

- The system is written in a high-level language, making it easy to read, understand, change, and move to other machines. Ritchie estimates that the first system in C was 20 to 40 percent larger and slower because it was not written in assembly language, but the advantages of using a higher-level language far outweigh the disadvantages (see page 1965 of [Ritchie 78b]).
- It has a simple user interface that has the power to provide the services that users want.
- It provides primitives that permit complex programs to be built from simpler programs.
- It uses a hierarchical file system that allows easy maintenance and efficient implementation.
- It uses a consistent format for files, the byte stream, making application programs easier to write.
- It provides a simple, consistent interface to peripheral devices.
- It is a multi-user, multiprocess system; each user can execute several processes simultaneously.
- It hides the machine architecture from the user, making it easier to write programs that run on different hardware implementations.

The philosophy of simplicity and consistency underscores the UNIX system and accounts for many of the reasons cited above.

Although the operating system and many of the command programs are written in C, UNIX systems support other languages, including Fortran, Basic, Pascal, Ada, Cobol, Lisp, and Prolog. The UNIX system can support any language that has a compiler or interpreter and a system interface that maps user requests for operating system services to the standard set of requests used on UNIX systems.

1.2 SYSTEM STRUCTURE

Figure 1.1 depicts the high-level architecture of the UNIX system. The hardware at the center of the diagram provides the operating system with basic services that will be described in Section 1.5. The operating system interacts directly³ with the hardware, providing common services to programs and insulating them from hardware idiosyncrasies. Viewing the system as a set of layers, the operating system is commonly called the *system kernel*, or just the kernel, emphasizing its

3. In some implementations of the UNIX system, the operating system interacts with a native operating system that, in turn, interacts with the underlying hardware and provides necessary services to the system. Such configurations allow installations to run other operating systems and their applications in parallel to the UNIX system. The classic example of such a configuration is the MERT system [Lycklama 78a]. More recent configurations include implementations for IBM System/370 computers [Felton 84] and for UNIVAC 1100 Series computers [Bodenstab 84].

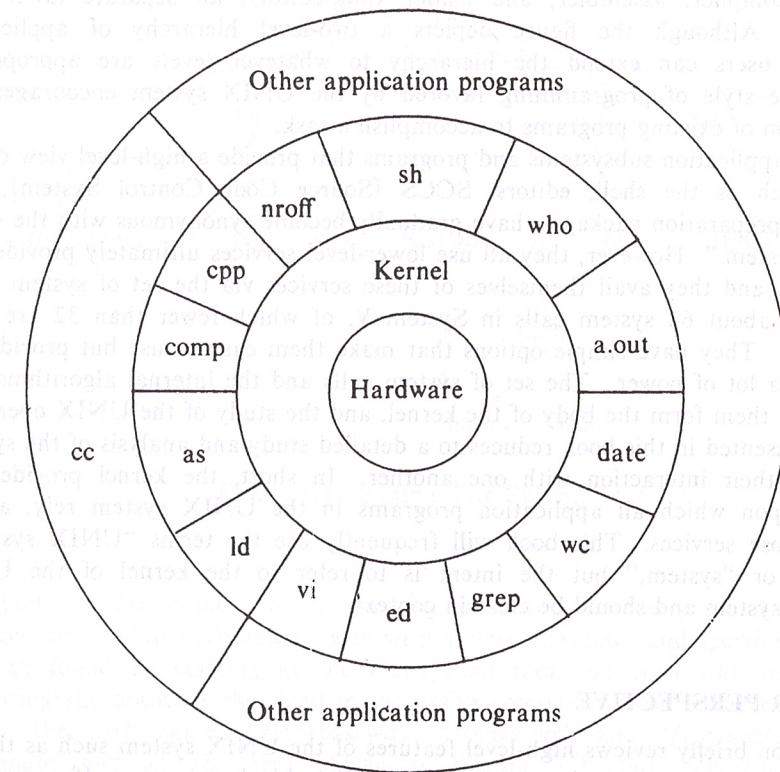


Figure 1.1. Architecture of UNIX Systems

isolation from user programs. Because programs are independent of the underlying hardware, it is easy to move them between UNIX systems running on different hardware if the programs do not make assumptions about the underlying hardware. For instance, programs that assume the size of a machine word are more difficult to move to other machines than programs that do not make this assumption.

Programs such as the shell and editors (*ed* and *vi*) shown in the outer layers interact with the kernel by invoking a well defined set of *system calls*. The system calls instruct the kernel to do various operations for the calling program and exchange data between the kernel and the program. Several programs shown in the figure are in standard system configurations and are known as *commands*, but private user programs may also exist in this layer as indicated by the program whose name is *a.out*, the standard name for executable files produced by the C compiler. Other application programs can build on top of lower-level programs, hence the existence of the outermost layer in the figure. For example, the standard C compiler, *cc*, is in the outermost layer of the figure: it invokes a C preprocessor,

two-pass compiler, assembler, and loader (link-editor), all separate lower-level programs. Although the figure depicts a two-level hierarchy of application programs, users can extend the hierarchy to whatever levels are appropriate. Indeed, the style of programming favored by the UNIX system encourages the combination of existing programs to accomplish a task.

Many application subsystems and programs that provide a high-level view of the system such as the shell, editors, SCCS (Source Code Control System), and document preparation packages, have gradually become synonymous with the name "UNIX system." However, they all use lower-level services ultimately provided by the kernel, and they avail themselves of these services via the set of system calls. There are about 64 system calls in System V, of which fewer than 32 are used frequently. They have simple options that make them easy to use but provide the user with a lot of power. The set of system calls and the internal algorithms that implement them form the body of the kernel, and the study of the UNIX operating system presented in this book reduces to a detailed study and analysis of the system calls and their interaction with one another. In short, the kernel provides the services upon which all application programs in the UNIX system rely, and it defines those services. This book will frequently use the terms "UNIX system," "kernel," or "system," but the intent is to refer to the kernel of the UNIX operating system and should be clear in context.

1.3 USER PERSPECTIVE

This section briefly reviews high-level features of the UNIX system such as the file system, the processing environment, and building block primitives (for example, *pipes*). Later chapters will explore kernel support of these features in detail.

1.3.1 The File System

The UNIX file system is characterized by

- a hierarchical structure,
- consistent treatment of file data,
- the ability to create and delete files,
- dynamic growth of files,
- the protection of file data.
- the treatment of peripheral devices (such as terminals and tape units) as files.

The file system is organized as a tree with a single root node called *root* (written `/`); every non-leaf node of the file system structure is a *directory* of files, and files at the leaf nodes of the tree are either directories, *regular files*, or *special device files*. The name of a file is given by a *path name* that describes how to locate the file in the file system hierarchy. A path name is a sequence of component names separated by slash characters; a component is a sequence of characters that

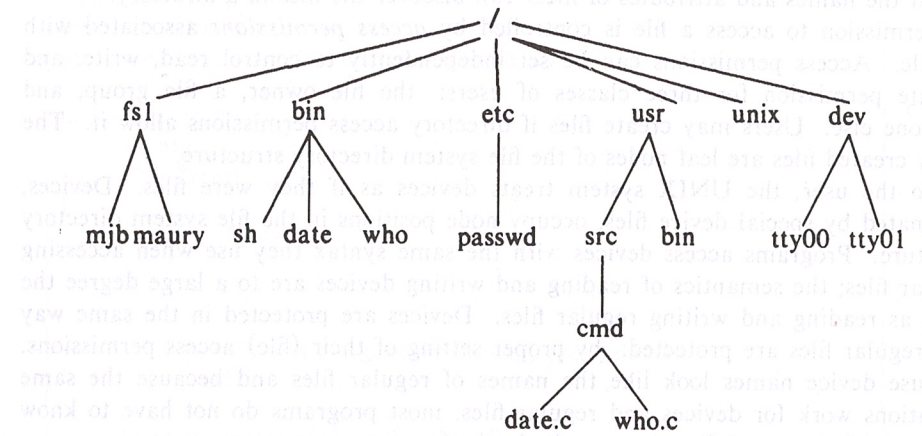


Figure 1.2. Sample File System Tree

designates a file name that is uniquely contained in the previous (directory) component. A full path name starts with a slash character and specifies a file that can be found by starting at the file system root and traversing the file tree, following the branches that lead to successive component names of the path name. Thus, the path names `/etc/passwd`, `/bin/who`, and `/usr/src/cmd/who.c` designate files in the tree shown in Figure 1.2, but `/bin/passwd` and `/usr/src/date.c` do not. A path name does not have to start from root but can be designated relative to the *current directory* of an executing process, by omitting the initial slash in the path name. Thus, starting from directory `/dev`, the path name `tty01` designates the file whose full path name is `/dev/tty01`.

Programs in the UNIX system have no knowledge of the internal format in which the kernel stores file data, treating the data as an unformatted stream of bytes. Programs may interpret the byte stream as they wish, but the interpretation has no bearing on how the operating system stores the data. Thus, the syntax of accessing the data in a file is defined by the system and is identical for all programs, but the semantics of the data are imposed by the program. For example, the text formatting program *troff* expects to find "new-line" characters at the end of each line of text, and the system accounting program *acctcom* expects to find fixed length records. Both programs use the same system services to access the data in the file as a byte stream, and internally, they parse the stream into a suitable format. If either program discovers that the format is incorrect, it is responsible for taking the appropriate action.

Directories are like regular files in this respect; the system treats the data in a directory as a byte stream, but the data contains the names of the files in the directory in a predictable format so that the operating system and programs such as