect standard input and standard output, as that is often useful. The syntax for redirecting standard input uses a less than sign (<) followed by the input file name. Similarly, standard output is redirected using a greater than sign (>). It is permitted to redirect bothin the same command. For example, the command

sort<in >OUt

causes sort to take its input from the file in and write its output to the file out. Since standard error has not been redirected, any error messages go to the screen. A program that reads its input from standard input, does some processing on it, and writes its output to standard output is called a filter.

Consider the following command line consisting of three separate commands:

sort<in >temp; head -30 <temp; rm temp

It first runs sort, taking the input from in and writing the output to temp. When that has been completed, the shell runs head, telling it to print the first 30 lines of temp and print them on standard output, which defaults to the terminal. Finally, the temporary file is removed.

It frequently occurs that the first program in a command line produces output that is used as the input on the next program. In the above example, we used the file temp to hold this output. However, Linux provides a simpler construction todo the same thing. In

sort<in I head -30

the vertical bar, called the pipe symbol, says to take the output from sort and useit as the input to head, eliminating the need for creating, using, and removing the temporary file. A collection of commands connected by pipe symbols, called a pipeline, may contain arbitrarily many commands. A four-component pipeline isshown by the following example:

grepter *.t I sort I head -20 I tail -5 >foo

Here all the lines containing the string "ter" in all the files ending in .t are written to standard output, where they are sorted. The first 20 of these are selected out by head,

which passes then to tail, which writes the last five (i.e., lines 16 to 20 in the sorted list) to foo. This is an example of how Linux provides basic building blocks (numerous filters), each of which does one job, along with a mechanism for them to be put together in almost limitless ways.

Linux is a general-purpose multiprogramming system. A single user can run several programs at once, each as a separate process. The shell syntax for running a process in the background is to follow its command with an ampersand. Thus

we -I <a >b &

runs the word-count program, we, to count the number of lines (-1 flag) m 1ts input, a, writing the result to b, but does it in the background. As soon as the command has been typed, the shell types the prompt and is ready to accept and handle the next command. Pipelines can also be put in the background, for example, by

sort<X I head &

Multiple pipelines can run in the background simultaneously. It is possible to put a list of shell commands in a file and then start a shell with this file as standard input. The (second) shell just processes them in order, the same as it would with commands typed on the keyboard. Files containing shell commands are called shell scripts. Shell scripts may assign values to shell variables and then read them later. They may also have parameters, and use if, for, while, and case constructs. Thus a shell script is really a program written in shell language. The Berkeley C shell is an alternative shell that has been designed to make shell scripts (and the command language in general) look like C programs in many respects. Since the shell is just another user program, other people have written and distributed a variety of other shells.

Structure of the Linux kernel.

The kernel sits directly on the hardware and enables interactions with 110 devices and the memory management unit and controls CPU access to them. At the lowest level, as shown in Fig. 10-3 it contains interrupt handlers, which are the primary way for interacting with devices, and the low-level dispatching mechanism. This dispatching occurs when an interrupt happens. The low-level code here stops the running process, saves its state in the kernel process structures, and starts the appropriate driver. Process dispatching also happens when the kernel completes some operations and it is time to start up a user process again. The dispatching code is in assembler and is quite distinct from scheduling.

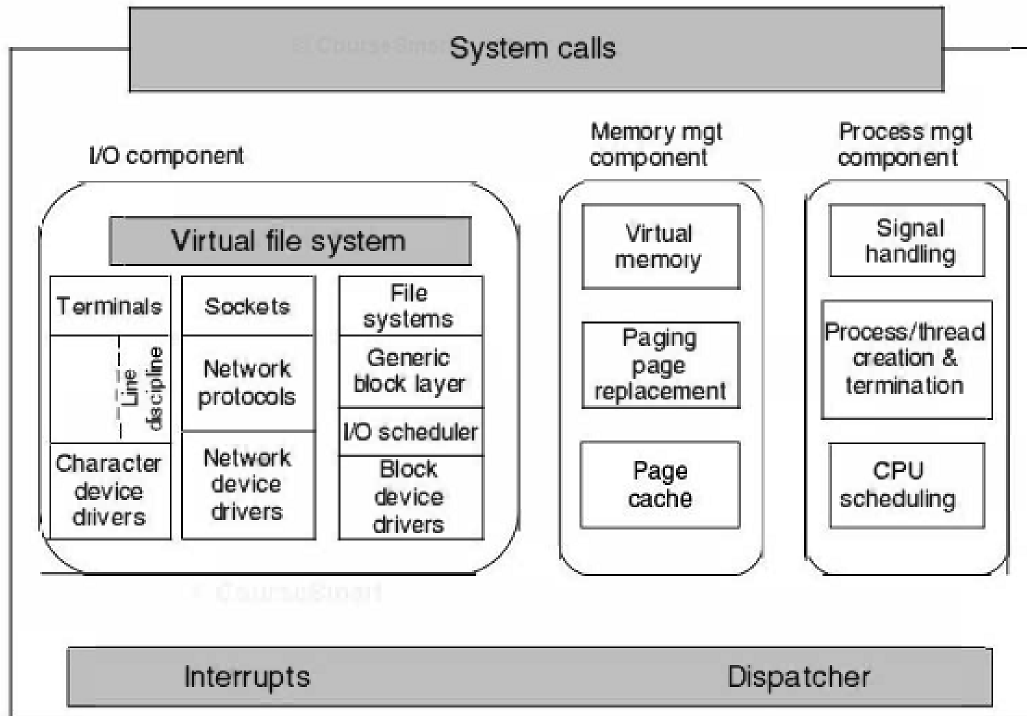

Figure 10-3. Structure of the Linux kernel

Next, we divide the various kernel subsystems into three main components. The I/O component in Fig. 10-3 contains all kernel pieces responsible for interacting with devices and performing network and storage I/O operations. At the highest level, the I/O operations are all integrated under a Virtual FileSystem layer. That is, at the top level, performing a read operation to a file, whether it is in memory or on disk, is the same as performing a read operation to retrieve a character from a terminal input. A the lowest level, all 110 operations pass through some device driver. All Linux drivers are classified as either character device drivers or block device drivers, with the main difference that seeks and random accesses are allowed on block devices and not on character devices. Technically, network devices are really character devices, but they are handledsomewhat differently, so that it is probably clearer to separate them, as has been done in the figure. Above the device driver level, the kernel code is different for each device type. Character devices may be used in two different ways. Some programs, such as visual editors like vi and emacs, want every key stroke as it is hit. Raw terminal (tty)/O makes this possible. Other software, such as the shell, is line oriented, and allows users to edit the whole line before hitting ENTER to send it to the program. In this case the character stream from the terminal device is passed through a so called line discipline, and appropriate formatting is applied.

Networking software is often modular, with different devices and protocols supported. The layer above the network drivers handles a kind of routing function, making sure that the right packet goes to the right device or protocol handler. Most Linux systems contain the full functionality of a hardware router within the kernel, although the performance is less than that of a hardware router. Above the router code is the actual protocol stack, always including IP and TCP, but also many additional protocols. Overlaying all the network is the socket interface, which allows programs to create sockets for particular networks and protocols, getting back a file descriptor for each socket to use later.
On top of the disk drivers is the 1/0 scheduler, which is responsible for ordering and issuing disk operation requests in a way that tries to conserve wasteful disk head movement or to meet some other system policy.
At the very top of the block device column are the file systems. Linux may have, and it does in fact, multiple file systems coexisting concurrently. In order to hide the gruesome architectural differences of various hardware devices from the file system implementation, a generic block device layer provides an abstraction used by all file systems.
To the right in are the other two key components of the Linux kernel. These are responsible for the memory and process management tasks. Memory management tasks include maintaining the virtual to physical memory mappings, maintaining a cache of recently accessed pages and implementing a good page replacement policy, and on-demand bringing in new pages of needed code and data into memory.
The key responsibility of the process management component is the creation and termination of processes. It also includes the process scheduler, which chooses which process or, rather, thread to run next. As we shall see in the next section, the Linux kernel treats both processes and threads simply as executable entities, and will schedule them based on a global scheduling policy. Finally, code for signal handling also belongs to this component.

While the three components are represented separately in the figure, they are highly interdependent. File systems typically access files through the block devices. However, in order to hide the large latencies of disk accesses, files are copied into the page cache in main memory. Some files may even be dynamically created and may only have an in-memory representation, such as files providing.

Some runtime resource usage information. In addition, the virtual memory system may rely on a disk partition or in-file swap area to back up parts of the main memory when it needs to free up certain pages, and therefore relies on the I/0 component. Numerous other interdependencies exist.

In addition to the static in-kernel components, Linux supports dynamically loadable modules. These modules can be used to add or replace the default device drivers, file system, networking, or other kernel codes. The modules are not shown in Fig. 10-3.

Finally, at the very top is the system call interface into the kernel. All system calls come here, causing a trap which switches the execution from user mode into protected kernel mode and passes control to one of the kernel components described above.

Discuss Linux scheduling.

Linux threads are kernel threads, so scheduling is based on threads, not processes. Linux distinguishes three classes of threads for scheduling purposes:

1. Real-time FIFO.
2. Real-time round robin.
3. Timesharing.

Real-time FIFO threads are the highest priority and are not preemptable except by a newly readied real-time FIFO thread with higher priority. Real-time round-robin threads are the same as real-time FIFO threads except that they have time quanta associated with them, and are preemptable by the clock. If multiple real-time round-robin threads are ready, each one is run for its quantum, after which it goes to the end of the list of real-time round-robin threads. Neither of these classes is actually real time in any sense. Deadlines cannot be specified and guarantees are not given. These classes are simply higher priority than threads in the standard timesharing class. The reason Linux calls them real time is that Linux is conformant to the P1003.4 standard ("real-time" extensions to UNIX) which uses those names. The real-time threads are internally represented with priority levels from 0 to 99, 0 being the highest and 99 the lowest real-time priority level.

The conventional, non-real-time threads are scheduled according to the following algorithm. Internally, the non-real-time threads are associated with priority levels from 100 to 139, that is, Linux internally distinguishes among 140 priority levels. As for the real-time round robin threads, Linux associates time quantum values for each of the non-real-time priority levels. The quantum is the number of clock ticks the thread may continue to run for. In the current Linux version, the clock runs at 1000Hz and each tick is lms, which is called a jiffy.

Like most UNIX systems, Linux associates a nice value with each thread. The default is 0, but this can be changed using the nice(value) system call, where value ranges from -20 to +19. This value determines the static priority of each thread.

A key data structure used by the Linux scheduler is a runqueue. A runqueue is associated with each CPU in the system, and among other information, it maintains two arrays, active and expired. As shown in Fig. 10-10, each of these fields is a pointer to an array of

140 list heads, each corresponding to a different priority. The list head points to a doubly linked list of processes at a given priority.

The scheduler selects a task from the highest-priority active array. If that task's timeslice (quantum) expires, it is moved to an expired list (potentially at a different priority level). If the task blocks, for instance to wait on an 1/0 event, before its time slice expires, once the event occurs and its execution can resume, it is placed back on the original active array, and its timeslice is decremented to reflect the CPU time it already consumed. Once its timeslice is

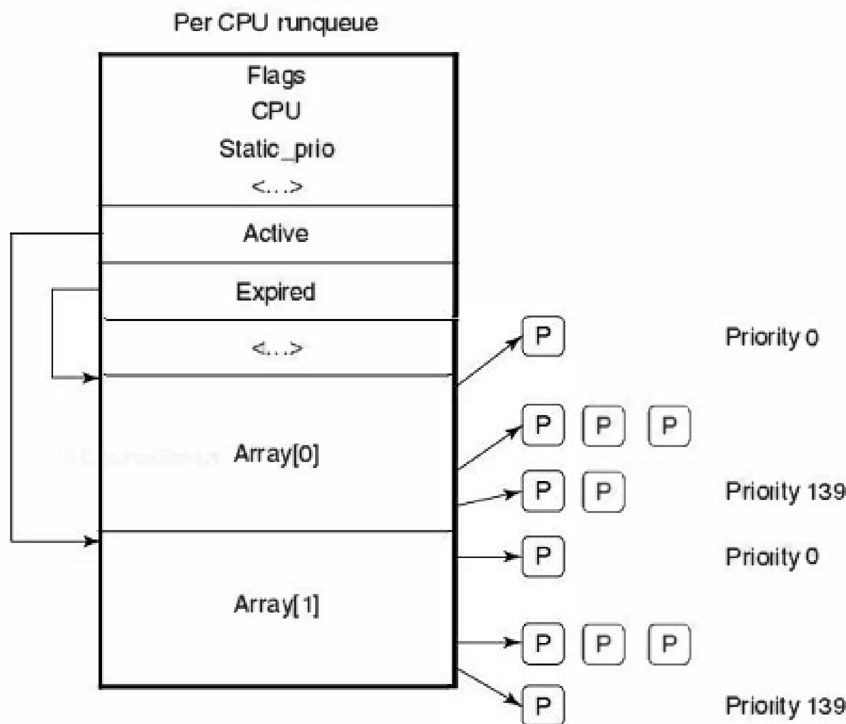

Figure 10-10. Illustration of Linux runqueue and priority arrays.

Fully exhausted, it too will be placed on an expired array. When there are no more tasks in any of the active arrays, the scheduler simply swaps the pointers, so the expired arrays now become active, and vice versa. This method ensures that low-priority tasks will not starve.

Different priority levels are assigned different timeslice values. Linux assigns higher quanta to higher-priority processes. For instance, tasks running at priority level 100 will receive time quanta of 800 msec, whereas tasks at priority level of 139 will receive 5 msec.

The idea behind this scheme is to get processes out of the kernel fast. If a process is trying to read a disk file, making it wait a second between read calls will slow it down enormously. It is far better to let it run immediately after each request is completed, so that it can make the next one quickly. Similarly, if a process was blocked waiting for keyboard input, it is clearly an interactive process, and as such should be given a high priority as soon as it is ready in order to ensure that interactive processes get good service. In this light, CPU-bound processes basically get any service that is left over when all the 110 bound and interactive processes are blocked.
Since Linux does not know a priori whether a task is I/O-or CPU-bound, it relies on continuously maintaining interactivity heuristics. In this manner, Linux distinguishes between static and dynamic priority. The threads' dynamic priority is continuously recalculated, so as to

(1) Reward interactive threads, and

(2) Punish CPU-hogging threads.

The maximum priority bonus is -5, since lower-priority values correspond to higher priority received by the scheduler. The maximum priority penalty is +5.

More specifically, the scheduler maintains a sleep_avg variable associated with each task. Whenever a task is awakened, this variable is incremented, whenever a task is preempted or its quantum expires, this variable is decremented by the corresponding value. This value is used to dynamically map the task's bonus to values from -5 to +5. The Linux scheduler recalculates the new priority level as a thread is moved from the active to the expired list.
In addition, the scheduler includes features particularly useful for multiprocessor or multicore platforms. First, the runqueue structure is associated with each CPU in the multiprocessing platform. The scheduler tries to maintain benefits from affinity scheduling, and to schedule tasks on the CPU on which they were previously executing. Second, a set of system calls is available to further specifyor modify the affinity requirements of a select thread. Finally, the scheduler performs periodic load balancing across runqueue of different CPUs to ensure thatthe system load is well balanced, while still meeting certain performance or affinity requirements.
The scheduler considers only runnable tasks, which are placed on the appropriate runqueue. Tasks which are not runnable and are waiting on various I/0 operations or other kernel events are placed on another data structure, waitqueue.
A waitqueue is associated with each event that tasks may wait on. The head of the waitqueue includes a pointer to a linked list of tasks and a spinlock. The spinlock is necessary so as to ensure that the waitqueue can be concurrently manipulated through both the main kernel code and interrupt handlers or other asynchronous invocations.

In fact, the kernel code contains synchronization variables in numerous locations. Earlier Linux kernels had just one big kernel lock (BLK). This proved highly inefficient, particularly on multiprocessor platforms, since it prevented processes on different CPUs from executing kernel code concurrently. Hence, many new synchronization points were introduced at much finer granularity.

Write short notes on Linux NFS.

NFS Architecture

The basic idea behind NFS isto allow an arbitrary collection ofclients and servers to share a common file system. In many cases, all the clients and servers are on the same LAN, but this is not required. It is also possible to run NFS over a wide area network if the server is far from the client.

Each NFS server exports one or more of its directories for access by remote clients. When a directory is made available, so are all of its subdirectories, so in fact, entire directory trees are normally exported as a unit. The list of directories a server exports is maintained in a file, often /etc/exports, so these directories can be exported automatically whenever the server is booted. Client's access exported directories by mounting them. When a client mounts a (remote) directory, it becomes part of its directory hierarchy, as shown in .

In this example, client 1 has mounted the bin directory of server 1 on its own bin directory, so it can now refer to the shell as /binlsh and get the shell on server 1. Diskless workstations often have only a skeleton file system (in RAM) and get all their files from remote servers like this. Similarly, client 1 has mounted server 2's directory! Projects on its directory /usrlast/work so it can now access file a as /usr!ast/worklprojl!a. Finally, client 2has also mounted the projects directory and can also access file a, only as lmntlprojl!a. As seen here, the same file can have different names on different clients due to its being mounted in a different place in the respective trees. The mount point is entirely local to the clients; the server does not know where it is mounted on any of its clients.

NFS Protocols

Since one of the goals of NFS is to support a heterogeneous system, with clients and servers possibly running different operating systems on different hardware, it is essential that the interface between the clients and servers be well defined. Only then is it possible for anyone to be able to write a new client implementation and expect it to work correctly with existing servers, and vice versa.

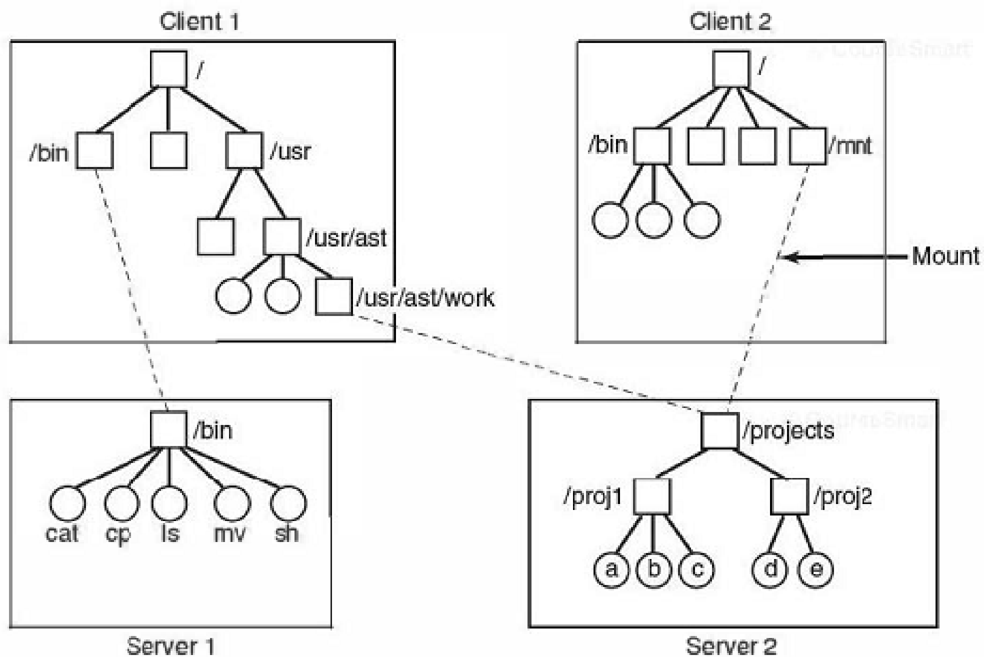

Figure 10-35. Examples of remote mounted file systems. Directories are shown as squares and files are shown as circles.

NFS accomplishes this goal by defining two client-server protocols. A protocol is a set of requests sent by clients to servers, along with the corresponding replies sent by the servers back to the clients.

The first NFS protocol handles mounting. A client can send a path name to a server and request permission to mount that directory somewhere in its directory hierarchy. The place where it is to be mounted is not contained in the message, as the server does not care where it is to be mounted. If the path name is legal and the directory specified has been exported, the server returns a file handle to the client. The file handle contains fields uniquely identifying the file system type, the disk, the i-node number of the directory, and security information. Subsequent calls to read and write files in the mounted directory orany of its subdirectories use the file handle.

When Linux boots, it runs the /etc/rc shell script before going multiuser. Commands to mount remote file systems can be placed in this script, thus automatically mounting the necessary remote file systems before allowing any logins.

Alternatively, most versions of Linux also support automounting. This feature allows a set of remote directories to be associated with a local directory. None of these remote directories are mounted (or their servers even contacted) when the client is booted. Instead, the first time a remote file is opened, the operating system sends a message to each of the servers. The first

one to reply wins, and its directory is mountedAutomounting has two principal advantages over static mounting via the

/etc/rc file. First, if one of the NFS servers named in /etc/rc happens to be down,

it is impossible to bring the client up, at least not without some difficulty, delay, and quite a few error messages. If the user does not even need that server at the moment, all that work is wasted. Second, by allowing the client to try a set of servers in parallel, a degree of fault tolerance can be achieved (because only one of them needs to be up), and the performance can be improved (by choosing the first one to reply-presumably the least heavily loaded).

On the other hand, it is tacitly assumed that all the file systems specified as alternatives for the automount are identical. Since NFS provides no support for file or directory replication, it is up to the user to arrange for all the file systems to be the same. Consequently, automounting is most often used for read-only file systems containing system binaries and other files that rarely change.

The second NFS protocol is for directory and file access. Clients can send messages to servers to manipulate directories and read and write files. They can also access file attributes, such as file mode, size, and time of last modification. Most Linux system calls are supported by NFS, with the perhaps surprising exceptions of open and close.

The omission of open and close is not an accident. It is fully intentional. It is not necessary to open a file before reading it, or to close it when done. Instead, to read a file, a client sends the server a lookup message containing the file name, with a request to look it up and return a file handle, which is a structure that identifies the file (i.e., contains a file system identifier and i-node number, among other data). Unlike an open call, this lookup operation does not copy any information into internal system tables. The read call contains the file handle of the file to read, the offset in the file to begin reading, and the number of bytes desired.

Each such message is self-contained. The advantage of this scheme is that the server does not have to remember anything about open connections in between calls to it. Thus if a server crashes and then recovers, no information about open files is lost, because there is none. A server like this that does not maintain state information about open files is said to be stateless.

Unfortunately, the NFS method makes it difficult to achieve the exact Linux file semantics. For example, in Linux a file can be opened and locked so that other processes cannot access it. When thefile is closed, the locks are released.

In a stateless server such as NFS, locks cannot be associated with open files, because the server does not know which files are open. NFS therefore needs a separate, additional mechanism to handle locking.

NFS Implementation

Although the implementation of the client and server code is independent of the NFS protocols, most Linux systems use a three-layer implementation similar to that of Fig. 10-36. The top layer is the system call layer. This handles calls like open, read, and close. After parsing the call and checking the parameters, it invokes the second layer, the Virtual File System (VFS) layer.

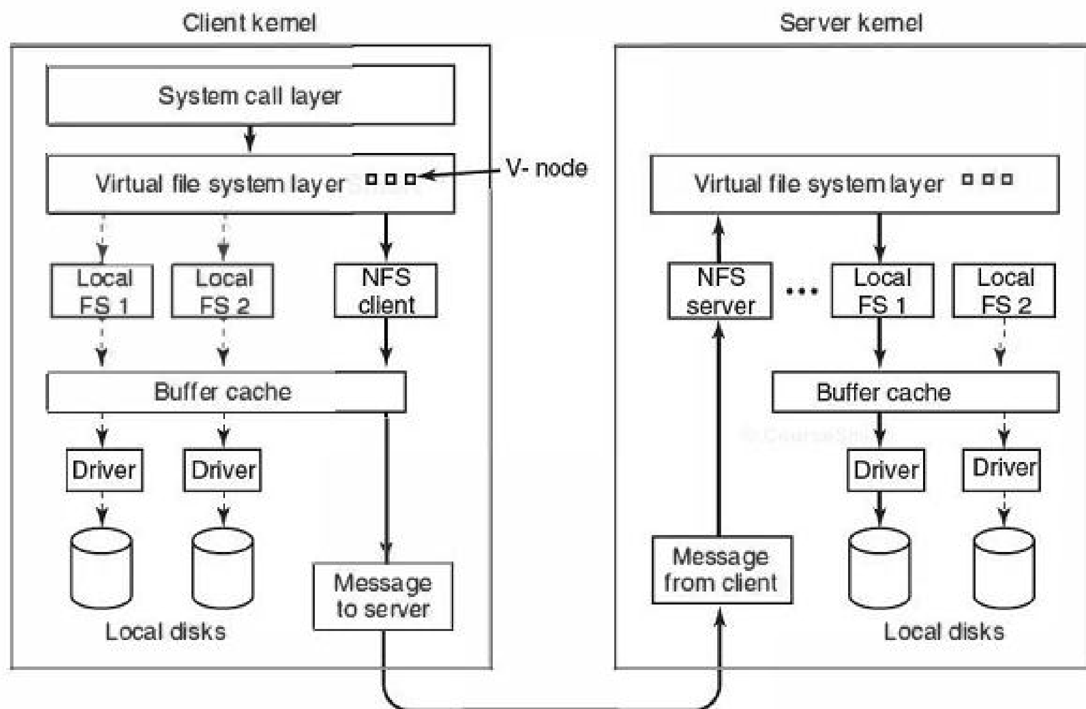

Figure 10-36. The NFS layer structure

The task of the VFS layer is to maintain a table with one entry for each open file. The VFS layer has an entry, a virtual i-node, or v-node, for every open file. V -nodes are used to tell whether the file is local or remote. For remote files, enough information is provided to be able to access them. For local files, the file system and i-node are recorded because modem Linux systems can support multiple file systems (e.g., ext2fs, /proc, FAT, etc.). Although VFS was invented to support NFS, most modern Linux systems now support it as an integral part of the operating system, even if NFS is not used.

To see how v-nodes are used, let us trace a sequence of mount, open, and read system calls. To mount a remote file system, the system administrator (or /etc/rc) calls the mount program specifying the remote directory, the local directory on which it is to be mounted, and other information. The mount program parses the name of the remote directory to be mounted and discovers the name of the NFS server on which the remote directory is located. It then contactsthat machine, asking for a file handle for the

remote directory. If the directory exists and is available for remote mounting, the server returns a file handle for the directory.

Finally, it makes a mount system call, passing the handle to the kernel. The kernel then constructs a v-node for the remote directory and asks the NFS client code in Fig. 10-36 to create an r-node (remote i-node) in its internal tables to hold the file handle. The v-node points to the r-node. Each v-node in the VFS layer will ultimately contain either a pointer to an r-node in the NFS client code, or a pointer to an i-node in one of the local file systems (shown as dashed lines in Fig. 10-36). Thus from the v-node it is possible to see if a file or directory is local or remote. If it is local, the correct file system and i-node can be located. If it is remote, the remote host and file handle can be located.

When a remote file is opened on the client, at some point during the parsing of the path name, the kernel hits the directory on which the remote file system is mounted. It sees that this directory is remote and in the directory's v-node finds the pointer to the r-node. It then asks the NFS client code to open the file. The NFS client code looks up the remaining portion of the path name on the remote server associated with the mounted directory and gets back a file handle for it. It makes an r-node for the remote file in its tables and reports back to the VFS layer, which puts in its tables a v-node for the file that points to the r-node. Again here we see that every open file or directory has a v-node that points to either an r-node or an i-node.

The caller is given a file descriptor for the remote file. This file descriptor is mapped onto the v-node by tables in the VFS layer. Note that no table entries are made on the server side. Although the server is prepared to provide file handles upon request, it does not keep track of which files happen to have file handles outstanding and which do not. When a file handle is sent to it for file access, it checks the handle, and if it is valid, uses it. Validation can include verifying an authentication key contained in the RPC headers, if security is enabled.

When the file descriptor is used in a subsequent system call, for example, read, the VFS layer locates the corresponding v-node, and from that determines whether it is local or remote and also which i-node or r-node describes it. It then sends a message to the server containing the handle, the file offset (which is maintained on the client side, not the server side), and the byte count. For efficiency reasons, transfers between client and server are done in large chunks, normally 8192 bytes, even if fewer bytes are requested.

When the request message arrives at the server, it is passed to the VFS layer there, which determines which local file system holds the requested file. The VFS layer then makes a call to that local file system to read and return the bytes. These data are then passed back to the client. After the client's VFS layer has gotten the8-KB chunk it asked for, it automatically issues a request for the next chunk, so it will have it should it be needed shortly. This feature, known as read ahead, improves performance considerably.

For writes an analogous path is followed from client to server. Also, transfers are done in 8-KB chunks here too. If a write system call supplies fewer than 8 KB bytes of data, the data are just accumulated locally. Only when the entire 8-KB chunk is full is it sent to the server. However, when a file is closed, all of its data are sent to the server immediately.

Another technique used to improve performance is caching, as in ordinary UNIX. Servers cache data to avoid disk accesses, but this is invisible to the clients. Clients maintain two caches, one for file attributes (i-nodes) and one for file data. When either an i-node or a file block is needed, a check is made to see if it can be satisfied out of the cache. If so, network traffic can be avoided.

While client caching helps performance enormously, it also introduces some nasty problems. Suppose that two clients are both caching the same file block and that one of them modifies it. When the other one reads the block, it gets the old (stale) value. The cache is not coherent.

Given the potential severity of this problem, the NFS implementation does several things to mitigate it. For one, associated with each cache block is a timer.

When the timer expires, the entry is discarded. Normally, the timer is 3 sec for data blocks and 30 sec for directory blocks. Doing this reduces the risk somewhat. In addition, whenever a cached file is opened, a message is sent to the server to find out when the file was last modified. If the last modification occurred after the local copy was cached, the cache copy is discarded and the new copy fetched from the server. Finally, once every 30 sec a cache timer expires, and all the dirty (i.e., modified) blocks in the cache are sent to the server. While not perfect, these patches make the system highly usable in most practical circumstances.

WINDOWS REGISTRY.

Windows attaches a special kind of file system (optimized for small files) to the NT namespace. This file system is called the registry. The registry is organized into separate volumes called hives. Each hive is kept in a separate file (in the directory C:\ Windows\system32\config\ of the boot volume). When a Windows system boots, one particular hive named SYSTEM is loaded into memory by the same boot program that loads the kernel and other boot files, such as boot drivers, from the boot volume.

Windows keeps a great deal of crucial information in the SYSTEM hive, including information about what drivers to use with what devices, what software to run initially, and many parameters governing the operation of the system. This information is used even by the boot program itself to determine which drivers are boot drivers, being needed immediately upon boot. Such drivers include thosethat understand the file system and disk drivers for the volume containing the operating system itself.

Other configuration hives are used after the system boots to describe information about the software installed on the system, particular users, and the classes of user-mode COM (Component Object-Model) objects that are installed on the system. Login information for local users is kept in the SAM (Security Access Manager) hive. Information for network

users is maintained by the [sass service in the SECURITY hive, and coordinated with the network directory servers so that users can have a common account name and password across an entire network. A list of the hives used in Windows Vista is shown .

The registry gathers these files into a central store, which is available early in the process of booting the system. This is important for implementing Windows plug-andplay functionality. But the registry has become very disorganized as Windows has evolved. There are poorly defined conventions about how the configuration information should be arranged, and many applications take an ad hoc approach.

Most users, applications, and all drivers run with full privileges, and frequently modify system parameters in the registry directly-sometimes interfering with each other and destabilizing the system.

Hive file	Mounted name	Use
SYSTEM	HKLMTEM	OS configuration information, used by kernel
HARDWARE	HKLM DWARE	In-memory hive recording hardware detected
BCD	HKLM BCD*	Boot Configuration Database
SAM	HKLM	Local user account information
SECURITY	HKLM URITY	lsass' account and other security information
DEFAULT	HKEY_USERS .DEFAULT	Default hive for new users
NTUSER.DAT	HKEY_USERS <user id>	User-specific hive, kept in home directory
SOFTWARE	HKLM TWARE	Application classes registered by COM
COMPONENTS	HKLM NENTS	Manifests and dependencies for sys. components

Figure 11-11. The registry hives in Windows Vista. HKLM is a short-hand for *HKEY_LOCAL_MACHINE*.

The registry is a strange cross between a file system and a database, and yet really unlike either.

To explore the registry Windows has a GUI program called regedit that allows you to open and explore the directories (called keys) and data items (called values).

Procmon watches all the registry accesses that take place in the system and is very illuminating. Some programs will access the same key over and over tens of thousands of times.

As the name implies, regedit allows users to edit the registry-but be very careful if you ever do. It is very easy to render your system unable to boot, or damage the installation of applications so that you cannot fix them without a lot of wizardry.

Beginning with Windows Vista Microsoft has introduced a kernel-based transaction manager with support for coordinated transactions that span both file system and registry operations. Microsoft plans to use this facility in the future to avoid some of the metadata corruption problems that occur when software installation does not complete correctly and leaves around partial state in the system directories and registry hives.

The registry is accessible to the Win32 programmer. There are calls to create and delete keys, look up values within keys, and more. Some of the more useful ones are listed in Fig. 11-12.

Win32 API function	Description
RegCreateKeyEx	Create a new registry key
RegDeleteKey	Delete a registry key
RegOpenKeyEx	Open a key to get a handle to it
RegEnumKeyEx	Enumerate the subkeys subordinate to the key of the handle
RegQueryValueEx	Look up the data for a value within a key

Figure 11-12. Some of the Win32 API calls for using the registry

When the system is turned off, most of the registry information is stored on the disk in the hives. Because their integrity is so critical to correct system functioning, backups are made automatically and metadata writes are flushed to disk to prevent corruption in the event of a system crash. Loss of the registry requires reinstalling all software on the system.

Write note on Job, Process, threads and Fiber management API calls.

New processes are created using the Win32 API function CreateProcess. This function has many parameters and lots of options. It takes the name of the file to be executed, the command-line strings (unparsed), and a pointer to the environment strings. There are also flags and values that control many details such as how security is configured for the process and first thread, debugger configuration, and scheduling priorities. A flag also specifies whether open handles in the creator are to be passed to the new process. The function also takes the current working directory for the new process and an optional data structure with information about the GUI Window the process is to use. Rather than returning just a process ID for the new process, Win32 returns both handles and IDs, both for the new process and for its initial thread.

The large number of parameters reveals a number of differences from the design of process creation in UNIX.

1. The actual search path for finding the program to execute is buried in the library code for Win32, but managed more explicitly in UNIX.
2. The current working directory is a kernel-mode concept in UNIX but a user-mode string in Windows. Windows does open a handle on the current directory for each process, with the same annoying effect as in UNIX: You cannot delete the directory, unless it happens to be across the network, in which case you can delete it.
3. UNIX parses the command line and passes an array of parameters, while Win32 leaves argument parsing up to the individual program. As a consequence, different programs may handle wildcards (e.g., *.txt) and other special symbols in an inconsistent way.

4. Whether file descriptors can be inherited in UNIX is a property of the handle. In Windows it is a property of both the handle and a parameter to process creation.
5. Win32 is GUI-oriented, so new processes are directly passedinformation abouttheir primary window, while this information is passed as parameters to GUI applications in UNIX.
6. Windows does not have a SETUID bit as a property of the executable, but one process can create a process that runs as a different user, as long as it can obtain a token with that user's credentials.
7. The process and thread handle returned from Windows can be used to modify the new process/thread in many substantive ways, including duplication of handles and setting up the environment variables in the new process. UNIX just makes modifications to the new process between the fork and exec calls.

Some of these differences are historical and philosophical. UNIX was designed to be command-line-oriented rather than Gill-oriented like Windows.

UNIX users are more sophisticated, and understand concepts like PATH variables. Windows Vista inherited a lot of legacy from MS-DOS.

The comparison is also skewed because Win32 is a user-mode wrapper around the native NT process execution, much as the system library function wraps fork/exec in UNIX. The actual NT system calls for creating processes and threads, NtCreateProcess and NtCreateThread, are much simpler than the Win32 versions. The main parameters to NT process creation are a handle on a section representing the program file to run, a flag specifying whether the new process should, by default, inherit handles from the creator, and parameters related to the security model. All the details of setting up the environment strings, and creating the initial thread, are left to user-mode code that can use the handle on the new process to manipulate its virtual address space directly.

To support the POSIX subsystem, native process creation has an option to create a new process by copying the virtual address space of another process rather than mapping a section object for a new program. This is only used to implement fork for POSIX, and not by Win32.

Thread creation passes the CPU context to use for the new thread (which includes the stack pointer and initial instruction pointer), a template for the TEB,and a flag saying whether the thread should be immediately run or created in a suspended state (waiting for somebody to call NtResumeThread on its handle). Creation of the user-mode stack and pushing of the argvlargc parameters is left to user-mode code calling the native NT memory management APis on the process handle.

In the Windows Vista release, a new native API for processes was included which moves many of the user-mode steps into the kernel-mode executive, and combines process creation with creation of the initial thread. The reason for the change was to support the use of processes as security boundaries. Normally, all processes created by a user are considered to be equally trusted. It is the user, as represented by a token that determines where the trust

boundary is. This change in Windows Vista allows processes to also provide trust boundaries, but this means that the creating process does not have sufficient rights regarding a new process handle to implement the details of process creation in user mode.

Write note on Process and Threads in SYMBIAN os.

Symbian OS is a multitasking operating system that uses the concepts of processes and threads much like other operating systems do. However, the structure of the Symbian OS kernel and the way it approaches the possible scarcity of resources influences the way that it views these multitasking objects.

Threads and Nano threads

Instead of processes as the basis for multitasking, Symbian OS favors threads and is built around the thread concept. Threads form the central unit of multitasking. A process is simply seen by the operating system as a collection of threads with a process control block and some memory space.

Thread support in Symbian OS is based in the nanokernel with nanothreads. The nanokernel provides only simple thread support; each thread is supported by a nanokernel-based nanothread. The nanokernel provides for nanothread scheduling, synchronization, and timing services. Nanothreads run in privileged mode and need a stack to store their run-time environment data. Nanothreads cannot run in user mode. This fact means that the operating system can keep close, tight control over each one. Each nanothread needs a very minimal set of data to run: basically, the location of its stack and how big that stack is. The operating system keeps control of everything else, such as the code each thread uses, and stores a thread's context on its run-time stack. Nanothreads have thread states like processes have states. The model used by the Symbian OS nanokernel adds a few states to the basic model. In addition to the basic states, nanothreads can be in the following states:

1. Suspended. This is when a thread suspends another thread and is meant to be different from the waiting state, where a thread is blocked by some upper layer object (e.g., a Symbian OS thread).
2. Fast Semaphore Wait. A thread in this state is waiting for a fast semaphore a type of sentinel variable to be signaled. Fast semaphores are nanokernel level semaphores.
3. DFC Wait. A thread in this state is waiting for a delayed function call or DFC to be added to the DFC queue. DFCs are used in device driver implementation. They represent calls to the kernel that can be queued and scheduled for execution by the Symbian OS kernel layer.
4. Sleep. Sleeping threads are waiting for a specific amount of time to elapse.
5. Other. There is a generic state that is used when developers implement extra states for nanothreads. Developers do this when they extend the nanokernel functional for new phone platforms (called personality layers). Developers who do

this must also implement how states are transitioned to and from their extended implementations.

A nanothreadis essentially anultra-light-weight process. It has a mini-context that getsswitched as nanothreads get moved onto and out of the processor. Each nanothread has a state, as do processes. The keys to nanothreads are the tight control that the nanokernel has over them and the minimal data that make up the context of each one.

Symbian OS threads build upon nanothreads; the kernel adds support beyond what the nanokernel provides. User mode threads that are used for standard applications are implemented by Symbian OS threads. Each Symbian OS thread contains ananothread and addsits own run-time stack to the stack the nanothread uses. Symbian OS threads can operate in kernel mode via system calls. Symbian OS also add exception handling and exit signaling to the implementation.
Symbian OS threads implement their own set of states on top of the nanothread implementation. Because Symbian OS threads add some functionality to the minimal nanothread implementation, the new states reflect the new ideas built into Symbian OS threads. Symbian OS adds seven new states that Symbian OS threads can be in, focused on special blocking conditions that can happen to a Symbian OS thread. These special states include waiting and suspending on (normal) semaphores, mutex variables, and condition variables. Remember that, because of the implementation of Symbian OS threads on top of nanothreads, these states are implemented in terms of nanothread states, mostly by using the suspended nanothread state in various ways.

Processes

Processes in Symbian OS, then, are Symbian OS threads grouped together under a single process control block structure with a single memory space. There may be only a single thread of execution or there may be many threads less than one process control block. Scheduling a process, then, is really implemented byscheduling a thread and initializing the right process control block to use for its data needs. Symbian OS threads organized under a single process work together in several ways. First, there is a single main thread that is marked as the starting point for the process. Second, threads share scheduling parameters. Changing parameters, that is, the method of scheduling, for the process changes the parameters for all threads. Third, threads share memory space objects, including device and other object descriptors. Finally, when a process is terminated, the kernel terminates all threads in the process.

Active Objects

Active objects are specialized forms of threads, implemented in a a way as to lighten the burden they place on the operating environment. Since Symbian OS is focused on communication, manyapplications have a similar pattern of implementation: they write data to a communication socket or send information through a pipe, and then they block as they wait for a response from the receiver. Active objects are designed so that when they are

brought back from this blocked state, they have a single entry point into their code that is called. This simplifies their implementation.

Since they run in user space, active objects have the properties of Symbian OS threads. As such they have their own nanothread and can join with other Symbian OS threads to form a process to the operating system. If active objects are just Symbian OS threads, one can ask what advantage the operating system gains from this simplified thread model. The key to active objects is in scheduling. While waiting for events, all active objects reside within a single process andcan act as a single thread to the system. The kernel does not need to continually check each active object to see if it can be unblocked. Active objects in a single process, therefore, can be coordinated by a single scheduler implemented in a single thread. By combining code that would otherwise be implemented as multiple threads into one thread, by building fixed entry points into the code, and by using a single scheduler to coordinate their execution, active objects form an efficient and lightweight version of standard threads.

It is important to realize where active objects fit into the Symbian OS process structure. When a conventional thread makes a system call that blocks its execution while in the waiting state, the operating system still needs to check the thread. Between context switches, the operating system will spend time checking blocked processes in the wait state, determining if any needs to move to the ready state.

Active objects place themselves in the wait state and wait for a specific event. Therefore, the operating system does not need to check them but moves them when their specific event has been triggered. The result is less thread checking and faster performance

Interprocess Communication

In a multithreaded environment like Symbian OS, interprocess communication is crucial to system performance. Threads, especially in the form of system servers, communicate constantly.

A socket is the basic communication model used by Symbian OS. It is an abstract communication pipeline between two endpoints. The abstraction is used to hide both the methods oftransport and the management of data between the endpoints. The concept of a socket is used by Symbian OS to communicate between clients and servers, from threads to devices, and between threads themselves.

The socket model also forms the basis of device VO. Again abstraction is the key to making this model so useful. All the mechanics of exchanging data with a device are managed by the operating system rather than by the application. Forexample, sockets that work over TCP/IP in a networking environment can be easily adapted to work over a Bluetooth environment by changing parameters in the type of socket used. Most of the rest of the data exchange work in such a switchover is done by the operating system.

Symbian OS implements the standard synchronization primitives that one would find in a general purpose operating system. Several forms of semaphores and mutexes are in wide use across the operating system. These provide for synchronizing processes and threads.