In response, Stallman started the GNU project (a recursively defined acronym

for “GNU’s not UNIX”) to develop an entire, freely available, UNIX-like system,

consisting of a kernel and all associated software packages, and encouraged others

to join him. In 1985, Stallman founded the Free Software Foundation (FSF), a non

profit organization to support the GNU project as well as the development of free

software in general.

When the GNU project was started, BSD was not free in the sense that Stallman meant.

Use of BSD still required a license from AT&T, and users could

not freely modify and redistribute the AT&T code that formed part of BSD.

One of the important results of the GNU project was the development of the GNU

General Public License (GPL), the legal embodiment of Stallman’s notion of free

software. Much of the software in a Linux distribution, including the kernel, is

licensed under the GPL or one of a number of similar licenses. Software licensed

under the GPL must be made available in source code form, and must be freely

redistributable under the terms of the GPL. Modifications to GPL-licensed software

are freely permitted, but any distribution of such modified software must also

be under the terms of the GPL. If the modified software is distributed in executable form,

the author must also allow any recipients the option of obtaining the

modified source for no more than the cost of distribution. The first version of the

GPL was released in 1989. The current version of the license, version 3, was

released in 2007. Version 2 of the license, released in 1991, remains in wide use,

and is the license used for the Linux kernel. (Discussions of various free software

licenses can be found in [St. Laurent, 2004] and [Rosen, 2005].)

The GNU project did not initially produce a working UNIX kernel, but did

produce a wide range of other programs. Since these programs were designed to

run on a UNIX-like operating system, they could be, and were, used on existing

UNIX implementations and, in some cases, even ported to other operating systems.

Among the more well-known programs produced by the GNU project are the

Emacs text editor, GCC (originally the GNU C compiler, but now renamed the

GNU compiler collection, comprising compilers for C, C++, and other languages),

the bash shell, and glibc (the GNU C library).

By the early 1990s, the GNU project had produced a system that was virtually

complete, except for one important component: a working UNIX kernel. The GNU

project had started work on an ambitious kernel design, known as the GNU/HURD,

based on the Mach microkernel. However, the HURD was far from being in a form

that could be released. (At the time of writing, work continues on the HURD,

which currently runs only on the x86-32 architecture.)

Because a significant part of the program code that constitutes what is commonly known as the Linux system actually derives from the GNU project, Stallman prefers to use the term GNU/Linux to refer to the entire system. The question of naming (Linux versus GNU/Linux) is the source of some debate in the free software community. Since this book is primarily concerned with the API of the Linux kernel, we’ll generally use the term Linux.

The Linux Kernel

In 1991, Linus Torvalds, a Finnish student at the University of Helsinki, was inspired to write an operating system for his Intel 80386 PC. In the course of his studies, Torvalds had come into contact with Minix, a small UNIX-like operating system kernel developed in the mid-1980s by Andrew Tanenbaum, a university professor in Holland. Tanenbaum made Minix, complete with source code, available as a tool for teaching operating system design in university courses. The Minix kernel could be built and run on a 386 system. However, since its primary purpose was as a teaching tool, it was designed to be largely independent of the hardware architecture, and it did not take full advantage of the 386 processor’s capabilities

Torvalds therefore started on a project to create an efficient, full-featured UNIX kernel to run on the 386. Over a few months, Torvalds developed a basic kernel that allowed him to compile and run various GNU programs. Then, on October 5, 1991, Torvalds requested the help of other programmers, making the following now much-quoted announcement of version 0.02 of his kernel in the comp.os.minix Usenet newsgroup:

Following a time-honored tradition of giving UNIX clones names ending with the letter X, the kernel was (eventually) baptized Linux. Initially, Linux was placed under a more restrictive license, but Torvalds soon made it available under the GNU GPL. The call for support proved effective. Other programmers joined Torvalds in the development of Linux, adding various features, such as an improved file system, networking support, device drivers, and multiprocessor support. By March 1994, the developers were able to release version 1.0. Linux 1.2 appeared in March 1995, Linux 2.0 in June 1996, Linux 2.2 in January 1999, and Linux 2.4 in January 2001. Work on the 2.5 development kernel began in November 2001, and led to the release of Linux 2.6 in December 2003.

The Core Operating System: The Kernel The term operating system is commonly used with two different meanings:

1)To denote the entire package consisting of the central software managing a computer’s resources and all of the accompanying standard software tools, such as command-line interpreters, graphical user interfaces, file utilities, and editors. z

2)More narrowly, to refer to the central software that manages and allocates computer resources (i.e., the CPU, RAM, and devices).

The term kernel is often used as a synonym for the second meaning, and it is with this meaning of the term operating system that we are concerned in this book. Although it is possible to run programs on a computer without a kernel, the presence of a kernel greatly simplifies the writing and use of other programs, and increases the power and flexibility available to programmers. The kernel does this by providing a software layer to manage the limited resources of a computer.

The Linux kernel executable typically resides at the pathname /boot/vmlinuz, or something similar. The derivation of this filename is historical. On early UNIX implementations, the kernel was called unix. Later UNIX implementations, which implemented virtual memory, renamed the kernel as vmunix. On Linux, the filename mirrors the system name, with the z replacing the final x to signify that the kernel is a compressed executable.

Tasks performed by the kernel Among other things, the kernel performs the following tasks: z Process scheduling: A computer has one or more central processing units (CPUs), which execute the instructions of programs. Like other UNIX systems, Linux is a preemptive multitasking operating system, Multitasking means that multiple processes (i.e., running programs) can simultaneously reside in memory and each may receive use of the CPU(s). Preemptive means that the rules governing which processes receive use of the CPU and for how long are determined by the kernel process scheduler (rather than by the processes themselves). z Memory management: While computer memories are enormous by the standards of a decade or two ago, the size of software has also correspondingly grown, so that physical memory (RAM) remains a limited resource that the kernel must share among processes in an equitable and efficient fashion. Like most modern operating systems, Linux employs virtual memory management (Section 6.4), a technique that confers two main advantages: – Processes are isolated from one another and from the kernel, so that one process can’t read or modify the memory of another process or the kernel. – Only part of a process needs to be kept in memory, thereby lowering the memory requirements of each process and allowing more processes to be held in RAM simultaneously. This leads to better CPU utilization, since it increases the likelihood that, at any moment in time, there is at least one process that the CPU(s) can execute. z Provision of a file system: The kernel provides a file system on disk, allowing files to be created, retrieved, updated, deleted, and so on. z Creation and termination of processes: The kernel can load a new program into memory, providing it with the resources (e.g., CPU, memory, and access to files) that it needs in order to run. Such an instance of a running program is termed a process. Once a process has completed execution, the kernel ensures that the resources it uses are freed for subsequent reuse by later programs. z Access to devices: The devices (mice, monitors, keyboards, disk and tape drives, and so on) attached to a computer allow communication of information between the computer and the outside world, permitting input, output, or both. The kernel provides programs with an interface that standardizes and simplifies access to devices, while at the same time arbitrating access by multiple processes to each device. Fundamental Concepts 23 z Networking: The kernel transmits and receives network messages (packets) on behalf of user processes. This task includes routing of network packets to the target system. z Provision of a system call application programming interface (API): Processes can request the kernel to perform various tasks using kernel entry points known as system calls. The Linux system call API is the primary topic of this book. Section 3.1 details the steps that occur when a process performs a system call. In addition to the above features, multiuser operating systems such as Linux generally provide users with the abstraction of a virtual private computer; that is, each user can log on to the system and operate largely independently of other users. For example, each user has their own disk storage space (home directory). In addition, users can run programs, each of which gets a share of the CPU and operates in its own virtual address space, and these programs can independently access devices and transfer information over the network. The kernel resolves potential conflicts in accessing hardware resources, so users and processes are generally unaware of the conflicts.

Process versus kernel views of the system

In many everyday programming tasks, we are accustomed to thinking about programming in a process-oriented way. However, when considering various topics covered later in this book, it can be useful to reorient our perspective to consider things from the kernel’s point of view. To make the contrast clear, we now consider how things look first from a process viewpoint and then from a kernel viewpoint. A running system typically has numerous processes. For a process, many things happen asynchronously. An executing process doesn’t know when it will next time out, which other processes will then be scheduled for the CPU (and in what order), or when it will next be scheduled. The delivery of signals and the occurrence of interprocess communication events are mediated by the kernel, and can occur at any time for a process. Many things happen transparently for a process. A process doesn’t know where it is located in RAM or, in general, whether a particular part of its memory space is currently resident in memory or held in the swap area (a reserved area of disk space used to supplement the computer’s RAM). Similarly, a process doesn’t know where on the disk drive the files it accesses are being held; it simply refers to the files by name. A process operates in isolation; it can’t directly communicate with another process. A process can’t itself create a new process or even end its own existence. Finally, a process can’t communicate directly with the input and output devices attached to the computer. By contrast, a running system has one kernel that knows and controls everything. The kernel facilitates the running of all processes on the system. The kernel decides which process will next obtain access to the CPU, when it will do so, and for how long. The kernel maintains data structures containing information about all running processes and updates these structures as processes are created, change state, and terminate. The kernel maintains all of the low-level data structures that enable the filenames used by programs to be translated into physical locations on the disk. The kernel also maintains data structures that map the virtual memory of each process into the physical memory of the computer and the swap area(s) on disk. All communication between processes is done via mechanisms provided by the kernel. In response to requests from processes, the kernel creates new processes and terminates existing processes. Lastly, the kernel (in particular, device drivers) performs all direct communication with input and output devices, transferring information to and from user processes as required. Later in this book we’ll say things such as “a process can create another process,” “a process can create a pipe,” “a process can write data to a file,” and “a process can terminate by calling exit().” Remember, however, that the kernel mediates all such actions, and these statements are just shorthand for “a process can request that the kernel create another process,” and so on.

The Shell

A shell is a special-purpose program designed to read commands typed by a user and execute appropriate programs in response to those commands. Such a program is sometimes known as a command interpreter. The term login shell is used to denote the process that is created to run a shell when the user first logs in.

Whereas on some operating systems the command interpreter is an integral part of the kernel, on UNIX systems, the shell is a user process. Many different shells exist, and different users (or, for that matter, a single user) on the same computer can simultaneously use different shells. A number of important shells have appeared over time:

Bourne shell (sh): This is the oldest of the widely used shells, and was written by Steve Bourne. It was the standard shell for Seventh Edition UNIX. The Bourne shell contains many of the features familiar in all shells: I/O redirection, pipelines, filename generation (globbing), variables, manipulation of environment variables, command substitution, background command execution, and functions. All later UNIX implementations include the Bourne shell in addition to any other shells they might provide.

C shell (csh): This shell was written by Bill Joy at the University of California at Berkeley. The name derives from the resemblance of many of the flow-control constructs of this shell to those of the C programming language. The C shell provided several useful interactive features unavailable in the Bourne shell, including command history, command-line editing, job control, and aliases. The C shell was not backward compatible with the Bourne shell. Although the standard interactive shell on BSD was the C shell, shell scripts (described in a moment) were usually written for the Bourne shell, so as to be portable across all UNIX implementations.

Korn shell (ksh): This shell was written as the successor to the Bourne shell by David Korn at AT&T Bell Laboratories. While maintaining backward compatibility with the Bourne shell, it also incorporated interactive features similar to those provided by the C shell.

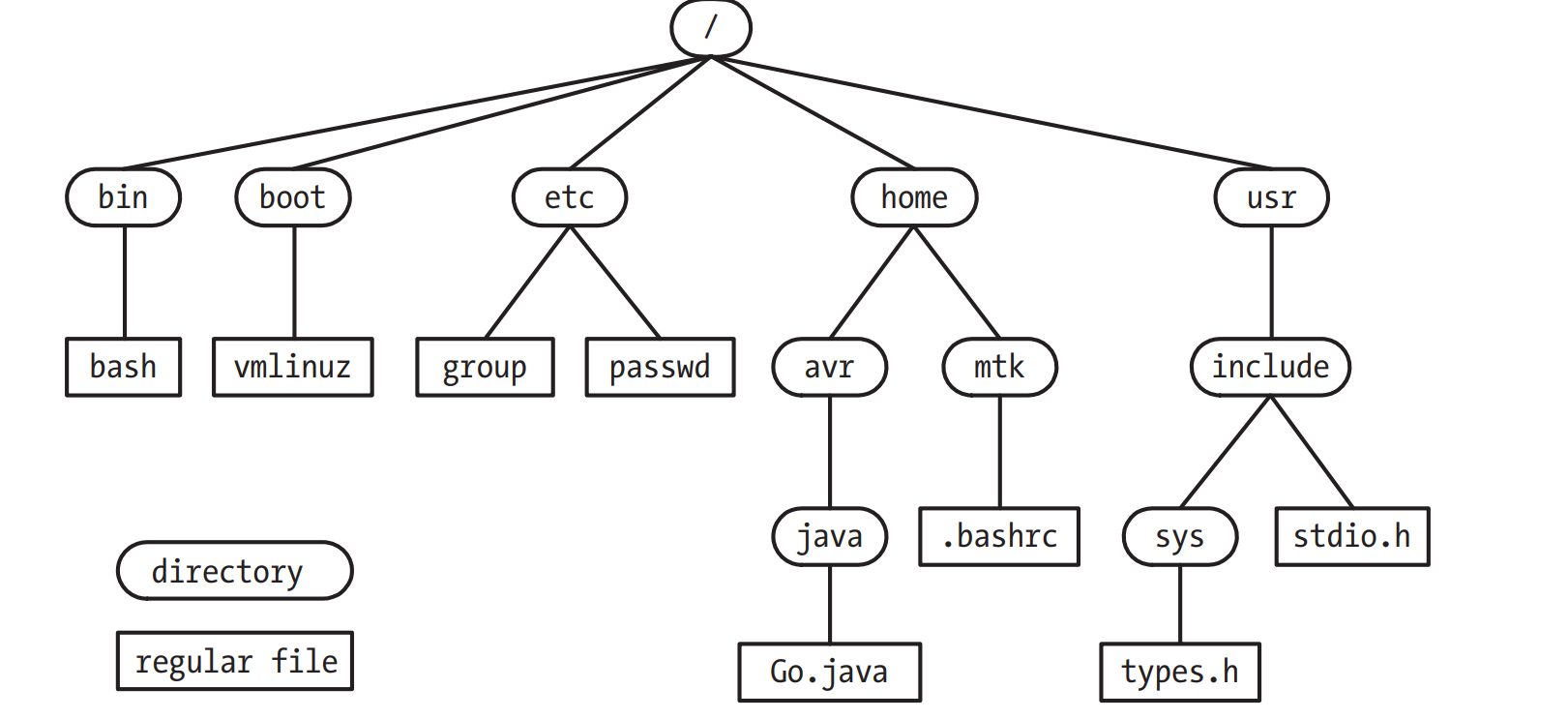
Bourne again shell (bash): This shell is the GNU project’s reimplementation of the Bourne shell. It supplies interactive features similar to those available in the C and Korn shells. The principal authors of bash are Brian Fox and Chet Ramey. Bash is probably the most widely used shell on Linux. (On Linux, the Bourne shell, sh, is actually provided by bash emulating sh as closely as possible.)

Users and Groups

Each user on the system is uniquely identified, and users may belong to groups. Users Every user of the system has a unique login name (username) and a corresponding numeric user ID (UID). For each user, these are defined by a line in the system password file, /etc/passwd, which includes the following additional information: z Group ID: the numeric group ID of the first of the groups of which the user is a member. z Home directory: the initial directory into which the user is placed after logging in. z Login shell: the name of the program to be executed to interpret user commands. The password record may also include the user’s password, in encrypted form. However, for security reasons, the password is often stored in the separate shadow password file, which is readable only by privileged users. Groups For administrative purposes—in particular, for controlling access to files and other system resources—it is useful to organize users into groups. For example, the people in a team working on a single project, and thus sharing a common set of files, might all be made members of the same group. In early UNIX implementations, a user could be a member of only one group. BSD allowed a user to simultaneously belong to multiple groups, an idea that was taken up by other UNIX implementations and the POSIX.1-1990 standard. Each group is identified by a single line in the system group file, /etc/group, which includes the following information: z Group name: the (unique) name of the group. z Group ID (GID): the numeric ID associated with this group. z User list: a comma-separated list of login names of users who are members of this group (and who are not otherwise identified as members of the group by virtue of the group ID field of their password file record). Superuser One user, known as the superuser, has special privileges within the system. The superuser account has user ID 0, and normally has the login name root. On typical UNIX systems, the superuser bypasses all permission checks in the system. Thus, for example, the superuser can access any file in the system, regardless of the permissions on that file, and can send signals to any user process in the system. The system administrator uses the superuser account to perform various administrative tasks on the system.

Single Directory Hierarchy, Directories, Links, and Files

The kernel maintains a single hierarchical directory structure to organize all files in the system. (This contrasts with operating systems such as Microsoft Windows, where each disk device has its own directory hierarchy.) At the base of this hierarchy is the root directory, named / (slash). All files and directories are children or further removed descendants of the root directory. Figure 2-1 shows an example of this hierarchical file structure.



File types

Within the file system, each file is marked with a type, indicating what kind of file it is. One of these file types denotes ordinary data files, which are usually called regular or plain files to distinguish them from other file types. These other file types include devices, pipes, sockets, directories, and symbolic links. The term file is commonly used to denote a file of any type, not just a regular file.

Directories and links

A directory is a special file whose contents take the form of a table of filenames coupled with references to the corresponding files. This filename-plus-reference association is called a link, and files may have multiple links, and thus multiple names, in the same or in different directories. Directories may contain links both to files and to other directories. The links between directories establish the directory hierarchy shown in Figure 2-1. Every directory contains at least two entries: . (dot), which is a link to the directory itself, and .. (dot-dot), which is a link to its parent directory, the directory above it in the hierarchy. Every directory, except the root directory, has a parent. For the root directory, the dot-dot entry is a link to the root directory itself (thus, /.. equates to /).

Filenames

On most Linux file systems, filenames can be up to 255 characters long. Filenames may contain any characters except slashes (/) and null characters (\0). However, it is advisable to employ only letters and digits, and the . (period), \_ (underscore), and - (hyphen) characters. This 65-character set, [-.\_a-zA-Z0-9], is referred to in SUSv3 as the portable filename character set. We should avoid the use of characters in filenames that are not in the portable filename character set because those characters may have special meanings within the shell, within regular expressions, or in other contexts. If a filename containing characters with special meanings appears in such contexts, then these characters must be escaped; that is, specially marked—typically with a preceding backslash (\)— to indicate that they should not be interpreted with those special meanings. In contexts where no escape mechanism is available, the filename is not usable. We should also avoid filenames beginning with a hyphen (-), since such filenames may be mistaken for options when specified in a shell command.’

Pathnames

A pathname is a string consisting of an optional initial slash (/) followed by a series of filenames separated by slashes. All but the last of these component filenames identifies a directory (or a symbolic link that resolves to a directory). The last component of a pathname may identify any type of file, including a directory. The series of component filenames preceding the final slash is sometimes referred to as the directory part of a pathname, while the name following the final slash is sometimes referred to as the file or base part of the pathname. A pathname is read from left to right; each filename resides in the directory specified by the preceding part of the pathname. The string .. can be used anywhere in a pathname to refer to the parent of the location so far specified in the pathname.

A pathname describes the location of a file within the single directory hierarchy, and is either absolute or relative:

An absolute pathname begins with a slash (/) and specifies the location of a file with respect to the root directory. Examples of absolute pathnames for files in Figure 2-1 are /home/mtk/.bashrc, /usr/include, and / (the pathname of the root directory).

A relative pathname specifies the location of a file relative to a process’s current working directory (see below), and is distinguished from an absolute pathname by the absence of an initial slash. In Figure 2-1, from the directory usr, the file types.h could be referenced using the relative pathname include/sys/types.h, while from the directory avr, the file .bashrc could be accessed using the relative pathname ../mtk/.bashrc.

Current working directory Each process has a current working directory (sometimes just referred to as the process’s working directory or current directory). This is the process’s “current location” within the single directory hierarchy, and it is from this directory that relative pathnames are interpreted for the process. A process inherits its current working directory from its parent process. A login shell has its initial current working directory set to the location named in the home directory field of the user’s password file entry. The shell’s current working directory can be changed with the cd command. File ownership and permissions Each file has an associated user ID and group ID that define the owner of the file and the group to which it belongs. The ownership of a file is used to determine the access rights available to users of the file. For the purpose of accessing a file, the system divides users into three categories: the owner of the file (sometimes termed the user of the file), users who are members of the group matching the file’s group ID ( group), and the rest of the world (other). Three permission bits may be set for each of these categories of user (making a total of nine permission bits): read permission allows the contents of the file to be read; write permission allows modification of the contents of the file; and execute permission allows execution of the file, which is either a program or a script to be processed by some interpreter (usually, but not always, one of the shells). These permissions may also be set on directories, although their meanings are slightly different: read permission allows the contents of (i.e., the filenames in) the directory to be listed; write permission allows the contents of the directory to be changed (i.e.,filenames can be added, removed, and changed); and execute (sometimes called search) permission allows access to files within the directory (subject to the permissions on the files themselves).

Processes

Put most simply, a process is an instance of an executing program. When a program is executed, the kernel loads the code of the program into virtual memory, allocates space for program variables, and sets up kernel bookkeeping data structures to record various information (such as process ID, termination status, user IDs, and group IDs) about the process. From a kernel point of view, processes are the entities among which the kernel must share the various resources of the computer. For resources that are limited, such as memory, the kernel initially allocates some amount of the resource to the process, and adjusts this allocation over the lifetime of the process in response to the demands of the process and the overall system demand for that resource. When the process terminates, all such resources are released for reuse by other processes. Other resources, such as the CPU and network bandwidth, are renewable, but must be shared equitably among all processes.

Process

memory layout A process is logically divided into the following parts, known as segments:

Text: the instructions of the program.

Data: the static variables used by the program.

Heap: an area from which programs can dynamically allocate extra memory.

Stack: a piece of memory that grows and shrinks as functions are called and return and that is used to allocate storage for local variables and function call linkage information.

Process creation and program execution

A process can create a new process using the fork() system call. The process that calls fork() is referred to as the parent process, and the new process is referred to as the child process. The kernel creates the child process by making a duplicate of the parent process. The child inherits copies of the parent’s data, stack, and heap segments, which it may then modify independently of the parent’s copies. (The program text, which is placed in memory marked as read-only, is shared by the two processes.) The child process goes on either to execute a different set of functions in the same code as the parent, or, frequently, to use the execve() system call to load and execute an entirely new program. An execve() call destroys the existing text, data, stack, and heap segments, replacing them with new segments based on the code of the new program. Several related C library functions are layered on top of execve(), each providing a slightly different interface to the same functionality. All of these functions have names starting with the string exec, and where the differences don’t matter, we’ll use the notation exec() to refer generally to these functions. Be aware, however, that there is no actual function with the name exec(). Commonly, we’ll use the verb to exec to describe the operation performed execve() and the library functions layered on top of it.

Process ID and parent process ID

Each process has a unique integer process identifier (PID). Each process also has a parent process identifier (PPID) attribute, which identifies the process that requested the kernel to create this process.

Process termination and termination status

A process can terminate in one of two ways: by requesting its own termination using the \_exit() system call (or the related exit() library function), or by being killed by the delivery of a signal. In either case, the process yields a termination status, a small nonnegative integer value that is available for inspection by the parent process using the wait() system call. In the case of a call to \_exit(), the process explicitly specifies its own termination status. If a process is killed by a signal, the termination status is set according to the type of signal that caused the death of the process. (Sometimes, we’ll refer to the argument passed to \_exit() as the exit status of the process, as distinct from the termination status, which is either the value passed to \_exit() or an indication of the signal that killed the process.) By convention, a termination status of 0 indicates that the process succeeded, and a nonzero status indicates that some error occurred. Most shells make the termination status of the last executed program available via a shell variable named $?.

Process user and group identifiers(credentials)

Each process has a number of associated user IDs (UIDs) and group IDs (GIDs). These include:

Real user ID and real group ID: These identify the user and group to which the process belongs. A new process inherits these IDs from its parent. A login shell gets its real user ID and real group ID from the corresponding fields in the system password file.

Effective user ID and effective group ID: These two IDs (in conjunction with the supplementary group IDs discussed in a moment) are used in determining the permissions that the process has when accessing protected resources such as files and interprocess communication objects. Typically, the process’s effective IDs have the same values as the corresponding real IDs. Changing the effective IDs is a mechanism that allows a process to assume the privileges of another user or group, as described in a moment.

Supplementary group IDs: These IDs identify additional groups to which a process belongs. A new process inherits its supplementary group IDs from its parent. A login shell gets its supplementary group IDs from the system group file.

Privileged processes Traditionally, on UNIX systems, a privileged process is one whose effective user ID is 0 (superuser). Such a process bypasses the permission restrictions normally applied by the kernel. By contrast, the term unprivileged (or nonprivileged) is applied to processes run by other users. Such processes have a nonzero effective user ID and must abide by the permission rules enforced by the kernel. A process may be privileged because it was created by another privileged process—for example, by a login shell started by root (superuser). Another way a process may become privileged is via the set-user-ID mechanism, which allows a process to assume an effective user ID that is the same as the user ID of the program file that it is executing.

The init process When booting the system, the kernel creates a special process called init, the “parent of all processes,” which is derived from the program file /sbin/init. All processes on the system are created (using fork()) either by init or by one of its descendants. The init process always has the process ID 1 and runs with superuser privileges. The init process can’t be killed (not even by the superuser), and it terminates only when the system is shut down. The main task of init is to create and monitor a range of processes required by a running system.

Daemon processes A daemon is a special-purpose process that is created and handled by the system in the same way as other processes, but which is distinguished by the following characteristics: z It is long-lived. A daemon process is often started at system boot and remains in existence until the system is shut down. z It runs in the background, and has no controlling terminal from which it can read input or to which it can write output. Examples of daemon processes include syslogd, which records messages in the system log, and httpd, which serves web pages via the Hypertext Transfer Protocol (HTTP).