A SUN TECHNICAL REPORT

The UNIX System

Written and researched by Bill Courington
Book design and line illustrations by Neumeier Design Team
Electronic publishing by Adapt, Inc.
Printed by Malloy Lithographing, Inc.

Contents

Figures

Tables

Sun Technical Reports

The engineer or system architect evaluating a computer is often presented with two written descriptions of the product. These are the "brochure" and the "manual." Between these is a large gulf that makes it difficult to obtain a moderately detailed picture of a system, or a particular aspect of it. To fill this gap, Sun provides this first in a series of *Technical Reports*.

As implied in its name, a *Technical Report* introduces a subject to the technically competent engineer. What do we mean by "technically competent?" We mean that the reader is familiar with the general subject area, but not with the specifics of Sun's implementation. For example, the *UNIX*[1] *System Technical Report* describes how processes are created, how processes communicate, how they are scheduled, and so on, but it does not describe what a process *is*, nor what a multiprocess operating system is *good for*. The reader is expected to bring this level of knowledge from school or from experience with other operating systems. At the same time, some *Technical Reports* will cover topics that have not yet become part of the "general knowledge." Such a *Technical Report* provides commensurately more background information.

While *Technical Reports* are not aimed at the computer-naive, they nevertheless deliberately oversimplify their subjects and omit coverage of some features. If a subject appears to be missing, or is explained in insufficient detail for your needs, consult a Sun sales office for more information.

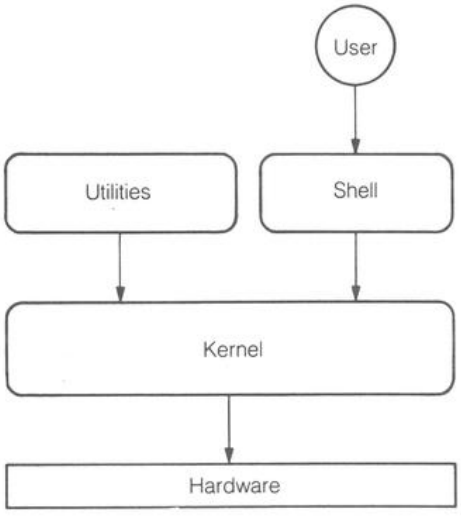
1. UNIX is a trademark of AT&T Bell Laboratories.

Introduction

This *Technical Report* describes the UNIX system — the operating system that is the heart of the Sun computing environment.[1] The material is organized in four chapters. Chapter 1 establishes a perspective that helps to explain why the UNIX system looks different from many other operating systems. It also explains how the several versions of the UNIX system came into being, and describes some of the differences between these versions.

The remaining three chapters reflect the structure of the UNIX system. It is convenient to think of the system as consisting of a **kernel**, a **shell**, and a large collection of **utilities** (see Figure 1); one chapter describes each of these.[2] The kernel manages the hardware (for example, processor cycles and memory) and supplies fundamental services (for example, filing) that the hardware does not provide. The shell is the interface between the kernel and system users; it is both a command interpreter and a programming language. Utilities are programs in the usual sense: compilers, editors, debuggers, and the like; however, the number of utilities (around 220), their scope (they include a compiler-compiler and typesetting software), and the ways in which they can be fit together is unusual indeed.

Fig. 1 *UNIX System Structure*



This "inside-to-outside" sequence differs from most descriptions of the UNIX system which begin with instructions for logging in, and discuss the kernel little if at all. However, readers of this report

1. The report describes Sun's Release 2.0.
2. Actually, the shell is just another utility, but is so important as to merit separate consideration.

are likely to be as interested in the services the UNIX system provides for programs as those it provides for users. Moreover, several important shell services are not the shell's own inventions but convenient notations that give users access to underlying kernel facilities. Exposing the kernel first provides a more accurate description of the system and shortens the total discussion as well. Such an approach does, however, rely on the reader's general familiarity with multiprocess operating systems.

The report also surveys two facilities that are Sun innovations and are not part of the UNIX system proper. These are the Network File System (covered in the Input/Output section of the kernel chapter) and the SunWindows[3] user interface, which is covered in its own chapter following the utilities. These facilities are fundamental to "what makes a Sun a Sun," are heavily used by Sun customers, and fit so well with the UNIX system that for practical purposes they are not considered as separate. While it is possible to use a Sun workstation as standalone "UNIX system box," with no Network File System and no windows, no one would willingly do so.

For these reasons, the topics are covered in this report. However, the subjects are large and rich and space does not permit their treatment in the same depth as the UNIX system. Therefore, the NFS and SunWindows descriptions should be considered as introductions only.

3. SunWindows is a trademark of Sun Microsystems, Inc.

1 *Perspective*

The UNIX operating system is a multiprocess (or multitask) operating system with facilities functionally similar to other minicomputer and mainframe operating systems. But the UNIX system is also a *phenomenon*. Designed by a small group of people for their own use, it has, with almost no conventional marketing effort, become perhaps the most widely used operating system for mid-range computers.

The UNIX system has been propelled to prominence not by a corporate campaign, but by the grassroots enthusiasm of its users. These users have diverse impressions and opinions of the system; as a result, the public image of the UNIX system is somewhat fuzzy. This chapter tries to clarify the picture by illuminating some key issues: why the UNIX system is so popular; why it is fairly criticized; how Sun has changed it; and why there are multiple versions of the UNIX system.

1.1 Sources of Strength

The UNIX system's extraordinary popularity can be credited to three of its fundamental properties:

1. The UNIX system is a *portable* operating system. Hardware vendors are attracted to it because they can implement it on a new computer with relatively little effort; in so doing, they inherit a large community of users. With implementations available on dozens of machines, both software vendors and users become computer-independent when they adopt the UNIX system. A high-level language program can make direct use of operating system facilities and yet be transported to a different machine by simply recompiling. Not only do programs move easily from one UNIX machine to another, but so do the tools and expertise of people. Personnel trained on one UNIX implementation are productive almost immediately on another.

2. The UNIX system provides an exceptionally *productive computing environment*, particularly for programmers. Included in the UNIX system is a rich set of software development tools, many of which apply to general text manipulation (for example, documentation) as well. Moreover, these tools can be used in combination to solve a remarkable number of problems without conventional programming. In sum, the UNIX system probably sup-

ports the ideal of reusable software more effectively than any other commercial operating system.

3. Finally, the UNIX system is an *elegant design* that is intellectually appealing and satisfying to use — many experienced users would say fun to use. It is difficult to capture the essence of the system's elegance, but it has to do with power derived from simplicity, rather than complexity. Instead of the "laundry list" of semi-independent (and sometimes conflicting) features that characterize many operating systems, the UNIX system provides a relatively small number of fundamental, but universal, facilities. It obtains generality by eliminating restrictions instead of adding options. In the words of one of its designers, it is a "simple, coherent system that pushes a few good ideas and models to the limit."[1] At the same time, users can connect these simple mechanisms to build powerful new tools that meet their own needs or reflect their own preferences.

It is easy to understand and appreciate the value of a portable operating system. The UNIX system's qualities of elegance and improved productivity are less tangible. This report tries to demonstrate them, but they can only be deeply understood by using the system for a few months. Nevertheless, it is these qualities that fuel the enthusiasm of UNIX system users; it is very hard to find a user who both deeply understands the UNIX system and would voluntarily choose another commercial operating system. The system's strengths greatly overbalance the shortcomings described in the next section.

1.2 Areas of Weakness

Critics tend to focus their complaints on the interface the UNIX system presents to novice users.[2] The command interface is terse and inconsistent, error reports are sketchy, and the system has no "help" facility. Some commands have strange names like **grep** and **awk**. The system presumes users know what they are doing; for example, its default action is to overwrite an existing file when a new file with the same name is created.

Why are these widely perceived shortcomings present in such a popular system? They are the consequence of the environment in which the UNIX system was designed and first used. This environment was a Bell Laboratories computer science research group in the early 1970s. The members of this community were computing experts, and they were building a system for their own

1. Dennis M. Ritchie, "Reflections on Software Research," *Communications of the ACM*, Volume 27, Number 8, August 1984

2. It is only fair to point out, however, that many expert users are proponents of the system's user interface, especially its terseness and its implied respect for their competence.

use — they had no pretensions of building a commercial operating system. Their hardware was a small minicomputer equipped with slow teletypewriter terminals. They designed the system so operating system utilities, which comprise much of the user interface, could be added by users, and many were. It is not surprising that this environment spawned a simple timesharing system, with a terse and inconsistent user interface (remember those slow terminals and their expert users).

The UNIX system initially spread to similar environments, namely research laboratories and university computer science departments. The user interface presented no particular difficulties to these communities, particularly when balanced against the system's strengths. Later, when the system was adopted by professional programmers in industry, the user interface again was not a major issue. Now, however, the UNIX system is used in many different environments. System users include the computer-naive as well as the computer scientist. As a result, the difficulty of the user interface for new users is appropriately receiving more attention. There may be another reason that the user interface has been slow to change: the hardware and conceptual ingredients for radically improving the user interface, such as dedicated processors, bit-mapped displays, pointing devices, and the notions of windows and menus, have only recently become widely available and affordable.

Other UNIX system weaknesses have been cited as well: small process address spaces, a slow file system, and overly constrained interprocess communication are among them. These kinds of defects have somewhat inhibited the system's acceptance as a general-purpose operating system. As will be shown in this report, some UNIX versions, including Sun's, have eliminated most of these problems, thereby widening the scope of problems to which the system can be effectively applied.

1.3 Guiding Evolution

UNIX system users have changed since the system was designed, as have the problems to which the UNIX system has been applied. But the hardware on which it runs has changed even more dramatically. Microcomputers with multimegabyte address spaces are common, and fast display terminals have completely replaced teletypewriters. Bit-mapped graphics displays and pointing devices, such as mice, are emerging as standard equipment. Increasingly, timesharing is giving way to networks of dedicated workstations.

To keep pace with changing patterns of use and developing hardware, the UNIX system has changed over the years, as it must continue to change. At the same time, to keep one of its most important strengths — portability — intact, it has also stayed the same and must continue to stay the same. These seemingly

paradoxical requirements can be reconciled by carefully distinguishing between *interfaces* and *implementations*.

Two kinds of interface are of interest, *program interfaces* and *user interfaces*. Sun's general approach is to maintain existing interfaces to preserve portability, and to add new interfaces to provide access to new capabilities. As an example of a such a new interface, consider the SunWindows system described later in this report. SunWindows supports the construction of applications that present interfaces based on the recent technologies of overlapping display windows, pop-up menus, and the mouse pointing device. Such an application is called a tool. A tool can be designed to assume its user knows nothing of the UNIX system, perhaps not even how to type. Thus SunWindows allows the development of alternate system interfaces. Although most tools are developed by Sun customers, several general-purpose ones are supplied with the SunWindows system. One of these is a terminal emulator that makes a window behave like an ordinary display terminal, thereby preserving the conventional UNIX system interface. Thus, the SunWindows system embraces new technologies, permits construction of natural interfaces for varieties of different users, and simultaneously maintains compatibility with the UNIX system.

Besides adding new interfaces, Sun has improved the implementation of several facilities, and will continue to do so. Where an interface defines *what* something does, an implementation embodies *how* something is done. Implementations are functionally invisible and therefore can be changed without adversely affecting programs or users. For example, the Sun file system is considerably faster than many UNIX file system implementations, and Sun's virtual memory management accommodates programs that are too large for many UNIX versions. Programs written for other UNIX versions work properly on Suns because Sun has not altered the file system and memory management interfaces, only their implementations.

1.4 Multiple Versions

Speaking of versions, as Figure 1.1 shows, there have been, and continue to be, several versions of the UNIX system. These are largely but not completely compatible. In general, two forces have driven the evolution of the UNIX system, new hardware capabilities (for example, virtual memory microprocessors and local area networks) and the quest for wider applicability (for example, from running on timeshared minicomputers to dedicated workstations). As the system matures and gains the commitment of more computer vendors, its shape should stabilize. For the present, compatibility is a larger problem than it should be. (On the other hand, it is hard to think of any commercial operating system that is available for processors of different architecture. The UNIX system is commercially available on at least a dozen of them.) The

following historical overview traces the ancestors of the currently extant versions and gives some reasons for their development.

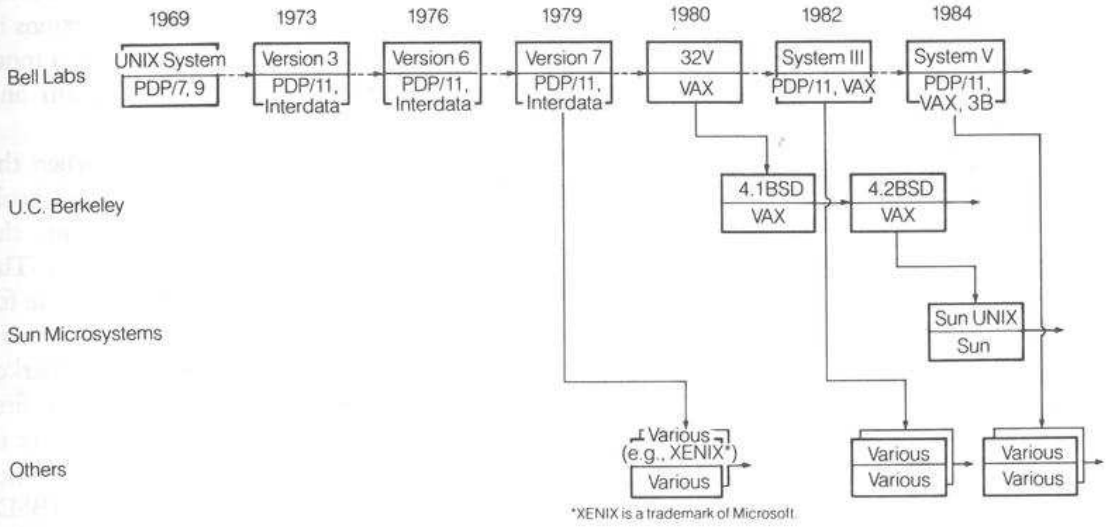


Fig. 1.1 *UNIX System Genealogy*

The name "UNIX" is not an acronym but a play on the phrase "castrated MULTICS," MULTICS being an operating system that influenced (both positively and negatively) the UNIX system's designers. In 1969 there was no simple powerful minicomputer operating system, so Ken Thompson at Bell Laboratories set out to build one for his own use and the use of his programming research group. He was soon joined by Dennis Ritchie.

Following the initial assembly language implementation for the PDP-7[3] and two assembly language versions for the PDP/11, a major milestone was reached in 1973. The system was translated from PDP/11 assembly language to the mid-level language C. (C was developed by Ritchie and was based on Thompson's language B, itself a descendent of Martin Richards's BCPL.) Recoding the system in C reduced its machine-dependent components to a few percent of the total lines of code, and made the job of transporting the UNIX system to a new machine a matter of a few months work. (It also increased the size of the system by about one-third.) Soon a version was running in Bell Laboratories on an Interdata 8/32, a machine whose architecture was much like an IBM/370 and very little like a PDP/11. The C implementation was described in the July 1974 issue of *Communications of the ACM*; this was the UNIX system's first exposure to a large audience outside Bell Labs. The essence of the system has not changed since then.

By 1974 Bell Labs was licensing Version 5, and shortly thereafter Version 6, to universities for a nominal fee.[4] The system

3. PDP is a trademark of Digital Equipment Corporation.
4. The early versions were also known by the editions of the manuals that described them, for example, the 6th and 7th Editions.

was unsupported, but *the source code was included in the distribution*. As a result, over the next few years the UNIX system was both used and studied by many students. Following graduation these enthusiastic students "marketed" the system in the commercial world. Version 7 is the common ancestor of virtually all UNIX versions in existence today. Compared to its predecessor, Version 7 was more easily transportable, and incorporated a better file system and additions to the C language.

The next major historical event occurred in 1980 when the U. S. Department of Defense chartered the University of California at Berkeley to redesign the UNIX system. Up to this time, the UNIX system had been a pure timesharing operating system. The Berkeley group was to transform it into an appropriate vehicle for research in distributed computing. The result is commonly known as 4BSD, for Fourth Berkeley Software Distribution (earlier Berkeley efforts for the PDP/11 are known as 2BSD). The first development in this project was called 4.1BSD; its roots were in System 32V, a UNIX version that ran on the VAX[5] , but did not exploit the virtual memory facilities of that machine. 4.1BSD, which amounted to a major redesign of System 32V, included:

- A greatly expanded process address space;
- Demand-paged virtual memory;
- A faster, more robust file system;
- Generalized interprocess communication, including basic local network support.

Berkeley also added utility programs such as a full-screen editor, a generic interface to smart terminals, and an alternate command interpreter. This second group of facilities, which many UNIX versions have adopted, is often described as the "Berkeley enhancements," but it is the first group that constitutes the really outstanding Berkeley achievement.

In 1982 AT&T offered their first commercial version of the UNIX system, called System III. System III added a number of new capabilities, including remote job entry, a Source Code Control System for managing large evolving software products, and accounting routines, but maintained the memory management limitations of its predecessors. System III's successor, System V, was largely compatible with System III and included a faster file system, an improved terminal driver, generalized interprocess communication (quite different from Berkeley's), shared memory, and semaphores.

Sun adopted the 4.2BSD design as the *starting point* for its operating system. While in many ways suitable for a network of technical workstations, 4.2BSD is still at its heart a minicomputer timesharing system. Accordingly, after transporting the system to its own hardware, Sun began a continuing program of adaptation

5. VAX is a trademark of Digital Equipment Corporation.

and extension. For example, support for graphics and a window/mouse interface were added and workstations without local disks gained the ability to transparently use the network for paging, file storage, and file sharing. Remnants of the timesharing heritage remain (for example the process scheduler and virtual memory manager), but can also be re-implemented to better fit the Sun workstation environment.

1.5 Conclusion

The UNIX system is unquestionably one of the great milestones in the history of computing. Its strengths are substantial and undeniable. It has no fatal flaws.[6] Its shortcomings, which are largely historical artifacts, are disappearing. Indeed, the system has proven remarkably adaptable; it has incorporated change more gracefully, than, for example, FORTRAN, with which it is sometimes compared.

In fact, the comparison with the FORTRAN of the 1950s and 1960s is fairly apt. FORTRAN was the first "programming environment" that transcended machine and manufacturer boundaries. The language itself was well-suited to a large class of computing problems, and was a great advance over assembly language. FORTRAN's greater significance, though, was its replacement of *dozens* of assembly languages, in other words, its nearly universal availability. FORTRAN demonstrated the power of a standard (or largely standard) language for programs and programmers.

The UNIX system, of course, is not a language but an operating system. Keeping in mind that it is considerably more powerful, flexible, and fun to work with than FORTRAN, the similarities between the two are nevertheless clear. Not without its faults, the UNIX system is a very good operating system. It is applicable to a wide spectrum of computing situations. It provides a largely standard environment for both programs and programmers that spans a wide range of machines. Finally, the shared pool of knowledge and software built upon this standard provides tremendous leverage for applying computers to business and technical problems.

6. The UNIX system does not, however, pretend to be suitable for applications with severe real-time requirements. At the same time, machines running true real-time operating systems can productively coexist and cooperate with Sun workstations on a network.

2 *The Kernel*

Strictly speaking, the kernel *is* the UNIX operating system; the shell and the utilities are ordinary programs with no special privileges. The kernel is a virtual machine that hides the characteristics of the underlying hardware, providing a collection of services that is independent of any particular computer. These services fall into four broad functional groups: processes, memory management, I/O, and timers. Before describing these services, it is necessary to see how they are invoked.

2.1 System Calls

The kernel performs some functions autonomously (for example, allocating CPU cycles and real memory) and others in response to requests for service from processes. These requests take the form of **system calls**. To the programmer, a system call is indistinguishable from an ordinary function or procedure call.

System calls are most often made from C programs (the C language is described later in this report), but they can be made from programs written in other languages as well. For the benefit of those readers not familiar with C, examples in this report are written in a pseudocode that resembles languages in the Pascal family. All system calls take the form of functions. A function always returns a value and for system calls, the value -1 indicates unsuccessful execution. When a call returns -1, an external variable called **errno** describes the nature of the problem.

2.2 Processes

In this report we consider a **program** to be a machine-interpretable *text* of instructions and data; it is analogous to a musical score. We define a **process** as one *execution* of a program, analogous to a performance. (Some operating systems give the term "task" to this notion of a unit of execution). Multiple concurrently executing processes are the norm in a UNIX system, and multiple processes frequently execute the same program (for example, a compiler or a shell) simultaneously. Each process, however, runs in its own address space, protected from all other processes.

2.2.1 Privilege

The UNIX system provides a number of system calls that, while necessary for system administration, can be disruptive or damaging if improperly used. Only **privileged** processes can execute such calls. Fundamental control over which processes are privileged and

which are not is vested in the system's **login** facility. In the UNIX system all[1] processes are ultimately created by users who log into the system by typing in a **login-name** and a **password**. A predefined login-name called **root** denotes the system's "superuser." The superuser is the person responsible for administering the system. For the most part, processes created by the superuser are privileged and all others are unprivileged; therefore the ability to create a privileged process is restricted to users who know the **root** password. However, the UNIX system also has a more fine-grained privilege-granting mechanism called **set-uid**, which allows non-privileged users to run trusted programs as privileged processes. This facility, which in fact is patented, is covered later in the chapter.

2.2.2 Creation and Termination

Compared to many operating systems, a typical UNIX system exhibits a rather high rate of process creation and termination. Some of this activity is due to the design of the shell, which customarily creates a process to execute a command (in the UNIX system, most commands are just ordinary utility programs — they generally are not embedded in the operating system). User-written programs often employ the same technique. For example, a process might want to count the characters in a file. A utility called **wc** does just that. Instead of executing its own code, the process can create a child process to execute **wc**. Thus, in the UNIX system, processes frequently use processes like programs use subroutines — and for the same reason: to avoid reinventing code that already exists. Of course, processes also create processes to run portions of algorithms in parallel — the conventional reason for creating a process in most operating systems — but the use of a process as something similar to a subroutine is largely unique to the UNIX system and the programming style that has come to be associated with it.

When one process creates another to do some work for it, the new process often has something in common with the old process. This may range from complete commonality (the old process is creating another instance of itself), to sharing files but running different programs (as in the character counting example above), to complete independence. It is difficult for a single "create process" call (such as typical multiprocess operating systems provide) to handle all of the possible situations well. The UNIX system provides several simple calls that can be issued in combination to achieve the desired effect. The calls are designed to take advantage of situations in which the new and old processes share some resources.

In the UNIX system a process is represented by three memory segments, called the **text** (or code), **data**, and **stack** segments, and by a set of data structures collectively known as the **process**

1. The exceptions are a few system processes created by the kernel. The most common system processes are various **daemons** that wake up periodically to perform housekeeping operations such as delivering electronic mail.

environment. A text segment contains code and constant data, a data segment contains variables, and a stack segment holds a process's stack. The process environment records the information the kernel needs to manage the process, such as register contents, priority, open files, and so on; for protection, a process cannot address its environment, but can alter it with system calls.

A new process is created with the **fork** system call. This call takes no parameters, and needs none, because the newly-created process (called the **child**) is effectively a duplicate of the process that calls **fork** (known as the **parent**).[2] **fork** copies the parent's data and stack segments to the child, and copies the parent's environment to the child as well. The child shares its parent's text segment to conserve memory space (UNIX programs are by default reentrant). Passing the environment down through the generations facilitates sharing, and minimizes the amount of work children have to do before they can run — rather than having to create a new environment, they only have to change that part of their inheritance that is inappropriate.

fork is unusual in that it *returns to both the parent and the child* — the child is executing the same program as the parent — but it returns a different value to each. The parent receives the **process ID** of the child (which is never 0) and the child receives 0. In this way the two executions of the parent's program can tell which is which and take different branches in the code. This is simpler than it sounds; Figure 2.1 shows an example.

Fig. 2.1 *Forking a Child Process*

```
(* parent calls fork *)
process_id := fork();

(* parent and child test process_id *)
if process_id = 0

then ...
        (* child takes this path *)
else ...
        (* parent takes this path *)
endif;
```

After checking the value returned by **fork**, the parent and child are executing different branches of the same program in parallel. In other words, the parent has just spawned an instance of itself. Often the child, while preserving much of its inherited environment, will make a few changes before proceeding with its real work.

If the child needs to execute a different program, as it often does, it issues an **exec** call. **exec** replaces the text and data segments of its caller with those of a new program read from a file specified in the call. **exec** does not alter its caller's environment; a child process may be executing a different program, but it still has access to its parent's files (possibly modified by the child between

2. **fork** operates in a manner somewhat reminiscent of biological cell division.

the fork and the **exec**). After issuing an **exec** call, the next instruction the child executes is the first instruction in the new program. Thus, a **fork** followed by an **exec** is equivalent to a traditional monolithic "create process" call. Note also that a multiphase sequential program whose phases communicate through files, such as a compiler, can use **exec** to advantage without **fork**; each phase can end with an **exec** that invokes the next phase.

When a child is ready to terminate, it issues an **exit** system call; this call takes a parameter whose value is returned to the child's parent. The preceding describes **normal** termination; a child may be terminated **abnormally** by any of several **signals** (discussed later in this chapter) initiated by the kernel, by a user, or by another process. When this happens, the parent is notified, also by a signal.

Back to the parent. The parent's environment is not changed by anything the child does, since the child is working with a copy of the parent's environment description. The parent is free to execute in parallel with the child. When, and if, the parent wants to wait for the child to terminate (parents often use children to simply run utilities in the manner of subroutines as mentioned at the beginning of this section) it issues the **wait** system call. **wait** returns the process id of the terminated child (a parent may be waiting for any of several children) and the status code passed by the child when it **exit**ed. Figure 2.2 shows how a parent can create a child and wait for it to terminate.

Fig. 2.2 *Waiting for a Child Process to Terminate*

```
(* parent calls fork *)
process_id := fork();

(* both parent and child
  are now executing this code *)
(* both test value returned by fork *)
if process_id = 0

then
        (* child *)

        (* ...optionally adjust environment... *)

        (* switch to new program *)
        exec(new_prog,...);

        (* new program issues exit
         call to terminate *)

else
        (* parent *)

        (* wait for termination *)
        process_id := wait(status);
endif;

(* child has terminated; parent is running again;
  it should check status to see if the child
  ran into problems. *)
```

Should the parent choose to run in parallel, it may wish to be notified when a child terminates normally or abnormally. It can do so by catching the SIGCHLD signal; upon receipt of this signal, the parent can issue a **wait** to find out what happened to the child.

Besides the basic **fork** and **exec** calls described above, variants are available for specific situations. For example, **vfork** is an optimized **fork** (and a Berkeley enhancement) that runs faster when the parent knows that it does not want to run in parallel with the child. In a normal **fork**, the stack and data segments and the environment of the parent are copied for the child, and the text segment is shared. **vfork** copies nothing — the child uses the parent's stack, data, and environment. Eliminating the segment copying makes **vfork** fast but also makes the child responsible for terminating without having made any alterations that might confuse the parent when it resumes.

2.2.3 Scheduling

The Sun UNIX system schedules ready processes according to their **base priorities**, tempered by dynamically-computed **priority adjustments**. Base priority values range from -20 (high), to +20 (low); a new process inherits its parent's base priority. In the absence of any explicit action (as discussed shortly), the base priority will be 0 (that is, a medium value).

As it runs, a process is charged for each 200 ms. of CPU time that it consumes (a hardware clock interrupts the processor every 200 ms.). Once per second the kernel recomputes the priority adjustments of all ready processes. It adds a process's adjustment to its base priority giving a **current priority**. After computing all current priorities, the kernel dispatches the process with the highest current priority. The priority adjustment gives preference to those processes that have consumed the least CPU time in the recent past. CPU consumption in "the recent past" is determined by an exponential decay of CPU usage — 90% of past CPU usage is forgotten in $5*n$ seconds where n is the average over the last minute of the number of runnable processes. Thus, the system penalizes CPU-intensive processes more heavily during periods of heavy loading and is more forgiving of them when lightly loaded. The scheduling technique has three results:

1. I/O-bound processes, which include most interactive processes, run very quickly after finishing an I/O operation (for example, after receiving a command from a terminal). The reason is that in the recent past such processes were waiting for I/O and thus consuming no CPU time.
2. CPU-bound processes do not starve, but run in bursts of up to one second, because as time passes, the scheduler forgets a process's CPU consumption. The greater the system load the less often a CPU-bound process will receive a turn on the CPU, minimizing its impact on interactive response.

3. Changes in process behavior (for example, a process that starts out I/O-bound as it reads data from a file, then becomes CPU-bound as it processes the data) are compensated for automatically.

A process may lower its base priority with the **setpriority** call; however, only a privileged process may raise its base priority, or change the priority of an unrelated process.

2.2.4 Signals

A UNIX **signal** is a software mechanism that is closely analogous to a hardware interrupt; it is a means of asynchronously notifying a process that an event has occurred. Unlike interrupts, signals all have the same priority; simultaneously occurring signals are delivered to a process serially, but in no defined sequence. Signals should *not* be used as a primitive interprocess communication mechanism. Pipes and sockets (described later in this chapter) are provided for this purpose.

A signal may ultimately originate with:

- The hardware; the kernel transforms hardware conditions such as addressing violations and arithmetic exceptions into signals;
- The kernel (for example, a process has asked to be notified when a device is ready for I/O);
- Another process (for example, a parent terminates a child that has run too long);
- A user (for example, typing Control-C to "interrupt" a running process).

One process may send a signal to another process with the **kill**[3] system call; it may send a signal to a group of processes (typically, those started directly, or indirectly, by one user) with the **killpg** system call. Only a privileged process can send a signal to a process outside its process group.

A process should expect to receive signals; it may deal with a signal in one of three ways:

1. It may accept the system's **default** action; often the default action is to terminate the process and notify its parent. For some signals (for example, addressing violations) the default action includes producing a file containing the memory image of the process for post-mortem debugging.
2. It may **ignore** the signal; the kernel simply discards a signal that a process is ignoring. Some "strong" signals, such as **SIGKILL**, which abnormally terminates a process, cannot be ignored.

3. The term "kill" is a misnomer — its sense is really "notify." The confusion may be historical — perhaps **kill** originally *did* kill a process and was later generalized to deliver any signal.

3. It may **catch** the signal. In this case the process designates a function (called a signal handler, analogous to an interrupt handler), which is automatically invoked when the signal is received. When the handler returns, execution resumes at the statement that was about to be executed when the signal was received. Often, a signal handler just performs some cleanup operations (for example, removing a temporary file) and then terminates the process gracefully. Rather sophisticated signal handling is also possible, however.

A process's signal disposition (that is, how the process wants the kernel to treat each type of signal that it could receive) is recorded in its environment. Since a child inherits its environment, in the absence of any action by the child, the kernel will deliver signals to it in the same way it would to its parent. If the child wants to do something different with signals, it issues the **sigvec**[4] system call early in its execution. This call takes parameters specifying how to treat each signal and the addresses of the handlers of signals that are to be caught.

If a process is catching signals, there may be critical sections of its code that must be executed without the possiblility of a signal being delivered. Consider an editor that is executing the following sequence of operations:

1. Rename current file to backup file.
2. Rename working file to current file.
3. Delete backup file.

If the process were to receive **SIGINT** (a "program interrupt" signal which a user may initiate by typing Control-C) during execution of this sequence, the user's files would be left in an inconsistent state. The Berkeley **sigblock** and **sigsetmask** calls may be used to temporarily delay incoming signals to protect such a critical region.

2.2.5 Pipes

A pipe is a conduit that enables a one-way byte stream to flow between two processes.[5] A process creates a pipe with the **pipe** system call. This call returns two **descriptors**; one of these represents the "write end" of the pipe (that is, the end into which bytes are written) and the other represents the pipe's "read end." (Descriptors are central to the UNIX model of I/O and are discussed later in this chapter.) A process can write a string of bytes into a pipe with the **write** system call. **write** takes three parameters:

1. the descriptor that represents the pipe's write end;

4. The conventional UNIX mechanism for dealing with signals is the **signal** system call. **sigvec** is a Berkeley addition that is more versatile but more complex to use. To preserve compatibility **signal** is provided as a library routine; it simply calls **sigvec**.

5. Since processes occupy separate address spaces, an address (pointer) sent through a pipe will not be meaningful to the receiving process. (Just as an address written to a file by one process is not meaningful if read by another process.)

2. the data area containing the bytes;
3. the number of bytes to write.

write returns the number of bytes actually written (which, in the case of a pipe, is always the number requested) or -1 in the case of an error. The writing process terminates the byte stream by issuing a **close** system call. A process wishing to read from a pipe issues a **read** system call. This call is essentially the mirror image of **write**, taking the same three parameters. **read** returns 0 (bytes transferred) when no more data remains in the pipe (that is, the writer has closed the other end of the pipe).

The kernel serializes simultaneous operations by the two processes on the pipe and buffers bytes written but not yet read. **read** and **write** block if the state of the pipe prevents immediate completion of the call (that is, writing to a full pipe or reading from an empty one). A blocked **write** call completes as soon as enough bytes are removed from the pipe; a blocked **read** completes when bytes are added to the pipe. An unlimited number of bytes can be transferred through a pipe.

Unlike files, pipes do not have system-wide names; to read or write a pipe, a process must have a descriptor for it. Descriptors can only be passed within a process or to a process's children (via environment inheritance). Therefore, to use a pipe a process must have created it or have inherited it from an ancestor that created it. In practice, a parent generally creates a pipe and then **fork**s the children who use it.

Thus, unrelated processes cannot communicate through pipes. Although pipes are very useful, this is an important limitation. Consider, for example, a general server process such as a print spooler. *Any* process should be able to send a file to the spooler for printing. To permit communication between unrelated processes, System III introduced named pipes, System V added message queues, and Berkeley added sockets.

2.2.6 Sockets

To permit processes running on different machines to communicate over a local area network, 4.2BSD introduced the notion of **sockets**. Processes on the same machine can also communicate via sockets, and need not be related in any way.[6] Thus, intramachine socket communication is just a degenerate case of intermachine communication, which is the subject of the *Network Services Technical Report*. Note, however, that the same **read** and **write** system calls work on sockets as on pipes — and on files and on devices, as will be shown later in this chapter.

6. Indeed, in the Sun UNIX system, pipes are implemented as degenerate sockets.

2.3 Memory Management

Based on the limitations of the PDP-11 architecture, early UNIX implementations provided comparatively small process address spaces. Depending on the PDP-11 model, a process was limited to either 64 Kbytes total, or 64 Kbytes for code and 64 Kbytes for data and stack. Multiprogramming was provided by swapping whole processes to disk. The fundamental innovation of the VAX architecture was very large process address spaces supported by virtual memory; Berkeley redesigned the memory management portion of the UNIX kernel to exploit the VAX's new capabilities. Sun's virtual memory implementation is, at present, predominantly a port of the Berkeley VAX design to the Sun architecture. As such, it bears the marks of a design oriented toward a machine running in a timesharing environment and is not ideal for a single-user workstation. At the same time, Sun's virtual memory management works reasonably well in practice and is far superior to UNIX implementations without virtual memory.

This section unavoidably introduces some hardware subjects because memory is managed jointly by the Sun hardware and the kernel. It is in no sense a complete description of the Sun-2 Memory Management Unit.

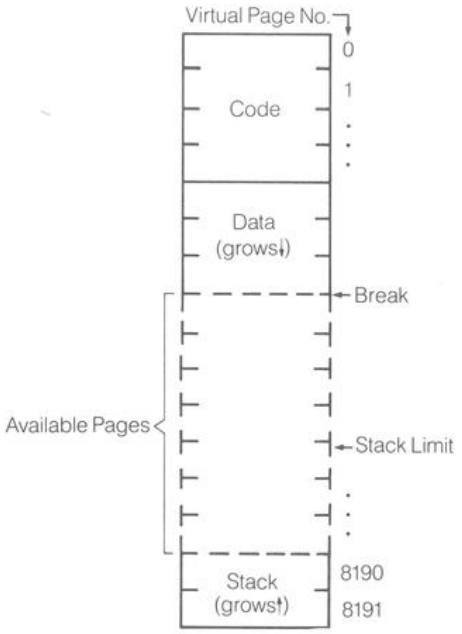
2.3.1 Segments, Pages, and Page Frames

The UNIX system defines the **segment** as the basic unit of memory management. As mentioned earlier in this chapter, each process has a code (or text) segment, a data segment, and a stack segment. All processes running the same program share a single code segment, but the data and stack segments of each process are private.

The memory management units of various PDP-11 models support the UNIX notion of segments well, allowing segments to be protected and swapped. They do not support virtual memory, however. Segment-based virtual memory is possible, but the variable size of segments complicates finding places for them when they are brought in from disk to memory. Consequently, most current virtual memory computers, including Suns, define the basic unit of memory management to be a fixed-size **page**.[7] A Sun-2 page is 2 kilobytes long. Each Sun-2 UNIX process runs in a private *virtual* address space of up to 8,192 pages or 16 megabytes, with its segments made up of these pages. Figure 2.3 shows how a process's segments are arranged in its virtual address space and its segments are composed of pages. (The "break" and "stack limit" references in the figure are explained toward the end of this section.) In contrast to the large virtual address spaces of processes, the Sun-2 architecture defines a single *physical* address space of 8 megabytes; a given workstation may have as little as one megabyte of physical memory. The physical address space is conceptually divided into page-size units called **page frames**. Because there may not be enough page frames to hold all the pages of a large process (or

7. The Sun-2 hardware also defines a second memory unit, called a **segment**, which is distinct from the UNIX notion of segment. A Sun-2 hardware segment is an aggregate of 16 pages whose purpose is to keep the address translation tables manageably small; it can be ignored for purposes of this discussion.

Fig. 2.3 *Process Segment Layout*



several small ones), at any instant many pages are likely to be held temporarily on a disk, or part of a disk, known as the **swap device**. Using data structures described in the next sections, the kernel shuffles pages between the swap device and page frames in a manner that is transparent to processes and, usually, to users.

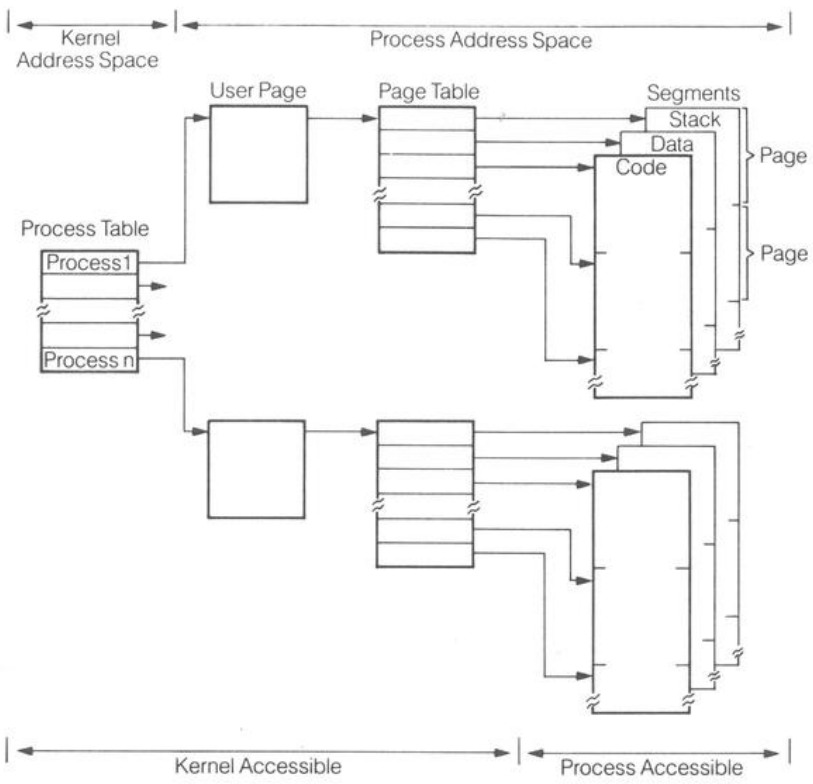
2.3.2 Data Structures

The kernel maintains an entry for each process in its **process table**. This table and other key memory management data structures are illustrated in Figure 2.4. The process table contains minimal information for each process, because the table is always kept in memory. More detailed information is kept in the per-process **user page**. A process's user page contains the bookkeeping data the kernel needs in order to suspend and resume execution of the process; for example, the process's register contents and file descriptors. The user page also has a pointer to the process's **page table**, the data structure that describes the current mapping (correspondence) between a process's pages and the workstation's page frames.

The **fork** and **exec** system calls are intimately associated with the memory management data structures. When a process **fork**s, the kernel:

- □ makes a new entry in its process table for the child process, copying most of its contents from the parent's entry.
- □ allocates a user page and a page table for the child, and fills them by copying from the parent's corresponding structures.
- □ allocates page frames for the child's page table entries and copies the contents of the parent's data and stack segments to the child's corresponding segments. The parent's code

Fig. 2.4 *Memory Management Data Structures*



segment is not copied, since the child initially executes the same program as its parent.

- allocates space on the swap device for the child's segments, page table, and user page.

The **vfork** system call eliminates some of the copying by filling the child's page table with entries pointing to its parent's page frames. However, sharing writable segments in this manner limits the behavior of both parent and child; such constraints are not always acceptable. Another solution would be to introduce another **fork** variant that marked the child's page table entries as "copy on write." Then data and stack segment pages would be copied one at a time from parent to child if, and when, the child attempted to write into them. All other pages would be shared.

When a process issues an **exec** call, the kernel frees the process's swap space, page frames, and page table. It then allocates new ones according to information contained in the object file the process is **exec**ing. The process's new segments can be loaded in two ways. By default, the kernel takes no explicit action and simply brings the pages in on demand, as described later in this section. (The data pages come from the object file; code pages come from the object file, unless they are already in memory or on the swap

device.) Alternatively, a program may be linked so that the kernel loads its segments when it is **exec**ed; this method pays off for small programs (under 32 Kbytes or so).

2.3.3 Memory States

From the standpoint of memory management, a process can be considered to be in one of three states:

Swapped A swapped process is completely resident on the swap device; the kernel keeps the disk address of the process's user page in the process table; from information recorded in the user page it can retrieve the page table and segments.

Resident A resident process has its user page and page table in memory, and usually, some of its segment pages as well.

Mapped A mapped process is resident, and in addition, its page table is loaded in the system **page map**. Only mapped processes can run. The page map, which is described more fully below, is actually a cache of 8 page tables that are likely to be needed in the near future. Therefore, the mapped "state" is more properly an optimization of the resident state.

The kernel changes the memory states of processes as necessary. Basically, when there are few processes and little demand for physical memory, all processes are mapped and memory management overhead is minimal. As the number of processes exceeds the number that can be mapped simultaneously, the kernel unmaps the least active processes and changes the more active ones from resident to mapped. Under conditions of high contention for memory, the kernel must move some processes between the resident and swapped states in order to maintain good performance. Later sections describe these state changes in more detail.

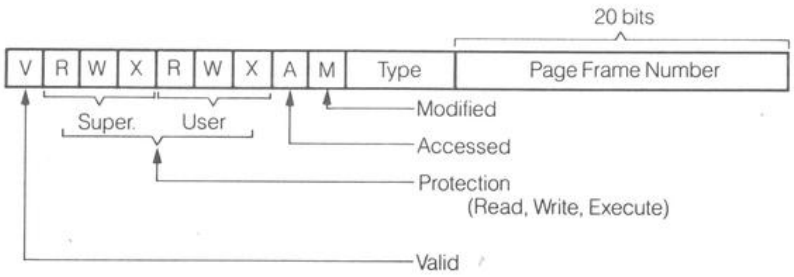
2.3.4 Address Mapping

A running process refers to memory with virtual addresses, which essentially consist of a virtual page number and a byte offset into the page. The hardware's memory management unit (MMU) translates the virtual page number into a page frame number, adds the offset and sends the resulting physical address to memory. Note that Direct Virtual Memory Access (DVMA)[8] transfers proceed by generating virtual addresses that are translated by the MMU in exactly the same way. Therefore, the following discussions of MMU services, such as memory protection, apply equally to references made by DMA devices as well as those made by the CPU.

The key to performing the virtual-to-physical address translation, or mapping, is the page map maintained by the kernel in the MMU's high-speed RAM. As mentioned earlier, each process has its own page table, containing one entry for each of its pages; when the process is running, its page table is loaded into the page map, which has a similar format. Figure 2.5 shows a page map entry. The fields in a page map entry are described below.

8. DVMA is a trademark of Sun Microsystems, Inc.

Fig. 2.5 *Page Map Entry*



Modified The MMU sets a page's modified bit whenever the hardware writes to the page; the kernel clears the bit when it updates the swapped copy of the page. Thus, the bit indicates whether a current copy of the page exists on the swap device.

Accessed Whenever the hardware "touches" a page (that is, reads it, writes it, or fetches an instruction from it), the MMU sets the page's accessed bit. By periodically clearing and checking accessed bits, the kernel can identify infrequently used pages; these are good candidates to page out to the swap device.

Valid The kernel clears the valid bit when it frees a page's page frame for reuse. An attempt to address an invalid page is detected by the MMU, which invokes the kernel to map the page to a page frame and restart the faulted instruction.

Protection The MMU checks every page access for conformance to the page's protection bits. For example, a bad pointer or array index cannot cause a write into a code segment because code segment pages are marked non-writable.[9] There are distinct protection bits for both the process (user) and the kernel (supervisor). These allow the kernel to store a process's page table and user page in the process's address space and yet make them inaccessible to the process. The storage occupied by the page table and the user page slightly reduce the effective size of a process's address space.

The protection bits keep a process from interfering with itself (and with some of the kernel's per-process data). The fact that each process has its own non-overlapping (except for shared code segments) page table prevents processes from interfering with each other and with the kernel.

Type The Sun architecture actually defines four physical address spaces: on-board memory, off-board memory, on-board I/O, and off-board I/O. The type field identifies the address space associated with the page. Because I/O registers, for example, are mapped into pages, device drivers can address these registers through the normal address translation mechanism. However, the type field is of no concern to the great majority of programs.

9. Data pages are marked executable as well as readable and writable. Thus, although it is not the usual case, the system does not prohibit programs, for example, that generate and execute code on the fly; in such cases, instructions are fetched from the data segment rather than the code segment.

To obtain good performance, there is not one page map but 8, one for each of 7 processes, or **contexts**, and one dedicated to the kernel. An MMU register, called the **context register** and maintained by the kernel, points to the map belonging to the running process. Because there are 8 maps, the processor can be switched among the 8 mapped processes by simply changing the context register; only when the process to be run is not mapped, must the overhead of loading its page table into a page map be incurred. When the kernel must map a new process, it overwrites the page map of the least-recently-run process, first updating the accessed and modified bits in the process's page table entries.

To speed response to interrupts and system calls, there are actually two context registers; one selects the running process's page map and one selects the kernel's page map. Which context register is used as the page map pointer for a particular instruction depends on the state of the processor, that is, whether it is running in MC68010 "user state" or "supervisor state." Normally the processor runs in user state; interrupts and system calls switch the MC68010 to supervisor state; the corresponding "return" instructions switch it to user state. Thus, the kernel need never change the context register in response to an interrupt or a system call; the memory references of all instructions executed in supervisor state are automatically mapped through the kernel's page map.

When execution switches to kernel code, the kernel's page map is used, so the kernel has no access to user process pages. When it needs access to a user page, as it must, for example, to execute a **`read`** or **`write`** system call, the kernel copies the data with the MC68010 Move Space instruction, which transfers bytes from one address space to another. However, when more than 512 bytes must be moved, the kernel temporarily adds the appropriate page(s) to its own page map and then issues an ordinary Move instruction.

2.3.5 Paging

The process of replacing the contents of page frames with different pages is called **paging**.[10] Two structures, in addition to the page tables, are central to paging, the **free list** and the **loop**. The free list contains page frames that are eligible to be reused. Page frames are added to the *head* of the free list when they are known to be no longer needed. Conversely, page frames are added to the *tail* of the free list when they may be needed again. Consider, for example, the termination of the last process executing a program (recall that code segments are shared between processes). The kernel adds the process's stack, data, user page, and page table frames to the head of the free list, since they can be of no use to another process. It adds the frames containing code pages to the tail of the free list, because it is possible that another process will execute the same program. Should this happen, the kernel will reclaim the code pages from the free list (if they are still there) rather than bringing them in from the

10. The kernel automatically locks a page that is the source or target of a DVMA transfer so it is not paged out during the transfer.

object file or the swap device. Page frames are allocated only from the head of the free list so that the pages that may be needed again stay associated with page frames as long as possible.

Page frames not on the free list are on the loop, a list that contains all allocated page frames, sequenced by physical address. However, frames containing kernel code and data are on neither the loop nor the free list because the kernel is not subject to paging.

2.3.5.1 Page Replacement

The **pager** is a system process whose job is to keep the free list large enough to maintain good performance. It runs when the free list drops below a lower threshold and continues until it has built the list back up to an upper threshold. The pager's replacement policy is to release, on a system-wide basis, page frames containing pages that have not recently been accessed.

To implement this policy, the pager maintains a frame pointer that it cycles through the loop in the manner of a clock hand sweeping the numbers on its dial. Given that this "hand" points to a page frame, the pager examines the page's accessed bit.[11] If the accessed bit is set, the pager simply clears it and proceeds to the next frame in the loop. If the bit is clear (the page has not recently been accessed), the pager marks the page invalid, adds the page frame to the tail of the free list, and proceeds to the next frame. If the not-recently-accessed page is also marked modified, the pager arranges for the page's swapped copy to be updated before adding the frame to the free list. In sum, the pager cycles through the allocated page frames freeing those that have not been accessed since its previous cycle.

2.3.5.2 Page Faults

The MMU detects an attempt to access a page that the pager has marked invalid. Such an attempt is called a **page fault** and invokes the kernel's page fault handler. An invalid page may be in either of two places; the page fault handler takes a different action depending on its location.

- If the page is on the free list, the page fault handler unlinks the associated page frame from the free list, adds it to the loop, marks it valid, and resumes the process. Reclaiming a page frame from the free list in this manner does not block the faulting process and is much faster than a disk read.
- If the page is paged out (the frame it formerly occupied has been allocated to another page), the page fault handler blocks the faulting process and schedules a disk read to retrieve the page from the swap device. Later, when the page has been read, the kernel allocates a frame from the head of the free list, adds it to the loop, updates the frame address in the process's page table, marks the page valid, and unblocks the process.

The combined actions of the pager and the page fault handler tend to keep frequently accessed pages associated with page frames,

11. The VAX has no accessed bit, so the standard Berkeley design uses the invalid bit to simulate it with software. As a result, the Berkeley design incurs more page faults than a Sun system due to its "overloading" the meaning of the invalid bit.

while little-used pages tend to migrate to the swap device. The pager simple-mindedly moves not-recently-accessed page frames to the free list, so they can eventually be reallocated. Concurrently, page faults taken on these pages nullify the pager's efforts. If a page is very frequently accessed, the pager will never see it marked not-accessed and will never add it to the free list. If a page is accessed moderately often, it may be added to the free list, but will be rapidly reclaimed by the page fault handler before it gets to the head of the list. Only the least-used pages get to the head of the free list and therefore have to be read from disk before they are used. The free list thus serves both as a source of available page frames and a cache of recently discarded pages that can be reclaimed quickly.

2.3.6 Swapping

The kernel cannot predict the memory reference patterns of an arbitrary group of processes. Accordingly, physical memory can be overcommitted; such an overcommittment is indicated when the pager cannot keep the free list above its minimum threshold. To forestall the possibility of thrashing (excessively high paging activity), the kernel initiates a measure more drastic than paging: it swaps whole processes to disk. Swapping processes out frees page frames; in addition to its segments, a process's page table and user page are eligible for swapping. More importantly, swapping reduces short-term contention for page frames (and CPU cycles). With less contention, some resident processes should run to completion, bringing the demand for physical memory back into the range that can be managed effectively with paging.

Swapping is the job of a kernel process called the **swapper**. It attempts to select for swapping the user process whose progress will be least impeded by losing memory residency. A process that has been blocked for a long time is likely to remain so (often it is waiting for keyboard input); therefore, the swapper selects the process that has been blocked the longest. If no resident process is blocked, it selects the process that has been resident the longest. This is an attempt to provide some measure of "fairness" among processes and is an example of an artifact from timesharing that is not ideal for a workstation, where a user may want to bias the swapper's selection.

A swapped process is swapped in when it both becomes ready and enough memory is available. After swapping a process in, the kernel makes sure the process makes some minimal progress before considering it for swapping out again.

2.3.7 Dynamic Allocation

The UNIX kernel leaves the implementation of dynamic memory management (for example, **malloc** and **free** in the C library, and **new** and **dispose** in Pascal) up to system libraries and language run-time systems. The kernel does provide, however, the mechanisms to support such implementations; these are the **sbrk** and **brk** system calls. **sbrk** extends a process's data segment (see Figure 2.3). The end of the data segment is called the **break**, and the **sbrk** system call "sets the break" to a new location. It does this

by adding the requested number of bytes, rounded up to a page boundary, to the data segment and returning a pointer to the base of the extension. **brk** can be used to shrink the data segment, returning pages to the kernel.

2.3.8 Stack Extension

In the Berkeley UNIX system each process has a maximum stack depth called its **stack limit**. A process inherits its stack limit from its parent; the limit can be changed by a C-Shell command or a system call. The kernel initially creates a one page stack for a new program. When, during execution, a data reference falls outside a process's segments (see Figure 2.3) but within the bounds of its stack limit, the kernel adds the referenced page, and any intervening pages, to the stack segment and restarts the faulted instruction. If the reference falls elsewhere, the kernel delivers a SIGSEGV (segment violation) signal to the process. Unlike the data segment, the stack segment cannot, at present, be shrunk during execution.

2.4 Input/Output

Compared to many operating systems UNIX system I/O is exceptionally simple and uniform. It is simple because there are only a few I/O system calls. It is uniform because operations on I/O devices of various kinds are interchangeable (constrained by the capabilities of the underlying device); this in turn makes a single program useful in a variety of situations.

2.4.1 Byte Streams

UNIX I/O system calls work on **byte streams**.[12] A byte stream is simply a sequence of bytes. It has no other system-defined structure: it contains no records, no blocks, and no extraneous data (for example, keys or length fields or end-of-file markers). The byte stream view of I/O minimizes storage requirements and simplifies the operating system (there are no "access methods"), but its most significant advantage is *uniformity*. For example, a program that translates a stream to hexadecimal notation works on *any* stream, whether it comes from a file, a terminal, or a communication line. As will be shown later in the report, the UNIX utilities exploit the stream-based nature of UNIX I/O to provide a very flexible set of tools.

While the system takes an unstructured view of streams, programs may interpret them according to a self-imposed discipline if they need to. For example, the compilers produce binary streams (object modules) that are structured for the convenience of the linker. Further, many programs use or produce **text streams**, streams of ASCII characters that are divided into lines by newline (0A hexadecimal) characters. But it is ordinary programs that establish these conventions when and if they are needed; the kernel

12. The phrase "byte stream" is not standard UNIX terminology, but is introduced here to emphasize the unified nature of UNIX I/O.

dictates nothing.

There are just two basic I/O system calls, **read** and **write** (notice that these are the same calls introduced earlier for pipes); they transfer a segment of a stream from or to a data area (for example, an array or structure) defined by a process. Where the stream ultimately comes from (in the case of a **read**) or goes to (for a **write**) is immaterial to the process issuing the call. The stream source or destination may be a file, a process on the same machine, a process on a different machine, a terminal, a printer, a communication line...anything that can generate or accept a stream of bytes.

2.4.2 Descriptors

Each byte stream accessible to a process is identified by a **descriptor**, that is, a descriptor is a handle on a byte stream. A process's descriptors are gathered in its **descriptor table**, one element of its process environment. Entries in the descriptor table are indexed from 0..*n* where *n* is a configuration parameter, typically 20. This index value is used to specify the target descriptor in a **read** or **write** call; that is, reading descriptor 4 obtains bytes from the stream currently associated with that descriptor. Because the descriptor table is part of a process's environment, a child process inherits access to all of its parent's byte streams. However, because the child's environment is a *copy* of the parent's, the child can alter its descriptor table without affecting its parent.

New descriptors are created differently according to the type of object associated with the byte stream. As mentioned earlier in this chapter, the **pipe** system call creates descriptors for the read and write ends of a pipe. The **open** call creates a descriptor for a file or a device, and the **socket** call makes a descriptor for a socket; **open** is described later in this chapter. In all cases, **open** returns the index value of the newly created descriptor which the process uses in subsequent **read** or **write** calls.

A descriptor table has a fixed capacity, limiting the number of streams a process may work with at once. The **close** system call notifies the system that no more I/O is to be done on a stream and frees its slot in the descriptor table for reuse. A process can also manipulate descriptor table entries with the **dup** system call, which makes a duplicate of a descriptor in the first available slot (that is, it saves a copy of a descriptor). The original descriptor can then be closed and a new descriptor created in the original position. To restore the initial descriptor, it can be **dup**'d from its saved location, after closing the new descriptor.

2.4.3 Files

In the UNIX system an **ordinary file** is a byte stream that has a name and is stored on disk.[13] The kernel tracks the size of each file and allocates space for files automatically and incrementally as they grow. The maximum size of a file is 512 megabytes, and a file must reside on one volume.

13. A file on a tape is considered an attribute of a tape drive and is read or written as a *device*.

The **open** call returns a descriptor for an existing file, or for a new file.[14] **open** takes three parameters, **name** (file names are discussed shortly), **flags**, and **mode**. The flags parameter tells the system how the process intends to use the file. The basic flags are read-only, write-only, and read-and-write, that is, update. Other flags, which may be specified in addition to those just described, are **create** (create a new file), **append** (start writing at the end of an existing file), **truncate** (set an existing file's length to zero, effectively overwriting its contents), and **exclusive** (return an error if the file exists). The mode parameter tells the system what **permissions** to associate with a newly created file. Permissions are the UNIX system's file protection mechanism and are discussed shortly.

Besides **read** and **write**, files respond to the **lseek** system call.[15] **lseek** effects random access on *any* file; there is no concept of a "random" vs. a "sequential" file in the UNIX system; every file can be manipulated in either way. **lseek** works as follows. Associated with each file is an implicit **I/O pointer**, maintained by the kernel, that indicates where in the stream bytes will next be read or written. As **read** or **write** calls are executed, the system moves the I/O pointer along by the number of bytes transferred. To jump to an arbitrary byte in the file, all that is necessary is to update the I/O pointer, and that is just what **lseek** does. Besides a descriptor, the call takes arguments specifying an offset by which the pointer should be moved, and the origin from which it should be moved (the origin may be the beginning or end of the file, or the current location).

2.4.4 Directories

To enable it to find a file when a process presents a file name, the UNIX system records file names in **directories**. A directory is just a file that happens to contain a set of file names and their associated disk addresses.[16] To maintain the integrity of the file system, the kernel prevents users from writing into directories; otherwise, a directory is indistinguishable from an ordinary file. This means that programs can extract information from directories (for example, the names of all files) by simply **read**ing them — no special provisions are required.[17] Every user has a personal directory called a **login directory**; directories subordinate to the login directory can be created in an unlimited fashion. Since every user has a separate login

14. In older UNIX systems, **open** operates only on existing files, and a different call, **creat** (note spelling), is used to create a new file. The newer **open** call is upward-compatible with the traditional **open** call; **creat** is also provided for compatibility.

15. **lseek** means "long seek." Early versions of the UNIX system had a **seek** system call that was capable of moving only 64K bytes in a file; **lseek** was added to permit repositioning anywhere in a file with a single call.

16. This is an oversimplification. A directory entry actually points not to the file itself but to a record called an **inode**. The **inode** contains pointers to the disk blocks containing the actual file. The level of indirection provided by **inodes** allows a single copy of a file to be accessible through multiple directories, each of which may give the file a different name. While a file may have many names, it has only one set of attributes (for example, access permissions, discussed shortly) since these are recorded in its **inode**.

17. However, it is better practice to use library routines supplied with the system to extract directory information. The library routines insulate a program from possible changes in the format of directories.

directory, the file names one user creates (subordinate to his/her login directory) will not conflict with like-named files belonging to other users.

By convention, users generally construct file names from lower case letters and numerals, although upper case letters and most special characters are legal. Again by convention, a file name often ends with an **extension** (suffix) that indicates the file's content. For example, **.c** (as in **binsearch.c**), **.f** and **.p** are the conventional extensions for C, FORTRAN 77 and Pascal source files respectively. These conventions are established by users and by programs that use the file system, not by the file system itself. In many UNIX systems a file name is limited to 14 characters in length; Berkeley extended the maximum to 256.

Directories are arranged in a hierarchy (see Figure 2.6 for an example). The directory at the top of the hierarchy is called the "root," a reference to its position in the inverted tree. A file in the hierarchy can be identified by either an **absolute path name** or a **relative path name**. An absolute path name traces from the root through any intervening directories to the file itself. For example, **/usr/terry/notes/apr22.txt** is the absolute path name to a file in Figure 2.6. The initial slash in the example stands for the root and tells the system that this is an absolute path name; subsequent slashes separate directory names.

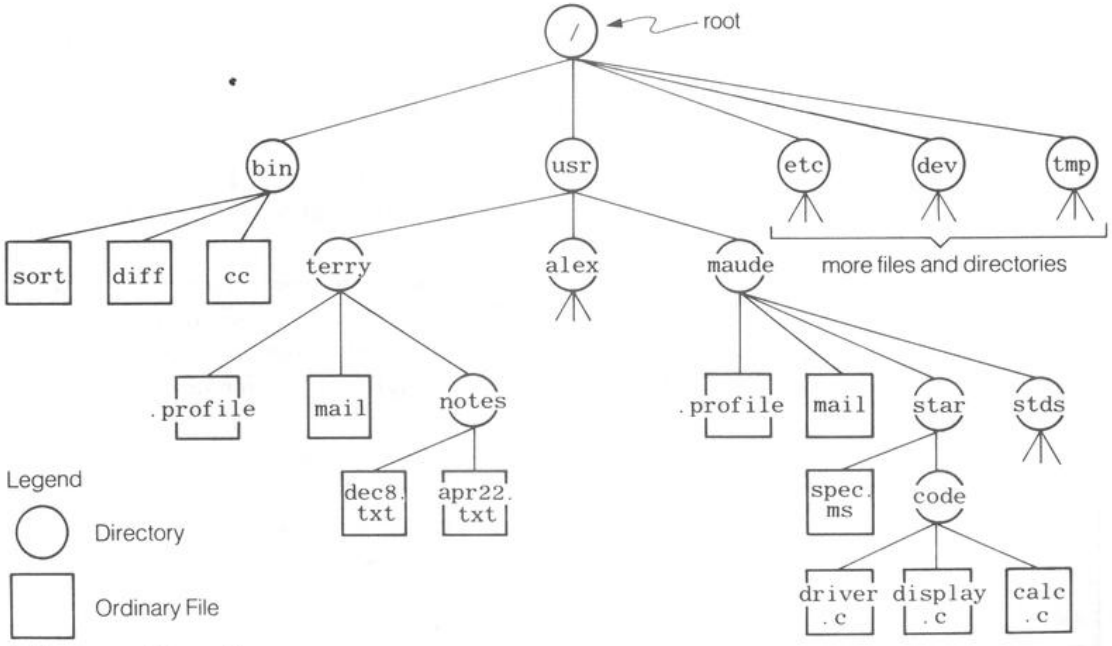


Fig. 2.6 *File System Hierarchy*

In practice, users cluster related files together in directories and tend to work for extended periods with the files in one directory (another example of the "locality of reference" principle). At any time in its execution a process has a **current** or **working directory** relative to which files can be named. The system interprets any path name that does not begin with a slash as relative to the current directory. If, for example, the current working directory of a

process is **/usr/terry**, then **notes/apr22.txt** refers to the file used in the previous example; similarly, if the current directory is **/usr/terry/notes**, then only **apr22.txt** is required to specify the same file. A number of system calls are available for changing the working directory and otherwise "navigating" about the file system tree.

While it appears to be a single entity to a user, a UNIX file system usually consists of several file systems. In other words, "the" tree of files actually consists of several subtrees. Each file system has a similar hierarchical structure; the individual file systems are linked to each other to form "the" file system. One file system, residing on the **root device** (a device predefined to the system at initialization time), is designated to anchor the overall file system. The subordinate file systems may reside on the same disk or on different disks; however, an individual file system must reside on a single disk.

The privileged **mount** system call attaches one file system to a directory of another. (Normally the receiving directory is empty. Should it contain files or directories, they will be temporarily inaccessible while the file system is mounted on the directory.) The **umount** (note spelling) call does the reverse, logically detaching a file system from the hierarchy. In this way multiple independent file systems, existing on one or more disk drives, can be integrated into a single hierarchy of files. The technique naturally supports removable disk packs, drives that must be taken offline, new drives, and so on. Moreover, Sun has extended the approach to allow file systems to be shared across the network, that is, one workstation may share a file system physically mounted on another workstation by "mounting" it on its local file system. The Sun Network File System is briefly described at the end of this section.

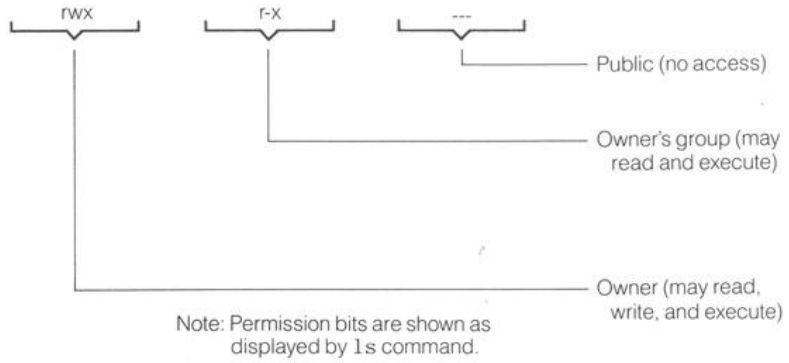
2.4.5 Permissions

Associated with every UNIX file are three sets of three **permission** bits (see Figure 2.7). The three bits define whether the file may be read, written or executed (any combination is permissible). Each set of permissions grants access rights to a different class of user. The three classes are the file's owner (initially its creator), users in the file owner's **group**, and the "general public" of users. A Berkeley extension allows a user to belong to multiple groups. A user's groups are assigned when the user's account is created on the system. A user might set the permissions on a file containing a program so that he/she can read, write, and execute it; members of his/her group may be allowed to execute the file, and other users may be granted no access to it at all.

Directories have the same nine permission bits, but they are interpreted differently:

- "Read" permission grants the ability to read the directory as a file, for example, to list the file names it contains.
- "Write" permission allows the addition and deletion of directory entries, that is, grants the ability to add a file to, or remove a file from, a directory.

Fig. 2.7 *File Access Permissions*



- "Execute" permission means "traverse" — a program may use the directory's name in a path name, but cannot read the directory's contents directly.

To prevent their contamination or deletion, system files are owned by the super-user. One might wonder why the concept of super-user is useful for a workstation dedicated to a single user who can login as **root** anytime. In fact, it does provide a measure of protection against a user's own careless errors, as well as mistakes made by colleagues borrowing a workstation. Everyone makes mistakes, but a mistake made when logged in as **root** can be catastrophic. For example, the command

```
rm -r *
```

recursively deletes all files in the current directory and all subordinate directories. It is thus a quick means of pruning large branches of the directory tree. Executing it while in the wrong directory, however, can wipe out the wrong files, possibly in great numbers. Two safeguards somewhat limit the damage when the command is issued without privilege. First, the system will ask for confirmation when it encounters a file that has only read permission (for this reason, users often mark their own important files read-only and edit copies of them). Second, the system will not remove a file owned by another user, for example, the system files which are owned by **root**. When **root** executes the

```
rm -r *
```

command, there are no safeguards; every subordinate file simply disappears without notice. Thus, a user logged in without privilege can shoot him/herself, but only in the foot; to inflict a headwound requires **root** privileges.

Each file has a tenth bit called **set-uid** which adds an important element of flexibility to the file protection system. There are often files that broad groups of users should be able to update in a *tightly controlled* fashion. Consider the system's **/etc/passwd** (password) file, for example. Any user should be able to change his/her own password, but allowing unrestricted write access to the file would constitute a gross breach of security. A running program

whose set-uid bit is set can change its **user-id** from the user who is running the program to the user who *owns* the program. It then acquires the owner's permissions with respect to the owner's files. Only the owner of a file can change the file's **set-uid** bit.

Here is how the **set-uid** bit enables the password file to be changed in a protected fashion. The password file is owned by **root** (the super-user); its permissions allow anyone to read the file (passwords are stored in an encrypted form), but only **root** has permission to write into it. There is a program called **passwd**, also owned by **root**, that any user can execute; **passwd** changes a password in **/etc/passwd**. The **passwd** program is stored as a file whose **set-uid** bit is on. When a user executes **passwd**, the program issues a **setuid** system call asking (in an argument to the call) that its user-id be changed to **root**. In executing the call, the kernel verifies that the program is owned by **root** and that its **set-uid** bit is on; in response it changes what is called the **effective user-id** of the process running **passwd** to the owner of **passwd**, that is, to **root**. This allows the **passwd** program to write into the password file. Note that **passwd** runs in its own process and **only** that process's effective user-id is affected by the **set-uid** bit. At the same time, it is important to recognize that a program like **passwd** that changes its **effective user-id** to **root** *must be a trusted program*; because it runs with privilege, there is essentially *no* check on its behavior.

The **set-uid** facility is not restricted to use by **root**; any user can restrict access to his/her files to his/her own programs, yet allow anyone (but not necessarily everyone) to execute such programs. Again, such a program *must* be trusted, because it can obtain access to *all* of the owner's files. A program that can set its effective user-id to **root** can do other things besides updating files owned by **root**; it can execute privileged system calls, such as **setpriority**.

Closely related to **set-uid** is **set-gid**, which changes the effective user-id not to the program's owner, but to the owner's group. Thus **set-gid** relaxes control over file access to programs that belong to any member of the owner's group.

2.4.6 File Sharing

In the UNIX system it is perfectly possible for two processes to open the same file; indeed, as mentioned earlier, a child process inherits its parent's open files. Further, any process that knows the name of a file and has appropriate permissions can open it. Note: processes that share files as a result of inheritance also share I/O pointers; therefore a **read** by either process will change the other process's position in the file. Normally this does not cause difficulties because the parent wants the child to take over the files while the child runs. When the child terminates, the parent resumes processing the files. Processes that share files as a result of separate **open**s have separate I/O pointers. In either case, buffers are shared system-wide, so a change made by one process will be seen by the others sharing the file.

If multiple processes wish to update a file in a consistent fashion, they need to synchronize their accesses so the updates are applied one at a time. The classic way to do this, and the only way in many UNIX implementations, is to have the updating processes attempt to create a "lock file" before proceeding to change the shared file. To do so, each process might execute code illustrated by the psuedocode in Figure 2.8.

Fig. 2.8 *Using a Lock File*

```
(* try to create the lock file *)
exclusive := open("lockfile", write_only,
        create_exclusive);

(* did it already exist; that is, did open fail? *)
if exclusive < 0
then
        (* ...delay and try again...
          someone else is updating *)
endif;

(* we have created the lock file,
  giving us exclusive access to
  the shared file *)

(* ...update the shared file... *)

(* delete the lock file
  so someone else can create it *)
close(exclusive);
unlink("lockfile");     (* unlink deletes a file *)
```

Besides being clumsy, there are a few difficulties with the lock file approach:

- It is slow.
- A system crash leaves lock files lying around; these must be removed manually.
- It doesn't work for privileged processes, which can always create files.

The Sun UNIX system provides a better way, using a server process. Two general approaches are possible.[18] In the first approach one server process is designated to update the file, with other client processes sending it update requests via a socket. The alternative is a server that serializes access to the file, but allows a process which has been granted access to do its own updating.

18. The Berkeley **flock** system call is an alternative that is an improvement over lock files but has its own difficulties. Like a lock file, it is an *advisory* lock, which only works if all updating processes observe the convention. More importantly, **flock** is only applicable to local files, not to files shared across the network.

2.4.7 File System Implementation

The UNIX file system completely hides the physical characteristics of the underlying disk (for example, its block size), and is happy to store or retrieve a single byte or 10,823 bytes; With such generality, one might well question its ability to perform. And, indeed, file system speed is probably the crucial factor affecting the overall performance of a UNIX implementation.

All UNIX systems utilize a buffering-caching scheme to minimize disk accesses and to overlap I/O with processing; both ordinary files and directories are cached. One pool of buffers is shared globally across all file systems and all processes using those file systems. Each buffer is the size of a file system block. In response to a **read** request, the system searches for a buffer that already contains the block; if such a buffer exists, the kernel simply moves the data to the process's data area and returns, having performed no physical I/O. If no buffer holds the sought-for block, the system initiates a physical read into the least-recently-used buffer; later, when the disk driver has filled the buffer, the system copies the data from the buffer. The kernel recognizes sequential access of a file, and pre-reads blocks to keep the buffers ahead of process requests.[19]

write system calls are handled similarly. The kernel looks for the buffer into which the process's data should be written; if the buffer exists, the system copies the data into it and marks the buffer "dirty" (needing to be written), but performs no I/O. A dirty buffer is scheduled for writing when the system needs a buffer to hold a new block and the buffer is the least-recently-used in the pool. If a block into which data is to be written is not present in a buffer, it is read in as described for a **read** call and then updated as described here.

The principal upshot of this caching technique is that frequently accessed blocks (for example, current directories) tend to stay in memory, allowing **read** and **write** calls to be completed quickly without physical I/O. On the other hand, the UNIX system's practice of delaying physical writes until a buffer is needed, means that in the event of a system crash, data a process thought was written may actually have been left in a buffer. (To reduce the potential magnitude of this problem, the kernel executes the **sync** system call every 30 seconds. This call writes all dirty buffers, bringing the state of the disk into synchronization with the state of the buffer pool.) Processes that must know that a **write** has really updated the disk can issue the **fsync** call, which works like **sync** for a single file.

As mentioned, all UNIX systems utilize this buffering-caching technique. However, much of the rest of the usual file system implementation is too simple-minded to realize high levels of performance. For example, after the system has been running for a while, new disk blocks are allocated in essentially

19. The read-ahead algorithm is simple. When block *n* of a file is to be read, the system checks to see if block *n-1* was the previously read block of the file. If so, block *n* is read and I/O is also started for block *n* + *1*. When the process subsequently requests block *n* + *1*, it will already be in memory or on its way in.

random locations over the disk, necessitating long seeks even for sequentially accessed files. One of the major contributions of 4.2BSD is a redesign of the file system internals, while retaining the familiar interface. The result is several times better performance.

Here, very briefly, and highly simplified, are the 4.2BSD alterations:

- File system blocks are 4K bytes and a block can be broken into separately allocated **fragments** whose size is a multiple of 1K bytes.[20] A file occupies 0 or more 4K blocks plus a fragment, if needed, of 1K, 2K or 3K bytes. (For example, an 11,000 byte file takes two whole blocks plus a third block fragment of 3K; the 1K fragment remaining in the third block is available for allocation). The larger block size transfers more data per disk transaction and makes fewer files subject to indirect access penalties.[21] At the same time, the smaller fragment keeps disk utilization roughly on a par with the traditional file implementation (small files tend to predominate in a UNIX system).
- The system knows about and exploits hardware characteristics: processor speed, disk rotational speed, and so on.
- To the extent possible, file blocks are allocated on the same cylinder in an optimal position for sequential access, considering the rotational characteristics of the drive.
- Files in the same directory are located near each other to speed operations that affect all files in a directory.

The 4.2BSD file system is also more robust than its predecessor. The critical piece of information in any UNIX file system is called the **super-block**. If the information in this block becomes unreadable, the file system cannot be repaired after a system crash or improper system shutdown. 4.2BSD replicates the super-block in such a way that any single track, cylinder or platter can be lost and a super-block will still be available. Moreover, the disk is updated in well-defined stages so that a post-crash recovery program (called **fsck**, for file system check) can either complete unfinished updates or back them out cleanly, leaving the file system internally consistent.

2.4.8 Devices

Devices fit naturally into the simple UNIX model of stream I/O; after all, just about any device can supply or accept a stream of bytes (or both). For example:

20. Block and fragment sizes are actually configuration options and can be different for each file system; these are the common values, however.

21. To keep **inodes** reasonably sized, only the first 12 blocks of a file are recorded in the **inode** and are therefore directly accessible. For larger files, up to three additional **inode** entries point to other structures that contain more block addresses. Thus, the larger the block size, the larger the file that can be addressed without indirection.

- A printer is a write-only stream.
- A terminal keyboard is a read-only stream and its display is a write-only stream.
- A tape drive is either a read-only or write-only stream (or a read/write stream, if a tape file is being extended).

To make devices easy to deal with, they are in fact integrated into the file system, where they are known as **special files**. Every UNIX system has a directory called **/dev** which contains an entry for each device in the system. A device name is identical to a file name and may therefore be passed as a parameter anywhere a file name is permitted. A particularly useful pseudo-device is **/dev/null**, which corresponds to the "bit bucket" provided in many operating systems. Reading from **/dev/null** produces an immediate end-of-stream indication, while bytes written to this device simply disappear.

To control access, file permission bits apply to devices as well. In general, devices are owned by **root**, and permission to write sensitive devices (for example, disks) is limited to **root**. When a user logs in on a workstation or terminal, the system gives ownership of the terminal or workstation to that person until he/she logs out. By default, all the terminals on a system are publicly writable, allowing messages to be sent among users via the **write** command. Such messages, however, can be intrusive, especially while editing a file.[22] To stop such intrusions, the user has only to alter the permissions on the terminal to make it unwritable to group members and members of the public.

Devices are not completely identical with files, of course, because many of them cannot fully mimic the disk drive that underlies a file. For example, **lseek** is really designed for files and disk devices, not terminals. In a slightly different vein, a **read** to a keyboard normally returns at the end of a line, so a program will often receive fewer characters than it asked for (of course, a well-written program should expect the same "short read" at the end of a file, so compatibility between file and keyboard input is really not an issue). Despite these minor, and inevitable, differences, the fundamental identity of files and devices (and pipes and sockets) is extremely useful in practice. For example, the UNIX **sort** utility sorts the lines in a text stream, writing the output to another stream. It makes no difference whether the input comes from a communication line, the keyboard, or a file, and the output can similarly be a file or a device or a process (via a pipe). Yet this versatility is obtained without a line of code in the program devoted to the idiosyncracies of (present or future!) devices.

Some devices are capable of more operations than **read** and **write**; intelligent terminals and tape drives are the most common examples. Each device, therefore, may support a number of

22. Actually, most people use the non-intrusive **mail** utility, described in a later chapter, to communicate. This example just shows how the permission bits fit devices as well as files.

device-specific operations via the **ioctl** system call; a parameter identifies the particular control command to be executed. A block containing device-related data can also be passed to the device driver in an **ioctl** call. Programs that use **ioctl** calls, of course, are device-dependent, as the **ioctl** commands available for one device are not necessarily available on another.

2.4.9 Terminal Database

Most users interact with the Sun UNIX system through a Sun workstation. Character terminals, however, can be connected to workstation serial ports to allow multiple users to share a workstation. With the addition of multiplexer boards, moderate numbers of terminals can be attached to a Sun system. Such terminals can often underscore, blink, update part of a screen, and so on. Unfortunately there are no universally adopted "escape sequences" for invoking these operations — even though the operations are in many cases identical. The Berkeley **termcap** file brings a measure of order to this chaos by defining a set of generic terminal operations and the code sequences that implement these operations on scores of terminals. Since **termcap** is an ordinary text file, new descriptions can be added simply by editing the file. **termcap** has been adopted by many other UNIX implementations.

2.4.10 Standard I/O And Redirection

The UNIX system defines the streams associated with descriptors 0, 1 and 2 as a process's **standard input**, **standard output** and **standard error** respectively. Except when there is good reason not to, programs read their input from the standard input, write their output to the standard output, and write diagnostics to standard error. Since a process inherits descriptors from its parent, its basic I/O channels are already set up; it need only establish descriptors for the auxiliary files and devices it needs. In addition to saving a small amount of effort, using the standard descriptors gives a program a large measure of universality automatically. Since files, devices, pipes and sockets all work essentially the same, a program will work with its standard descriptors hooked up to any of them. Thus, to take a simple example, a program that counts words in a text stream, writing the total to the standard output, works when the input is taken from a terminal, from a file, from another process, and likewise for the output. Programs that observe this simple convention tend to work together without any additional design effort by their programmers.

By default, the standard input is associated with the keyboard of the user who ultimately invoked the process; the standard output and standard error are associated with the user's display. As a rule, the standard error retains its association with the display, since error messages that disappear down a pipe, for example, are not very useful. Often, however, a new child process will **redirect** its standard input and/or standard output, before **exec**ing a program. (In this way the **exec**ed program need not be concerned with where its input comes from or where its output goes.) The most

common example of this is the shell, which provides users with a very convenient notation for asking that the standard input, output, and error of a program be redirected before it is executed; how this is done is described in the next chapter.

2.4.11 Non-blocking I/O

The standard UNIX I/O interface is synchronous to the caller; a calling process is blocked until its request can be satisfied. Often, in the case of files, the buffer cache makes this almost instantaneous. There are situations, though, in which a process does not want to block because of an I/O operation that cannot be completed immediately. Suppose, for example, that a process is monitoring several communication lines and wishes to read data from *any* of them. If it were to read a line on which no data was available, the process would block until data arrived on that line — even though data might be present on one or more of the other lines. To permit the straightforward implementation of these kinds of processes, Berkeley introduced the notion of **non-blocking I/O**. To use this facility, a process issues the **fcntl** system call before it does any I/O on the descriptor. (**fcntl** stands for "file control" — a more accurate description would be "descriptor control options.") One of the parameters passed to **fcntl** tells it to mark the descriptor as non-blocking.

Reading from a non-blocking descriptor produces an identifying error code if the **read** cannot be satisfied immediately because no data is available. Writing to a non-blocking descriptor transfers as many bytes as possible without blocking; as usual, the call returns the number of bytes actually written. If no bytes can be written, a **write** to a non-blocking descriptor also returns an identifying error code. Thus, where normal UNIX I/O can be thought of as unconditional, non-blocking I/O is conditional — a request on a non-blocking descriptor means "do it now if it is possible, otherwise return an error."

A process using non-blocking I/O can repeatedly poll (that is, try to **read** or **write**) the descriptor(s) of interest. Such polling, however, can waste a good deal of CPU time in some cases. Alternatively, the process may ask to be signalled asynchronously when a non-blocking descriptor becomes ready for I/O. This is also done with the **fcntl** system call. When invoked, the process's signal handler can issue the **select** call, described below, to find out which descriptor has become ready.

The Berkeley **select** call provides an alternative method of determining when I/O is possible on a descriptor; it can be used with both blocking and non-blocking descriptors. Furthermore, **select** has a multiplexing feature that makes it possible to find out when I/O is possible on any of several descriptors. **select** takes bit string arguments that specify the descriptors of interest; it returns the total number of descriptors that are ready for I/O. **select** also returns bit strings that specify which descriptors are ready to **read** and which are ready to **write**. **select** can be told to return immediately, or to block the process until at least one

descriptor is ready. A timeout argument can be specified to keep the process from blocking forever, or to allow the process to periodically do something else.

2.4.12 Network File System Overview

Most of the I/O facilities described in the previous sections are available on any UNIX system implementation, and all of them are provided in 4.2BSD implementations. 4.2BSD also provides low-level network I/O facilities, such as sockets, which are beyond the scope of this report. At a higher level, there are facilities in 4.2BSD for logging into a remote machine, copying files between machines, and executing single commands on remote machines. To these basic 4.2BSD network capabilities, Sun has added a network disk (**nd**) protocol that permits a diskless workstation to use (part of) the disk of another machine on the network. In this way, diskless nodes can boot, page, and swap across the network, eliminating the cost, noise, and space requirements of a local disk. **nd** and the network are fast enough so that several diskless workstations sharing a high-performance network disk exhibit about the same performance as machines with local disks.

Where **nd** makes a single network *disk* appear to be locally mounted, Sun's Network File System (NFS) makes any number of remote *file systems* appear to be mounted locally. In other words, NFS allows files to be accessed with standard UNIX I/O calls, independent of the location of the files. Users and programs see the usual hierarchy of directories and files; they manipulate these files and directories in the usual way, subject to normal permission checks. The fact that parts of their file system may reside on various machines around the network is invisible and of no concern.

As important as its functional capabilities is NFS's conformance to Sun's Network Services Architecture. This architecture is designed to support the cooperative operation of heterogeneous machines and operating systems on a common network. For example, NFS can be implemented by other vendors, allowing files to be shared transparently among Suns, mainframes, supercomputers, database machines, personal computers, and so on. These machines need not be based on any particular processor, nor must they run the UNIX operating system. In addition to file sharing, other network services may be provided by Sun and by other manufacturers; such services may supplied by and used by any machine on the network, independent of its architecture and its operating system. To encourage the development and propogation of network services, including NFS, to a wide range of machines, Sun has placed the key network service building blocks in the public domain. For the NFS service specifically, Sun is publishing a specification of its protocol.

2.4.12.1 Servers and Clients

In the network services model, the computers on the network are called **nodes**; each node has a different name. A given node may act as a **server** or as a **client** or as both. A server provides a service to processes or users on one or more clients. One such service is remote file system access, the service provided by NFS. A typical NFS file

server is a dedicated machine with high-performance disks, but as mentioned, any node can be a server, and a node may be both client and server; that is, it may both supply and use network services. A node "becomes" a client or a server by simply issuing appropriate system calls and/or commands — no formal registration is required, nor is there any fixed relationship between clients and servers. Adding a new client node to the network is simple; to use existing network services, it must simply learn the names of the servers and the interface they have defined to their services.

NFS provides a set of operations that allow servers to export file systems to the network and clients to import these file systems. Like other operations that manipulate file systems, these are privileged operations. Most often they will be executed as commands from **/etc/rc.local** (this file contains commands that are automatically executed when a system is booted). However, they can be executed at any time by a user or a program with **root** privileges.

2.4.12.2 Server Functions

An NFS server initiates service by issuing a **nfsd** command. **nfsd** starts a specified number of daemons (by default 4) on the server; these daemons field incoming requests for service. To make a file system available to clients, the server places the file system's name in a file called **/etc/exports**. Optional data in **/etc/exports** enable the server to limit the nodes that may import a file system. To withdraw a file system from the network, the server simply removes the file system's entry from **/etc/exports**. Any subsequent client reference to a such an "embargoed" file system is returned to the client just as if the server had never exported the file.

2.4.12.3 Client Functions

To gain access to an exported file system, a client simply **mount**s the file system as if it were recorded on a local disk pack. Sun has added parameters to **mount** that tell NFS where to find the file system on the network. A client can remove a network file system from its file system tree by issuing an **umount** call in the normal way.

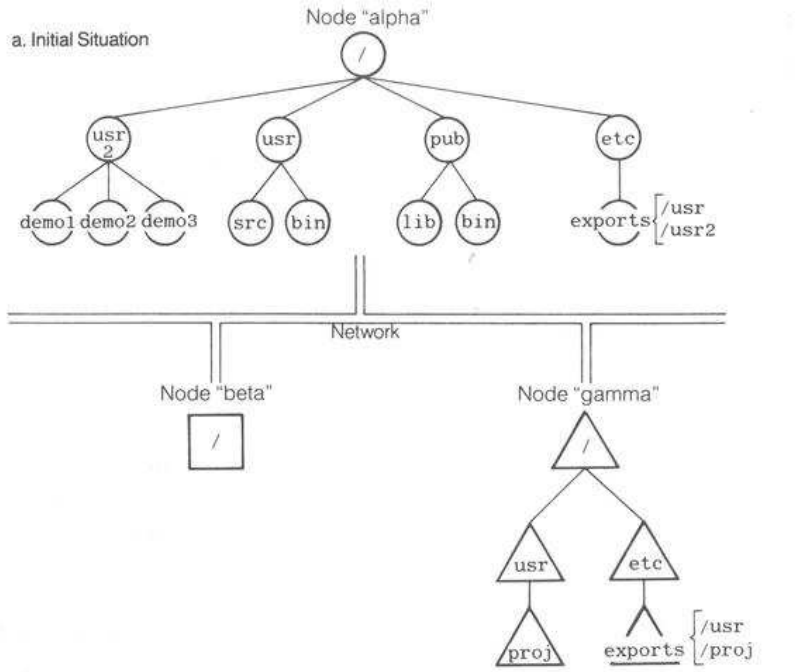
Figure 2.9 shows some typical, if simplified, interactions of three network nodes, one a server, one a client, and a third that acts as both server and client.

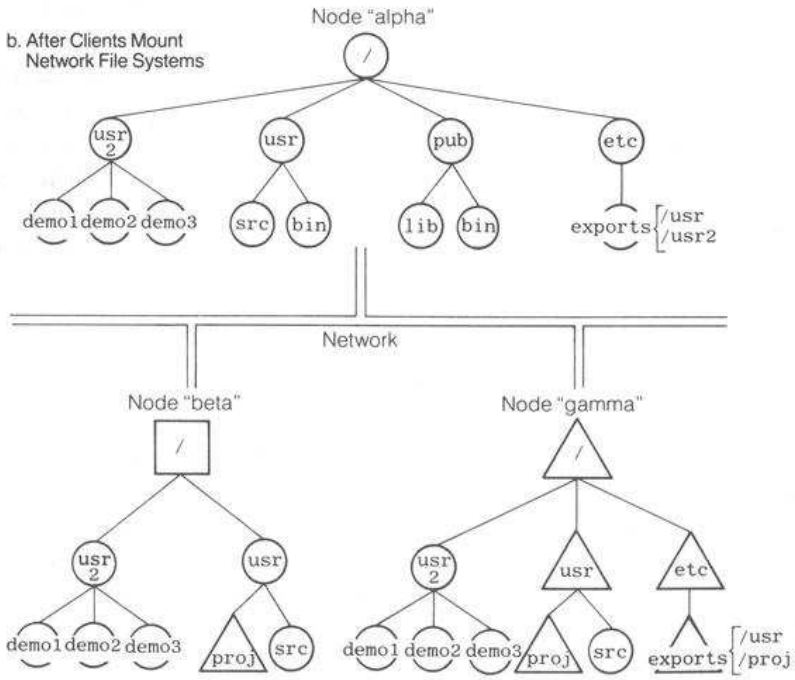
Having built a file system tree from some combination of network and local file systems, a client can issue ordinary UNIX I/O calls on the files. The physical location of a file in the tree is essentially transparent; not only do the normal file operations, such as **read**, **write** and **lseek**, work normally, but so does permission checking and so on. As a result, virtually all existing UNIX applications operate properly with network files.

NFS does not totally emulate the UNIX file system, however. For example, remote devices (such as modems and tape drives) cannot be accessed via NFS as UNIX special files.[23] Remote special

23. They can, of course, be accessed by logging into the associated node.

Fig. 2.9 *Exporting and Mounting Network Files*






```
mount -t nfs alpha:
  /usr/src /usr/src
mount -t nfs alpha:
  /usr2 /usr2
mount -t nfs gamma:
  /usr/proj /usr/proj
```

```
mount -t nfs alpha:
  /usr/src /usr/src
mount -t nfs alpha:
  /usr2 /usr2
```

files, and a few other UNIX file system details, have been sacrificed in order to give NFS a stateless protocol, as explained below. (Devices are inherently stateful.) The result is a system that provides

the most important file system operations at a higher level of reliability than would be possible with complete emulation.

2.4.12.4 NFS Implementation

The Network File System has been carefully designed to provide good performance, to be minimally affected by node failures, and to conform to the open systems principle.

To eliminate task switching overhead, Sun has built NFS into its UNIX system kernel. Servers are multithreaded and process multiple requests concurrently. Shortcuts in the code bypass processing in common cases. Both servers and clients perform asynchronous read-ahead, and clients write behind.

NFS allows users to make performance/space tradeoffs to obtain the performance they need in their environments. For example, shared file systems can be spread across multiple servers to gain the effect of multiprocessing. Frequently accessed read-only files can be duplicated on multiple servers, again to distribute the file processing load across multiple machines. Finally, clients with local disks can choose which files to access over the network and which to store locally.

Occasionally failing network nodes are a fact of life, and NFS has been designed to operate robustly in the face of such failures. The key to its robustness is a *stateless protocol* between the client and server sides of NFS. Basically this means that a server can treat every request from a client as brand new; it need remember nothing from the client's former transactions. As a result, the failure of a client does not affect a server in any way; the server need not be notified of the failure, since the server has kept no state information that must be cleaned up. If a server fails, clients need only retry their requests; when the server comes back up, they will be honored normally.

As mentioned, NFS is an example of a Sun network service. Inherent in the notion of network services is the idea of openness; that is, the network and the services available on it are open to nodes that are running non-Sun machines and operating systems. Such nodes may be attached to the network and act as clients and servers just as Sun workstations do. For each service they wish to use or supply, the node must implement the appropriate half of the client/server protocol. To assist other manufacturers in making their machines operate as NFS clients or servers, Sun is placing the NFS protocol in the public domain. As standard network services such as NFS become widely available, networks will become increasingly heterogeneous; users will be able to configure networks with nodes specific to their own needs and desires.

NFS and all network services are based on the Remote Procedure Call (RPC) and External Data Representation (XDR) developed by Sun. RPC provides a standard way for programs running on different operating systems to call each other, passing arguments and returning results. Programs that use RPC are shielded from the calling conventions of different operating systems; a client process calls RPC just as it would any local procedure; it is blocked until the call completes. Meanwhile, RPC on the client translates the call into a standard form and sends it across the

network to the appropriate server. RPC running on the server receives the call data, translates it from the standard form into a form that makes sense in the server's environment, and invokes the called procedure, passing it parameters. Results are returned by retracing the same steps in reverse.

XDR is a standard way to represent data passed between different machines. Programs use XDR to shield themselves from the byte ordering and data structure packing that vary from one machine and compiler to another. Because XDR and RPC provide the basis for constructing all network services, Sun has placed their code in the public domain.

2.5 Timers

A process can obtain the local time of day and Greenwich mean time with the Berkeley **gettimeofday** system call. The value returned is expressed in seconds and microseconds since midnight 1 January 1970. A privileged process may set the time with **settimeofday**.

Each process has three interval timers that it can set and query with the **setitimer** and **getitimer** system calls. These timers count down from their settings to zero and issue signals when they expire. One of the timers runs continuously, one runs only when the process is executing its own code, and the third runs when the process is executing its own code or kernel code.

The **time** and **profil** calls found in other UNIX system implementations are also available.

3 *The Shell*

The UNIX system kernel essentially provides a file system and an environment in which processes can execute. These processes are initiated by users with the aid of a program called a **shell**. This chapter describes a shell, in fact two of them. The discussion is divided into three parts. The first describes how a shell works, noting particularly its relationships with the kernel. The second part shows how a shell can be used to execute commands interactively. The last part describes shell programming, the important property of a shell that distinguishes it from typical command interpreters.

As mentioned, a shell is functionally equivalent to the command interpreter found in other operating systems; users give commands to a shell and the shell makes the proper system calls. *Unlike* most command interpreters, a UNIX system shell is wholly distinct from the kernel; not only is it a separate program, it is an ordinary program. Why this separation? The answer is flexibility. It is not easy to design a user interface that works really well for a range of users — and even at a single site, users may range from systems programmers to clerks to managers. Nevertheless, most operating systems embed the user interface in the operating system, dictating the style of user-system interaction, and also making it very difficult to change. Because a shell is an ordinary program, a UNIX site can replace it on a per-user basis. Further, the services provided by a shell are available to other programs as well as users; a process can simply **fork** a child process which **exec**s a shell as it would any program.

There are two standard shells, at least one of which is available on every UNIX system; both are supplied with the Sun UNIX system. The two shells are named **sh** and **csh**. **sh** stands for the **Bourne shell** (written by S. R. Bourne at Bell Laboratories) and **csh** stands for the **C-Shell** (so-called because its syntax resembles that of C); **csh** was written at Berkeley. Both shells can be used as interactive command interpreters and as programming languages. However, the C-Shell is a little better for interactive work, while many people prefer the Bourne shell for programming. (The interactive part of the C-Shell is mostly a small superset of the Bourne shell, making it easy for users to move between them.) Accordingly, many people use both shells, and the arrangement of this chapter follows this widely observed convention: the C-Shell is described as "the" interactive shell and the Bourne shell is described as "the" programmable shell. First, though, a side trip to see how either shell does its job.

3.1 How the Shell Works

The bulk of this chapter describes the command and programming interfaces the shell presents to the user. This section describes the shell's other interface: its interaction with the kernel. Besides exposing some of the shell's inner workings, it provides a useful review of the kernel, showing some of its facilities in action. As will be seen, much of what the shell does for the user is implemented in a remarkably simple fashion due to the design of the kernel.

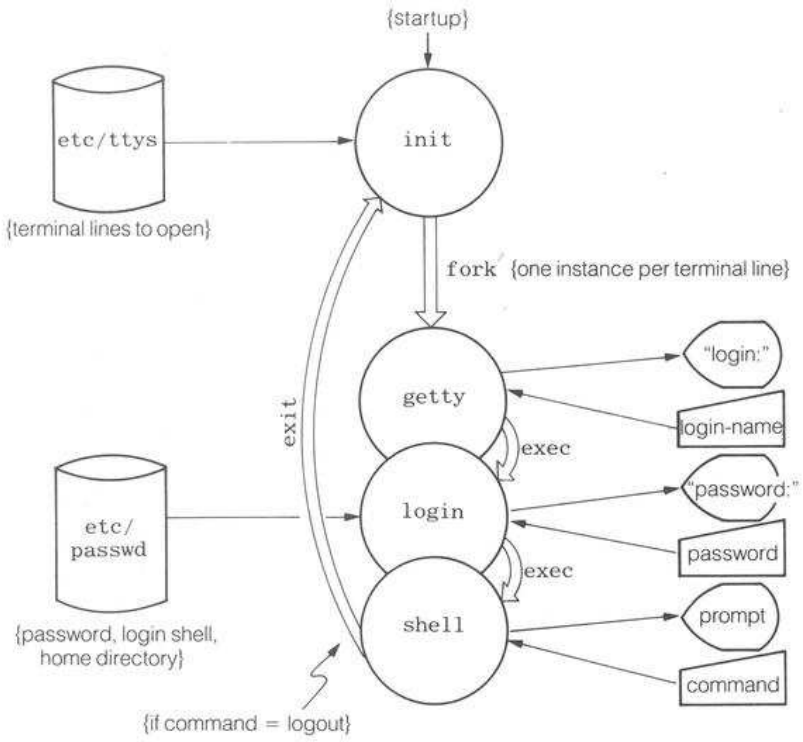
3.1.1 The Login/Logout Cycle

When a UNIX system starts up, a primordial system process called **init** is created. As shown in Figure 3.1, **init** consults a file called **/etc/ttys**[1] to find the terminal lines that should be opened. **init** then **fork**s a process for each of these lines. Each child process in turn **open**s its line (which may take some time if the line is a dial-up) and **exec**s a program called **getty**. **getty** displays a login prompt on the terminal. When **getty** receives a login-name from the terminal, it **exec**s a program called **login**, passing it the login-name. **login** then prompts for a password, turning off echoing while the user types it. Consulting the system's **/etc/passwd** file, **login** finds the login-name, encrypts the typed password and compares it to the stored password. If the encrypted passwords match, **login** looks up the name of the user's **login shell** (also in the password file), and **exec**s this program. If the login shell entry says **/bin/csh**, the C-Shell is started; if it says **/bin/sh**, the Bourne shell is started; if it says **/usr/terry/ownshell**, whatever program is stored in that file is started. Thus, there may be several shells on a given UNIX system, each presenting a command interface appropriate for a particular user. For example, if for some reason, a user only writes and executes BASIC programs, that user's login shell entry might be **/bin/basic**. Upon logging in, the user would be presented with a BASIC interpreter.

Another entry in **/etc/passwd** identifies the user's initial working directory, called the **home directory**. The C-Shell looks there for two files named **.login** and **.cshrc**.[2] These text files, if present, contain shell commands that the user wishes to be executed routinely. The **.login** commands are executed whenever the user logs in and the **.cshrc** commands are executed whenever the user starts a C-Shell.[3] A shell command frequently seen in **.login** and **.cshrc** files is **set**, which sets the value of one of the shell's **built-in variables** (it can also set user-defined variables). These variables define such things as the directories the shell will search when looking for a command, the type of terminal you are using, and so on; several of them will be described in this chapter. When

1. "tty" is an abbreviation for teletypewriter, the dominant terminal at the time the UNIX system was designed.
2. File names of the form *.xxxx*rc are called "are-see" files; the letters "rc" stands for "run command." Several utilities use them to execute user-specific standard startup sequences.
3. Since a shell is an ordinary program, *any* process can start an instance of it. Some editors, for example, have a command that permits the user to execute a shell command from the editor. To implement this shell escape, as it is called, they simply **fork** a process that **exec**s a shell. If a C-Shell is started in this manner, it will begin by executing the commands in the user's **.cshrc** file. By the way, the equivalent of **.login** for the Bourne shell is a file called **.profile**.

Fig. 3.1 *Login/Logout Cycle*



the C-Shell has initialized itself, it displays a prompt (by default *hostname*%, where *hostname* is the name of the machine the shell is running on), indicating its readiness to accept a command from the keyboard.

Having initialized itself, the shell basically executes an infinite loop in which it does the following:

- get next command;
- **fork** child process to run command;
- wait for child to terminate;
- repeat;

Later sections of this chapter elaborate on the shell's "main loop."

To stop the shell you type either the **logout** command or Control-D. Control-D is the special character that signifies end-of-stream from a keyboard — the shell interprets this as meaning there will be no more commands, so it logs you out.[4] **csh** executes a logout by terminating itself like any other process: it issues an **exit** system call. When **init** learns (via completion of its **wait** system call) of its child's termination, it starts the login cycle again by **fork**ing another child that **execs getty**.

4. There are a number of programs besides the shell that stop when you indicate that you have no more input for them by typing Control-D. It is fairly easy to make the mistake of thinking you typing to one of these programs when you are actually typing to the shell. In such a situation typing Control-D will summarily log you off the system. This is a common enough occurrence that the C-shell has a built-in variable called **ignoreeof**; when this variable is set, the C-shell responds to a Control-D by displaying a message telling you to use the **logout** command if you really want to logout.

3.1.2 The Shell Environment

Recall from the discussion of the kernel that each process inherits a process environment from its parent. This environment consists of such things as the process's base priority, its current directory, and file descriptors. The process environment is maintained by the kernel and can only be changed by system calls. When a process **exec**s a program, the program is passed a second environment, an environment derived from the shell. This **shell environment** consists of a number of user-related variables such as: login-name, shell name, home directory, command search path (explained shortly), terminal type, and **termcap** entry. These variables constitute a sort of "global argument list" that the user wishes to be made available to all commands he or she runs. C-Shell environment variables can be changed with its **setenv** command.

3.1.3 Commands

To make the shell do something, you type a **command**. For example, if you type

```
ls
```

the shell lists the contents of the current directory. If you type

```
cp oldfile newfile
```

the shell copies a file. Three types of commands can usefully be distinguished.

1. Commands the shell *must* implement itself. These include commands that alter the shell environment, that control the execution of multiple jobs, and that maintain and modify the command history buffer (jobs and command history are explained later in this chapter).
2. Commands that are implemented as utility programs distinct from the shell; the majority of commands fall into this class. When you type the name of one of these utilities, the shell executes it by **fork**ing a child process and **exec**ing the program — in other words, in exactly the same way any UNIX program executes another in the manner of a subroutine. Thus, any program may execute a utility, and any program may be executed by the shell as a command.
3. Commands that *could* be implemented as utilities, but are in fact built into the shell; there are only a few of these. Built-in commands are short and very frequently executed. Implementing them in the shell essentially saves the overhead of creating another process, yielding a noticeable performance improvement with the loss of some generality. In fact, the loss is not so great because most commands built into the shell for performance reasons are also implemented as utilities, making them accessible to other programs.

When you type a command, you do not know or need to know how

the command is implemented. The point of the preceding discussion is to show, that with a few necessary and pragmatic exceptions, shell commands are just ordinary programs with no special status. This means that new commands can added at will and that local versions of commands can replace those provided with the system, if necessary.

3.1.4 Command Execution

As ordinary programs,[5] commands have ordinary file names and may reside in any directory, although administration is simplified if they are gathered in just a few. You can enter a command by typing the command's full path name. For example, to check a text file for spelling errors you can type:

```
/usr/bin/spell file.txt
```

(**bin** conventionally stands for "binary" and often denotes a directory containing object modules.) Typing path names, however, is a nuisance, so the shell allows you to type just the command's file name, as if all commands were in your current directory. That is, to check the spelling in a file, you can just type:

```
spell file.txt
```

How, then does the shell find the file containing the command? It could start its search at the top of the file system tree but performance would be unacceptably slow. Alternatively, predefining a directory in which all commands were kept would place a somewhat arbitrary restriction on users. Therefore, the shell searches the directories listed in its **path** built-in variable. Every user can set this variable as he or she pleases, although most of the utilities that come with the system are located in two or three directories.

To further speed performance, the C-Shell maintains a hash table in memory that maps the short command names into absolute path names. It builds this table when it starts up by scanning the directories listed in the **path** variable for executable files. Given that it has a way to find the program that implements a command, the shell executes it just as any program executes another: it **fork**s a child process that in turn **exec**s the program (after looking it up in the hash table). Figure 3.2 shows, in pseudocode, a simplified version of the C-Shell's "main loop."

Of course, this represents the simplest case. However, subsequent sections of this chapter will show how the design of the kernel makes it simple for the shell to implement mechanisms like background processing and I/O redirection.

5. This section only considers the execution of commands implemented as utilities — the common case.

```
loop;
        display(prompt);
        get_next_command(cmd, arguments);
        process_id := fork();       (* fork a child to run the command *)
        if process_id = 0
        then
                (* child *) exec(lookup(cmd, hash_table), arguments);
                            exit();
        else
                (* parent *) process_id := wait(status);
        endif;
        (* parent--child has terminated *)
        if status /= 0                  (* if status not equal to 0 *)
        then display(error_msg(status));
endloop;
```

Fig. 3.2 *C-Shell Command Execution Logic*

3.2 Interacting with the C-Shell

This section describes the C-Shell's command interpreting facilities. The basics of issuing commands from a terminal or workstation, applying commands to groups of similarly named files, and giving names to frequently issued command lines are covered first. Described next are the two facilities, called history and job control, that are the principal advantages of the C-Shell over the Bourne shell for interactive work. I/O redirection and pipelines, which are essentially kernel services for which the shell provides access and a convenient notation, round out the discussion.

3.2.1 Command Syntax

Because commands reside in ordinary files and are executed by typing their file names, they are easy to add to the UNIX system. Over the years many different people have done so. Since the shell imposes almost no constraints on command syntax, some inconsistency between commands is inevitable. However, most commands follow the conventions described below.

Two kinds of arguments typically follow a command name, a list of **options** (sometimes called **flags** or **switches**) and a list of **file names**. The option list comes first; it is separated from the command name by a blank, and it begins with a hyphen ("-") character (the option character is commonly pronounced "minus" or "dash"). To illustrate, both of the following commands copy a file called **minutes.cur** to another file called **minutes.old**:

```
cp minutes.cur minutes.old
cp -i minutes.cur minutes.old
```

Many commands, including **cp**, write files; by default, most such commands create files that do not exist and write over files that do exist. These default actions are a good example of the pervasive UNIX notion that you know what you are doing when you enter a command. The **-i** option in the second version above specifies "*i*nteractive execution." In response to this option, the command

will ask for confirmation before writing over **minutes.old** if it exists.

The minus tells the **cp** command that an option list is present; otherwise it would interpret the "i" in the above example as a file name. (Notice that creating files whose names begin with '-' and other special characters, while legal, complicates life unnecessarily.) For some commands, multiple options are elided together after the minus; for example, the following command copies a group of files with the "*i*nteractive" and "*r*ecursive" options:

```
cp -ir suntech suntech.backup
```

This form of the **cp** command recursively copies all files in the **suntech** directory, and any subdirectories, to the **suntech.backup** directory, interactively asking for confirmation before overwriting any existing file. Other commands want multiple options separated by blanks, each preceded by a minus; for example,

```
diff -l -r oldversions newversions
```

finds the differences in the files in two directories, *r*ecursively descending any subdirectories they may contain, and generating the "*l*ong" output format (that is, producing more detailed information than the default "short" format).

Multiple commands may be strung together on a single line by separating them with semicolons; the shell executes the commands serially, displaying its prompt when the last command terminates. The following command line changes to a new working directory and lists its contents:

```
cd bookreviews; ls
```

3.2.2 File Name Shorthand

As mentioned in the last section, many commands take a list of file names. For example, to delete ("remove" in UNIX system vernacular) several sections of a manuscript, you could type:

```
rm sec1.txt sec2.txt sec3.txt
```

Such repetitive typing, however, is both tedious and conducive to error. It can be avoided using the shell's shorthand facility for applying a command to a *set* of files. The shorthand notation for a set of files is a pattern that zero or more file names may match. The pattern is constructed of ordinary characters and special characters equivalent to the "wildcards" found in many operating systems. There are a wealth of such special characters, but two of the most frequently used are "**?**" and "*". **?** matches any single character and * matches any string of characters, including the null string. When you type a shorthand file name, the shell finds all matching files (this is sometimes called "file name expansion") and passes their names to the command as if you had typed them all in full. Suppose, for example, your current directory contains three files, **sec1.txt**, **sec2.txt**, and **sec3.txt**. The following com-

mands then are equivalent:

```
rm sec1.txt sec2.txt sec3.txt
rm sec?.*
```

Note that if other file names in the current directory match the pattern (for example, **sec4.txt** or **sect.mss**), the second command will remove them as well.

Characters like **?** and *****, which have a special meaning to the shell, are called **metacharacters**. Sometimes you may want the shell to interpret one or more of these as ordinary characters with no special meaning (for example, when a file name contains a special character). If a metacharacter is preceded by the backslash (\) metacharacter, the shell will not intrepret the character specially. If a string of characters is surrounded by single-quote (') metacharacters, the shell will interpret everything inside the quotes as ordinary characters. Thus either of the following commands will delete a file whose name, for some reason, is **sec?.***:

```
rm sec\?.\*
rm 'sec?.*'
```

While the shell's file name shorthand facility is a great timesaver, it is potentially dangerous. Specifying a pattern that is more general than you intend can have surprising effects; these may be amusing or disastrous (the shell has no "undo" command). For example, as mentioned earlier, **rm sec?.*** will remove files named **sec4.txt** and **sec1.backup**, that is, any file in the working directory whose name matches the pattern. The **echo** command can be issued to "preview" the shell's expansion of a shorthand file name. **echo** is the UNIX system's simplest command: it merely displays its arguments. The shell passes all the file names it creates from your shorthand pattern to **echo** and **echo** writes them back to the terminal, having done no harm. By the way, the fact that file name expansion is performed by the shell rather than by individual commands, is an interesting point. The file name shorthand facility is localized in a single program, and yet applies to all programs — present and future — run via the shell with no effort on the part of their programmers or their users.

3.2.3 Command Aliases

The C-Shell has a simple string substitution facility that provides another kind of shorthand. Called **aliasing**, it can also be used to customize commands and to create what are effectively new ones.

Suppose you frequently update and print a table of sales offices. To do so requires a rather long command that is a nuisance to enter on a regular basis. The following **alias** command (typically issued in your **.cshrc** file):

```
alias ptab 'nice tbl salestab.ms | nice troff -ms -t | lpr -t -P1 &'
```

allows you to print the table by simply typing:

```
ptab
```

(The elements of this command are explained in later sections of this report. The important thing to note is that aliases are a convenient shorthand for long strings typed to the shell.)

Aliases are quite general; the shell will substitute any string for a command. Suppose you always use the **-l** option of the **ls** command. (This option lists the contents of a directory in "*l*ong" form, providing such information as permissions, owner, and size.) The following alias effectively redefines the **ls** command to your preferred mode of use:

```
alias ls 'ls -l'
```

An alias can substitute a command line argument for a variable in a substitution string. The earlier table printing example can be generalized to print any table by changing the **ptab** alias to take a file name argument. Here is how it is done:

```
alias ptab 'nice tbl \!* | nice troff -ms -t | lpr -t -P1 &'
```

The characters **\!*** are a "placeholder" for the argument you type after **ptab**. Thus, typing **ptab salestab.txt** produces the same result as the previous example in which the file name was hardwired into the alias.

Aliases are handy for making simple string substitutions in command lines. Real command *programming*, though, is done with shell scripts, the subject of the last part of this chapter.

3.2.4 Command History

One of the C-Shell's built-in variables is called **history**; the command:

```
set history=10
```

directs the shell to retain your last ten commands in a buffer. This command history mechanism is very useful and is one of the principal reasons why many people prefer the C-Shell to the Bourne shell for interactive work. Using the history mechanism, with very few keystrokes you can: repeat a previous command, obtain an argument from a previous command, or correct a typo in a previous command and reissue it. Note that the facilities offered by the history mechanism are quite extensive; only the basics are outlined here.

A previously issued command in the history buffer can be referred to by its number. The shell assigns these sequence numbers automatically; they can be displayed in either of two ways. The **history** command lists the contents of the history buffer in this format:

```
1 cd suntech/unix
2 ls -l
3 cd programs
4 cc testprog.c
5 history
```

You can also have the current command number displayed as part of the shell's prompt, allowing you to see recent command numbers that are still on the screen. The shell's prompt string is the value of the built-in variable **prompt**. By default, this value is '*hostname*%', for the Sun C-Shell, where *hostname* is the name of the machine on which the shell is running. To see how this works, suppose that your machine is called **sunstar**, the last command issued was number 94, and your **.cshrc** or **.login** file contains:

```
set prompt 'sunstar \!%'
```

then the shell will prompt for the next command with:

```
sunstar 95%
```

Thus, the exclamation point (many UNIX system users pronounce this character "bang") in the prompt string directs the shell to insert the current command number in its place. The exclamation point must be "quoted" with a preceding backslash to prevent the shell from interpreting it as a history command when it executes the commands in the file on startup.

To re-execute, say, command number 66 (assuming it is still in the history buffer), you just type **!66**. Sometimes you won't know a command's number (it may have rolled off the screen), but you will remember its name. In such a case you can re-run the command by typing !*string*, where *string* contains enough characters of the command name to identify it. The shell searches backward through the buffer and executes the first matching command, that is, the one most recently issued. Using the example history buffer shown above, typing **!c** would execute command number 4; typing **!cd** would execute command number 3. Command number 1 cannot be re-executed by name since it has the same name as the more recently issued command number 3.

A previously issued command can also be re-executed with modifications (for example, to correct a typo, to apply the command to a different file, or to apply a different command to a previously typed file). Suppose, for example, that you just misspelled the word "programs" in the following command:

```
cd suntech/unix/pograms
```

It can be fixed up and re-executed by simply typing

```
^po^pro^
```

This means "in the previous command, change the string '**po**' to '**pro**' and then execute the resulting command." Here is a longer

form of the same kind of "re-execute with substitution" command that is useful when the target command is not the most-recently issued:

```
!cd:s/po/pro/
```

The shell interprets this command as "find the most recent command beginning with the characters **cd**, substitute the characters '**pro**' for '**po**', and execute the result." The slashes used as substitution string delimiters in this example can actually be almost any character, a handy feature when the target string contains slashes (for example, it is a path name).

The shell thinks of a command line as comprised of "words" separated by spaces (or a few special characters such as semicolons). Each word on a line has an implied number, the first word being number 0. Some of these words may be long path names; retyping such a name is a nuisance and an invitation to err. The C-Shell permits a word or sequence of words from a previous command to be effectively copied from the history buffer into the current command. To see how this works, suppose the history buffer looks like this:

```
65 cd /usr/terry/project/newindow/notes
66 ls -l
67 cd
```

Typing

```
cp !65:1/apr22meeting tempnote
```

picks up word 1 from command 65, yielding the following command:

```
cp /usr/terry/project/newindow/notes/apr22meeting tempnote
```

To pick up a series of words from a previous command, the same general notation is used, with the starting and ending word numbers separated by a hyphen as in: **cp !68:1-2**.

3.2.5 Job Control

The C-Shell considers each separately entered command, sequence of commands (separated by semicolons), and pipeline[6] to be a **job**. Users can freely create multiple jobs, move them between foreground and background, suspend them, and resume them. In short, job control gives users access to the multiprocess capabilities of the kernel. Along with command history, job control is one of the principal advantages of the C-Shell over the Bourne shell for interactive use.

6. Pipelines are an extremely useful means of connecting simple commands to perform complex operations. They are described in detail shortly.

The simplest job control command runs a job in the background. Consider, for example, the compilation of a large program which may run for a long time. Rather than wait for such a command to terminate, you can direct the C-shell to run it as a background job as shown in the following example:

```
f77 anewprogram.f &
```

Here, **f77** is the name of the FORTRAN 77 compiler, **anewprogram.f** is the source file, and **&** is the metacharacter that tells the shell to run the job in the background. After starting the job, the shell reports the **job id** and the **process id** of each process it **fork**s to start the job, and then issues a prompt. You can then enter more commands while the job runs along in parallel in the background. When the job terminates, the shell notifies you with a short message.[7]

You can find out what your jobs are doing with the **jobs** command. The following is typical of its output:

```
[1]    Running                cc part1.c
[2]  - Stopped                cc part2.c
[3]  + Stopped                 mail
```

Job ids are listed first. The **current job** is marked with a '+' and the **previous job** is identified with a '-'; these are useful for shorthand versions of job control commands that are not described here. The job's status and the command line that invoked it are also shown. To refer to a job in a command, you can use a percent sign followed by the job's id or characters from the command line that invoked it in a fashion that is analogous to the command history facility. For example, job 3 above can be identified by either **%3** or **%mail**.

The job control commands and a few related commands and special keys are shown in Table 3.1. Note that you can change the mapping of special keys to functions with the **stty** command. Allowing several jobs to read from and write to your workstation or terminal concurrently could be confusing. Therefore a background job is automatically suspended when it tries to read input from the keyboard. At your convenience you can bring it to the foreground, give it the input it wants, and then return it to the background. If you want to see a background command's output on the terminal, but do not want it interspersed with the output of foreground jobs, you can issue the following command:

7. Implementation note: Starting a job in the background is trivial for the shell. As usual, it **fork**s a child process, which in turn **exec**s the command. The difference is that the shell doesn't wait for its child to terminate — after displaying the new process's ID, it goes ahead and displays the next prompt. The kernel sends the shell a signal when the child terminates.

Table 3.1 *Job Control Commands*

Command	Function
`stop`	Suspend background job
`bg`	Move suspended job to background and restart
`fg`	Bring background or suspended job to foreground and resume
`kill`	Abort background job
`nice`	Run job at reduced base priority
`Control-Z`	Suspend foreground job and return to shell
`Control-C`	Abort foreground job
`Control-\`	Abort foreground job and dump memory to file for post-mortem debugging
`Control-S`	Suspend terminal output
`Control-Q`	Resume terminal output
`Control-O`	Discard terminal output

```
stty tostop
```

This command suspends a background job if it tries to write to the display. If you bring it to the foreground, it will be resumed and you can see its output.

The preceding techniques are satisfactory when background jobs do relatively little workstation I/O. The common practice when running jobs that do a lot of standard I/O is to redirect the I/O to files. This topic will be taken up shortly.

Here are a few examples of using job control commands.

- Suppose you erroneously start a long C compile in the foreground by forgetting to end the command line with **&**. Just type Control-Z to stop the job, then give the shell the **bg** command to put the compile in the background as you originally intended.
- Suppose that in the midst of a long edit you need to find the name of another file. Typing Control-Z suspends the edit job and returns you to the shell. Now you can type whatever commands you need to find the file name. When found, type **fg** and you are right back in the editor exactly where you left off.
- Suppose you want to run several background jobs while you continue to work in the foreground. As described in the kernel chapter, the scheduling algorithm will divide processor time more or less fairly among all the processes, favoring I/O-intensive processes, as your foreground work is likely to be. Nevertheless, if there are enough background proceses, foreground response time can deteriorate noticeably. "**nice**ing" the background processes will bias their base priorities so they get less favorable treatment from the scheduler, thereby improving foreground response. For example:

  ```
  nice f77 crunch.f &
  ```

3.2.6 I/O Redirection

In the absence of file name arguments, most commands read their input from the standard input and write their output to the standard output. Recall from the discussion of the kernel that one of the things a newly spawned process may do before **exec**ing a new program is to redirect the program's I/O. The shell provides a very convenient notation for redirecting a command's standard I/O before executing it.[8] Suppose, for example, you want to save the output of the **ls** command in a file. The following is all that is necessary:

```
ls -l > dirlist
```

As usual in the UNIX system, if the file **dirlist** exists it is overwritten. However, setting the C-Shell's **noclobber** built-in variable changes its default action so that it will not overwrite an existing file with redirected I/O unless explicitly directed to do so. To append the output to an existing file, the following notation is used:

```
ls -l >> dirlist
```

If the file does not exist, it is created. The standard error can be redirected to the same file as standard out by typing a command like this:

```
ls -l thisdir >& dirlist
```

To redirect the standard out and standard error to different files, use the following form:

```
(ls -l thisdir > dirlist) >& errlist
```

The shell can redirect a command's standard input from the keyboard to a file using a similar notation (the **wc** command counts words, lines and characters):

```
wc < longlist
```

I/O may be redirected from/to devices, as well as files, since for the most part the UNIX system does not distinguish between them. A particularly handy device (a pseudo-device, really) is **/dev/null**. Any output written to this device is simply thrown away; reading from **/dev/null** results in an immediate end-of-stream condition. Redirecting a program's standard output to **/dev/null** is a simple way to deal with commands that produce volumes of output that are irrelevant to the context in which the command is being invoked. For example, suppose you want to find out how long a

8. Implementation note: Before **exec**ing the program that implements the command, the child shell examines the arguments typed by the user. If they indicate I/O redirection, the child shell simply **close**s the relevant descriptor and **open**s the file or device specified in the command line. (Recall that **open** always returns the lowest-numbered available descriptor.) For example, if standard output is to be redirected to a file called **dirlist**, the child shell **close**s descriptor 1 and opens **dirlist**. Neither the parent shell (which has its own descriptors) nor the program subsequently executed have any idea that the redirection has taken place.

command takes to run. The **time** command reports the execution time of another command whose name is passed to it as an argument. For example:

```
time grep e longlist > /dev/null
```

will tell how long it takes to search a list for all occurrences of the letter "e", without bothering to display what may be thousands of lines on the screen.

3.2.7 Pipelines

Suppose you want to find the files in a large directory that have the suffix **.txt**, listing them in decreasing order of size. This is easy using the **ls**, **grep**, and **sort** commands as follows:

```
ls -l bigdir > temp1
grep '\.txt' temp1 > temp2
sort +3nr temp2
```

(The size of each file is the third field in the lines produced by **ls -l**. The **sort +3nr** command sorts the lines into *r*everse *n*umerical order based on this field.) At the heart of this example are the **temp1** and **temp2** files which hold intermediate results and should be removed after executing the three commands. Instead of having the processes communicate through temporary files, why not just connect them with pipes? The shell makes it easy:

```
ls -l bigdir | grep '\.txt' | sort +3nr
```

Besides saving typing (and remembering to remove the temporary files), such **pipelines** have two additional advantages. First, unlike files, there are no size constraints on pipes. Second, all the processes in the pipeline run in parallel. The result is less elapsed running time.[9]

Commands coupled in various ways with pipelines form an amazingly versatile toolkit. Suppose, for example, that you want to find out which users are consuming the most disk space. The following pipeline does the job:

```
du -s /usr/* | sort -nr | head
```

(This example assumes that all home directories are themselves located in the **/usr** directory.) The **du** command produces one line per user that summarizes that person's total disk usage. These lines are piped to the **sort** utility, which sorts them numerically in reverse order (so the largest consumers come out first). The sorted lines are piped to the **head** utility which, without arguments, just

9. Implementation note: To execute a pipeline the parent shell first creates the number of pipes entered as arguments on the command line. It then **fork**s a child for each command in the pipeline. Each child examines the arguments and uses the **dup** system call to replace a standard descriptor with a descriptor representing the read (for standard input) or write (for standard output) end of a pipe. Again, neither the parent shell nor the program to be executed is affected by the redirection. If the pipeline is executed in the foreground, the parent shell waits for the last process in the pipeline to terminate; if it is executed in the background, the shell does not wait but issues its next prompt.

displays the first ten lines. Quickly connecting simple commands like this to solve problems that arise everyday is very characteristic of the programming style that has come to be associated with the UNIX system. More examples will be found in the next chapter.

3.3 Programming the Bourne Shell

The first part of this chapter emphasized that UNIX system commands are, in the main, ordinary programs executed as processes by a shell; a shell is another ordinary program. It follows from this that a shell should be able to execute a different shell or another instance of itself as a command. Moreover, facilities such as I/O redirection should apply to shells executed as commands just as they do to other programs. In fact, all of this is true. Consider the following three commands:

```
sh
sh < makedoc
sh makedoc
```

Assuming you are typing these commands to the C-Shell, here is what happens. In the first case, the C-Shell **fork**s an instance of itself, passing the descriptors for your workstation or terminal, and waits. The child C-shell then **exec**s the Bourne shell (the Bourne shell's name is **sh**). The Bourne shell, running as a separate process, issues its prompt and executes the commands you type. When you type Control-D, the child process running the Bourne shell exits. This causes the parent C-Shell's **wait** system call to return, and the parent then writes another prompt on your workstation or terminal. In the second case, the sequence is identical except that the child C-Shell redirects its standard input to the file **makedoc** before **exec**ing the Bourne shell. The Bourne shell then reads and executes commands from this file, essentially unaware that it is not connnected to a keyboard (the shell does not issue prompts, however, when it executes commands from a file). When it reaches the end of the file, the process running the Bourne shell exits, and the C-Shell issues its next prompt as before. The final example shows that the shell, like most utilities, follows the convention of reading its input from a file (rather than the standard input) if one is specified as an argument. The result is identical to the second example; the only difference is that the child shell explicitly **open**s the file **makedoc** instead of reading the standard input.

Many operating system command interpreters can execute commands from a file. Two things distinguish the shell's capabilities in this area. First, reading commands from a file is not a special feature built into the shell, but a natural consequence of the mechanisms and conventions that apply to all UNIX programs: the unity of devices and files, and the notions of standard I/O and redirection. Second, the shell interprets more than a command language, it interprets a real programming language with variables,

input/output, flow control, and debugging facilities. This language is a remarkable tool for getting work done rapidly. The following items help explain why shell programs can be developed so quickly:

- The language is interpreted; you can test a fix immediately without waiting for compilation and linking.
- The C-shell's job control facility makes switching between program editing and program testing extremely fast; to test a change, for example, you just type Control-Z to put your editing session in the background and return you to the C-shell. After running the program, you resume the edit session by just typing **fg**.
- The UNIX system utilities described in the next chapter form what amounts to a library of very high level subroutines for shell programs. For that matter, shell programs can call other shell programs or programs written in conventional languages in exactly the same manner.

The rest of this chapter describes the most important elements of the shell programming language and shows these elements in action in a few simple programs.

3.3.1 Installing and Running Shell Scripts

A file used as input to a shell is variously known as a **shell script**, a **shell file**, a **shell program**, or a **shell procedure**. Suppose you have written, with any editor, a Bourne shell script called **multirep**; this program performs a "global replace" operation across multiple text files. You can execute this script from the C-Shell just as you would execute any command, that is, by typing:

```
multirep from-string to-string file-1 file-2 file-n
```

To do so, however, requires a small amount of preparation. First, the file containing the script must be placed in a directory that is in your C-Shell's search path. (Private programs, including shell programs, are often kept in a directory subordinate to a user's home directory called **bin**, but this is just a convention.) Second, the file must be made executable. This can be done by simply typing

```
chmod +x multirep
```

meaning "add execute permission to the file **multirep**." Third, the first line of the script must be the following:

```
#! /bin/sh
```

This special notation tells the kernel which shell is to interpret the rest of the script. *All* executable files in the 4.2 BSD UNIX system, whether shell scripts or object modules or **awk** programs (**awk** is a utility described in the next chapter), are similarly self-identifying. This means that a process wishing to execute a program in a child process can do so without concern for whether the program contains machine instructions, or must be interpreted by another program,

such as a shell. The kernel examines the file containing the program and invokes the appropriate interpreter if necessary.

Note the importance of shell programs being executed just like other commands. First, there is nothing new to learn to run a shell program. Second, shell programs can be used in pipelines, have their I/O redirected and so on, just like other commands. Third, a new command may first be implemented as a shell program, and later, when — and if — there is a demand for better performance, it can be reimplemented as a compiled program. No user of the shell program, whether a user or a program, will ever know that the change has occurred. As a matter of fact, some commands supplied with the UNIX system *are* implemented as shell programs.

3.3.2 Variables

All shell variables are of the same type, namely character string. To declare a variable, you assign it a value. To obtain the value of a variable, you mention its name preceded by a **$** character. For example:

```
FRUIT=apple
echo 'value of FRUIT is' $FRUIT
```

Note that blanks around the " = " assignment operator are illegal. A reference to a variable that has not been assigned a value returns a null string. Many shell programmers use upper case names for variables, as the example shows, but this is a convention, not an edict of the shell. Another way to give a variable a value is to assign it the output of a command. For example,

```
DIR='pwd'
```

assigns the name of the current directory to the variable **DIR**. Note that when assigning a variable the result of a command, the name of the command is surrrounded by *backquotes*, not apostrophes.

The shell predefines a number of variables whose values can be used in scripts. For example:

- Arguments typed on the command line that invoke a script are available in variables named **1** (the first argument), **2**, and so on; the variable **0** contains the name of the command that invoked the script.
- # contains the number of arguments.
- **HOME** contains the pathname of the invoker's home directory.
- **PATH** contains the list of directories that the shell searches for commands.
- **$** contains the id of the process that is interpreting the script; this value is commonly used to generate a unique file name.[10]

Even though all variables contain character string values, they

can be interpreted as numbers when they contain digit characters. This is done with the **expr** command. **expr** evaluates its arguments as an arithmetic expression and writes the result to the standard output. It supports the normal four arithmetic operations and the remainder operation. For example,

```
SUM='expr $SUM + 1'
```

(note the backquotes) increments the value of **SUM**. (This is another example of a variable taking its value from the output of a command.) **expr** also provides comparison, substring and regular expression matching operations; regular expressions are explained in the next chapter.

Now for an example. Suppose you regularly write letters that go into window envelopes. After some tinkering you develop a skeleton file that has the address positioned so it will show through the window when the letter is properly folded. To make a new letter you could always edit the skeleton file directly, but it is safer to make it read-only and edit a copy of it. The script shown in Figure 3.3 does the job. Note that the example shows line numbers to simplify its explanation; line numbers are illegal in actual shell scripts.

Fig. 3.3 *Form Letter Shell Script*

```
1  #! /bin/sh
2  # Make a letter for a window envelope
3  cd $HOME/letters
4  NAME="$1.`getdate`"
5  cp window.form $NAME
6  chmod u+w $NAME
7  vi +/Date $NAME
8  lpr -Plaser $NAME
```

If the script is stored in a file called **window**, you can produce a letter to the Internal Revenue Service by typing:

```
window IRS
```

Line 1, as mentioned earlier, tells the kernel to interpret this script with the Bourne shell. The "#" character in the first column of line 2 marks a comment line (a few implementations of the Bourne shell

10. The UNIX system recycles process ids when it starts up. Because of this a file name created with the **$** builtin variable is not guaranteed to be unique. Consider a shell program that creates a file in the current directory using the **$** variable as follows:

    ```
    mv $1 alpha.$$
    ```

    Supposing the id of the process interpreting the shell program is 32778, this command effectively renames the file passed in the first argument to **alpa.32778**. It is possible that before the system was last started up, this program (or another) ran with the same process id and created the same file; this file may still exist. To simpify the creation temporary files, UNIX systems have a directory called **/tmp** that is intended for temporary files. The file name **/tmp/alpha.$$** is guaranteed to be unique since the system automatically removes the files in **/tmp** when it starts up.

define comments as beginning with a colon). Line 3 changes the current directory to the **letters** subdirectory of your home directory ("hard wiring" the assumption that such a directory exists and is where you want to keep copies of your letters would be a poor practice in a script written for public use, but it simplifies this example). Line 4 sets the value of the variable **NAME** to a file name created from the first comand argument (IRS, in this case) concatenated with the output of a program called **getdate**. **getdate** is a 12-line C program, not explained here, that picks up the system date, formats it, and writes it to the standard output. Thus, programs that you write can be invoked in shell scripts just like system-supplied utilities. Line 5 copies the skeleton letter to the file name contained in **NAME**. Line 6 adds write permission for the user (owner) of the new file. In line 7, the script invokes the **vi** (pronounced "vee-eye" and described in the next chapter) screen editor on the file containing the letter, positioning the cursor at the "Date" field in the skeleton. Now you type in the date, salutation and body of the letter, using the strings you placed in the skeleton (for example, "Addressee Line 1") as guides; when you exit **vi**, the completed letter is saved and line 8 of the script spools it to a printer called **laser**.

3.3.3 Input/Output

When it is invoked, the shell process interpreting a script normally has its standard input, output, and error descriptors connected to the workstation or terminal. The script can redirect the I/O of commands that it issues to files and pipes. In general, it is good practice to redirect diagnostic messages to descriptor 2 (standard error) in case the script has been invoked with the standard output redirected to a pipe. (Diagnostic messages that disappear down a pipe are of little use.) An example of this follows:

```
echo 'File $FILE does not exist' >&2
```

A script can conduct a dialog with its invoking user by writing lines with the **echo** command and reading lines with the **read** command.

3.3.4 Flow Control

The window letter example shown in Figure 3.3 was just a list of commands executed in straight line fashion. Shell programming becomes more interesting when scripts make use of the shell's flow control constructs. The shell provides an **if/then/else** construct for conditional branching and a **case** statement for multiway branching. The **if** construct is terminated by a **fi** statement while the end of a **case** is delimited by an **esac** statement. The condition tested by an **if** statement is usually implemented with the **test** command, described shortly. However, any command may be used. The shell considers a command that exits with status of 0 (that is, succeeds) to have returned a value of "true" and a command that exits with a nonzero status to have returned a value of "false." The sense of an **if**-test is: "if the following command returns true, execute the commands up to the

else statement (or the **fi** statement if no **else** statement is present); otherwise, execute the commands following the **else** statement if one is present."

The **test** command operates on files, variables, and literal character strings. You can use it to find out if a file exists, is readable, is writable, is longer than zero bytes, is or is not a directory, and so on. You can test strings for equality, zero length, null value, etc. Test conditions can be combined with and/or/not operators and grouped with parentheses.

Another name for the **test** command is **[**, that is, the left bracket character. The **test** command itself ignores an argument that is the right bracket character. In combination, these make it possible to test a variable called, say **NFILES**, for a value less than three in either of the following ways.

```
if test $NFILES -lt 3
if [ $NFILES -lt 3
```

The second form is more compact and more resembles other programming languages, so it is commonly used.

Shell script loops can be controlled in three ways:

1. A **while-do** tests a condition (usually expressed by a **test** command) at the top of the loop. If true, the statements up to the loop-terminating **done** statement are executed.
2. An **until-do** is identical except that the test is performed at the bottom of the loop.
3. A **for-do** tests a condition at the top of the loop, and if true, sets a variable to a new value and executes the statements in the loop body.

The **exit** statement terminates a shell program, optionally passing a nonzero return code.

To see some simple flow control statements in action, refer to Figure 3.4 which shows an implementation of the **multirep** script described earlier. Typing:

```
multirep alpha beta chapter1.txt chapter1.art
```

will change all occurrences of the string "alpha" to "beta" in the files **chapter1.txt**, and **chapter1.art**. The example also illustrates some simple parameter checking and I/O redirection. As in the previous example, the script lines are numbered for reference, an illegal practice in actual scripts.

Line 5 shows a very common use of the **test** (**[**) command. **multirep** must be called with at least from- and to-string arguments and one filename; line 5 checks that the built-in variable # has a value of at least 3, that is, that the script was invoked with at least 3 arguments. If so, execution continues at line 10. If not, the script writes a diagnostic message to the standard error (recall that the built-in variable $0 holds the name by which the script was

```
 1  #! /bin/sh
 2  # Global replace over multiple files.
 3  #
 4  # If fewer than 3 arguments, quit.
 5  if      [ $# -lt 3 ]
 6  then
 7          echo "Usage $0: from-string to-string files" >&2
 8          exit 1
 9  fi
10  #
11  # Pick up from-string and to-string arguments.
12  FROM=$1; shift
13  TO=$1; shift
14  #
15  # Do replacement for each remaining file argument.
16  until [ $# -eq 0 ]
17  do
18  #       Make sure target file exists.
19          if      [ ! -r $1 ]
20          then    echo "no file $1" >&2; shift
21
22  #       Do replacement with stream editor,
23  #       writing result to temporary file.
24          else    sed -e "s/$FROM/$TO/g" $1 > /tmp/$0$$
25  #               Replace orginal file with temporary.
26                  mv /tmp/$0$$ $1;
27                  echo "changed file $1" >&2
28                  shift
29          fi
30  done
```

Fig. 3.4 *Global Replace Shell Script*

invoked), and exits. In lines 12 and 13, the variables called **FROM** and **TO** receive their values from the first two command line arguments. The **shift** statement, not previously discussed, moves the value of the variable 2 to 1, 3 to 2, and so on, and is very useful for processing an unknown number of arguments, as will be shown. The program's "main loop" begins on line 16, where it checks to see if another file remains to be processed. The loop stops when **$#** is reduced to zero as a result of **shift** statements in the loop body. Line 19 tests for the existence of the current file in the argument list (the file whose name is represented by the **1** built-in variable). The **-r** option of **test** means "exists and is readable," and the exclamation point negates the condition, returning true if the file does not exist or is not readable. Presented with a nonexistent (or non-readable) file, the program writes a diagnostic and goes on to the next file argument.

The heart of the program is line 24. **sed** is a stream-oriented (that is, noninteractive) editor, which is described in the next chapter. Here it is told to globally substitute the string $TO for the string $FROM in the file named $1. **sed** does not alter the original file, but writes the updated version to the standard output. The script redirects standard out to a temporary file that takes its name from the command used to invoke the script and its process id. This is likely to be a unique name, and is an easy name to find if necessary when debugging the script. The rest is simple. The original file is overwritten with the temporary file, a confirming message is displayed, and the arguments are shifted to prepare to process the next file.

3.3.5 Debugging

Shell scripts tend to be easy to debug because they are largely composed of commands that are themselves debugged and whose operation you already understand. Shell scripts are also typically rather short compared to their function because the commands are often far more powerful than typical programming language statements.

The shell gives reasonably good syntax error diagnostics as it interprets a script, but it also has debugging options that can be invoked with the **set** command. **set -v** causes a trace of each statement as it is executed, while **set -x** traces statements and prints variable values as well. These options can be turned off with the **set -** statement. A script can be traced without modifying it by invoking it as follows (assuming the name of the shell program is **script**):

```
sh -x script
```

Of course, inserting **echo** commands at strategic points in the code provides another way to selectively display what is going on during execution.

4 *The Utilities*

On the order of 220 utilities are supplied as part of the Sun UNIX system. This chapter briefly describes those that are most widely used. The level of detail varies considerably according to both the importance of the utility and the expected familiarity of its function. For example, the C language is covered more extensively than FORTRAN.

For purposes of discussion, the utilities are divided into six classes:

1. basics
2. editors
3. programming tools
4. filters
5. formatters
6. communication

At the end of the chapter are some examples showing how utilities may be used together to solve problems they do not address individually. Their *cooperative* nature is probably what most distinguishes UNIX system utilities from their counterparts in other operating systems. By using the utilities as an extremely powerful subroutine library, short, simple shell programs can very often be substituted for time-consuming, error-prone conventional programming.

4.1 The Basics

Table 4.1 summarizes the functions provided by the most basic UNIX system utilities. Note that a few of these are built into the shell and therefore can only be invoked from it. The great majority of them, however, and all the utilities described in the rest of the chapter, are implemented as ordinary programs that any process can execute as a child process.

Table 4.1 *Basic Shell Commands*

Command	Function
pwd	Print working directory
cd	Change working directory*
pushd	Push directory on stack and change to new working directory*
popd	Pop directory from stack, changing to new working directory*
dirs	Display directories on stack*
ls	List directory contents
mkdir	Make new directory
rmdir	Remove directory
mv	Move (rename) file
cp	Copy file
rm	Remove file
cat	Concatenate streams
tee	Copy standard input to standard output and one or more files
more	Display stream in screenfuls
head	Display first lines of stream
tail	Display last lines of stream
od	Display stream in octal, hexadecimal, etc.
wc	Count words, lines, characters in stream
cmp	Compare streams
diff	Display differences between streams
uniq	Display unique lines in a stream
lpr	Spool stream to printer
chmod	Change file access mode (permissions)
find	Search for files meeting criteria
echo	Display arguments
at	Execute shell script at specified time
test	Evaluate expression

* Built into shell, not executable as a program

4.2 Editors

4.2.1 ed

ed is a line-oriented editor notable for its very terse user interface.[1] Because there are much better alternatives, **ed** is rarely used on Sun workstations. However, **ed** *is* universally available on UNIX systems, and so provides a common denominator. Moreover, it is historically significant. Other more advanced editors, which are discussed in this section, are derived from **ed**. In addition, the UNIX system's notion of a **regular expression** originated in **ed**; regular expressions are prominent not only in UNIX system editors but in the frequently used class of utilities known as filters.

Regular expressions are similar to the file name shorthand facility provided by the shell. A regular expression is a string of characters that specifies a pattern that may be used to find matching character strings. For example, 'chapter [1-4]$' is a regular expression that would match all the following strings: 'chapter 1', 'chapter 2', 'chapter 3', and 'chapter 4' with the added constraint

1. For example, **ed** has exactly one error message, namely "?". Such terseness, which seems excessive today, made more sense when **ed** was developed — the teletypewriter was the dominant terminal.

(specified by the '$') that the string falls at the end of a line. Thus, a regular expression is a convenient notation for indicating that an operation is to be applied to a whole *class* of character strings. In an editor, a regular expression provides, for example, a quick way to change all instances of a particular class of string (for example, changing chapters 1-4 to sections 1-4, without changing chapters 5-10).

4.2.2 ex And vi

To remedy the weaknesses of **ed** and exploit the increasing availability of intelligent display terminals, Berkeley created two editors that have gained wide acceptance in the UNIX community. Like **ed**, **ex** is a line-oriented editor. It is a major enhancement of **ed** that preserves **ed**'s commands, making it easy for experienced **ed** users to become **ex** users. Besides having more capabilities, **ex** is substantially easier to use than **ed**.

vi, pronounced "vee-eye," is a screen-oriented ("*vi*sual") editor. It is fast, very powerful and somewhat difficult to master. In practice, most Sun users edit with **vi**, reserving **ex** primarily for use on 300 baud dial-up lines (**vi** works fairly well at 1200 baud). There are, however, some notable exceptions to this dichotomy, exceptions that are made possible by **vi**'s ability to interpret **ex** commands.[2] In particular, block moves and global substitutions are easier to do with **ex** commands than **vi** commands.

Functionally, **vi** is similar to other powerful screen editors so its more pedestrian features are not described here. Here are some of its more unusual facilities:

- The cursor can be moved by words, lines, sentences, paragraphs, parentheses, brackets, and braces; these objects can also be replaced or deleted as units.
- The previous command can be "undone" and the previous nine deletions can be recovered.
- The edit buffer is periodically saved in a file that can be recovered in case of a system crash, a dropped telephone line, or other interruption. The recovered file is normally within a few commands of completely up-to-date.
- Multiple buffers are available for holding temporary information.
- Search and replace operations may incorporate regular expressions; they may be case-sensitive or not, as desired.
- Automatic word wrap at the right margin and autoindent at the left margin are both available.
- A single command can be passed to a shell at any time; it is also possible to escape to a shell for an extended period and return to **vi** with the editor in the same state.[3]

2. In fact, **vi** and **ex** are a single program, but most people think of them as two.

- **vi** provides modest assistance for LISP, C, and **troff** (discussed later in this chapter) source files. It can check for balanced parentheses, brackets and braces.
- Terminal keys can be mapped to arbitrary character strings, allowing, for example, function keys to invoke long command strings (rare in **vi**) or insert frequently used text strings.
- Filters, such as **sort** and **uniq**, can be applied to the contents of the edit buffer.

4.3 Programming Tools

The UNIX system comes with standard software development aids like compilers, an assembler, a linker, and debuggers. Other tools are more unusual. These include utilities for *managing* software, for monitoring program performance, and even for building new translators. But before describing these, an overview of the program translation sequence is in order.

4.3.1 Program Translation

Figure 4.1 shows how a source file flows through the utilities that transform it to an executable program. The translation steps are usually invisible to the programmer since a single command can invoke the entire sequence of utilities. For example, typing

```
cc driver.c
```

passes a C language source file called **driver.c** through the usual stages, producing an executable object program. Adding the **-O** option to this command line invokes the optimizer step.

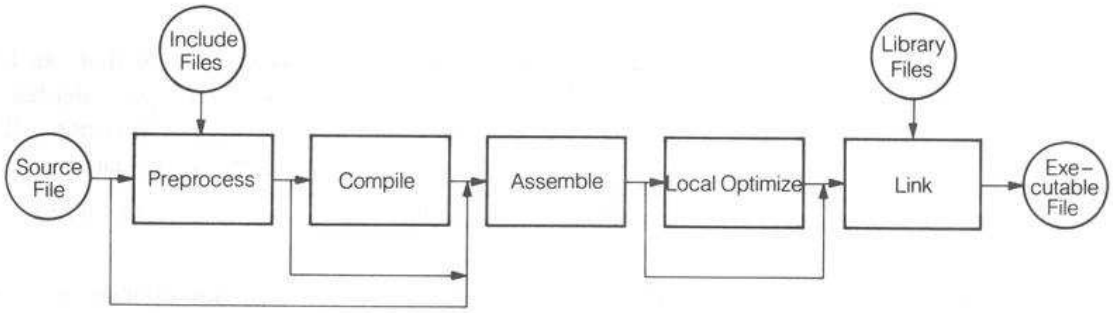


Fig. 4.1 *Program Translation Stages*

4.3.1.1 Preprocessor

The C compiler automatically invokes the preprocessor, while the FORTRAN and Pascal compilers invoke it if told to do so with a command line option. Assembly language programmers can run it as a separate program (it is called **cpp**), passing its output to the assembler. The preprocessor is independent of any language and can

3. These shells are children **forked** by **vi**. To temporarily escape to the shell that **forked** **vi**, just type Control-Z (that is, use the C-Shell's job control facility.)

be used to define symbolic constants, insert other files into the source stream, expand macros, and conditionally compile segments of code.

The preprocessor looks for lines beginning with #; it writes all other lines directly to the standard output. Figure 4.2 shows some typical preprocessor directives, interspersed with descriptive comments that do not correspond to any particular language.

```
(* define some symbolic constants *)
(* Note that uppercase names are a convention only *)
#define TRUE 1
#define CLEAR_SCREEN "\014"
#define MAX 1200

(* use a symbolic constant to declare the size
of an integer array in C *)
int table[MAX];

(* a #include directive is replaced by the file it names *)
(* note that an included file may contain #includes *)
#include "declarations.c"

(* define a macro to square a number in C *)
#define sqr(x) ((x)*(x))
(* invoke the macro *)
b = sqr(diagonal+2);

(* conditionally compile one C statement or
another, depending on whether the symbolic
constant CLEAR_SCREEN has been defined *)
#ifdef CLEAR_SCREEN
        putchar(CLEAR_SCREEN);
#else
        clearscreen();
#endif
```

Fig. 4.2 *Preprocessor Examples*

Although **cpp** is the most widely used preprocessor, another, called **m4**, is also available on UNIX systems. **m4** is a powerful, general-purpose macro processor that includes capabilities for macro arguments, conditional macro expansion, arithmetic, file manipulation and string processing.

4.3.1.2 Compilers

Three compilers are supplied with the Sun workstation; these translate the FORTRAN 77, Pascal and C languages. The compilers have a good deal in common:

- They rely on the preprocessor for macro facilities.
- They share a common code generator; assembly language output can be obtained, if desired.
- They observe a common set of run-time conventions, so routines written in different languages can call each other.

- They are compatible with a single set of debuggers and execution profilers.

The compilers also take a common approach to the code they generate for floating point arithmetic. A particular Sun may or may not have floating point hardware; the compilers address this dichotomy as follows. By default, the compilers produce code that runs on either configuration, automatically using the floating point hardware if it is present. Alternatively, a compiler can be directed, via a command line option, to assume that floating point hardware is present; the resulting code runs somewhat faster than the code generated by default. However, such a program terminates abnormally if run on a system that does not have the supporting hardware.[4] Thus, users can make the tradeoff between portability and performance to fit a given situation.

4.3.1.3 Assembler

Assembly language is sometimes used in UNIX systems to implement critical bottleneck routines. (It is also used in the kernel in the rare cases where particular machine instructions must be issued, for example, to disable interrupts.) The Sun UNIX assembler generates MC68010 machine instructions. It is a full-featured assembler, except that it has no built-in macro or conditional assembly facility, relying instead on the capabilities of the preprocessor.

4.3.1.4 Local Optimizer

Invoked by a compiler option, the local optimizer performs a "peephole optimization" of short segments of compiler-generated assembly language. It basically eliminates unnecessary instructions and replaces sequences of instructions with instructions that exploit the MC68010 more effectively. As an example of the former, consider the following (pseudocode):

```
a := b;
c := a;
```

A compiler might generate a store from a register into **a** for the first statement followed by a load from **a** for the second. The optimizer can eliminate the load by noticing that the value of **a** is still loaded.

The local optimizer also rearranges "if" statements, removing unnecessary branches.

4.3.1.5 Linker

The linker does what linkers (sometimes called linkage editors or binders or loaders) normally do: it combines separately compiled files, resolves intermodule references, and searches libraries for unresolved references (typically language run-time support routines). A linked file may be subsequently linked with other object files.

4. The mechanism works as follows. The compiler inserts instructions into the program that probe for the presence of the floating point hardware when the program starts. Floating point operations are implemented as subroutine calls that, depending on the result of the earlier probe, either use the hardware or perform the computation in software. When the compiler is told to assume the hardware is present, it emits inline code that uses the hardware to perform computations, eliminating the overhead of the initial probe and the subroutine calls. This code will fail, however, if the expected hardware is not present.

4.3.2 Languages

4.3.2.1 FORTRAN 77

The FORTRAN 77 compiler is called **f77**; it fits cleanly into the UNIX system environment. Most library routines callable from C are callable from FORTRAN as well, so FORTRAN programs can create processes, traverse directories and so on. FORTRAN units 5, 6 and 0 are by default connected to the standard input, output and error streams, respectively, and a FORTRAN program's standard I/O may be redirected via the shell as usual.

The following summarizes those aspects of **f77** that may differ from other FORTRAN 77 implementations.

Options **f77** compiler options include run-time checking of array subscripts, flagging of undeclared variables, and identification of constructs that are not compatible with FORTRAN 66.

Violations The implementation violates the ANSI X.63 standard in eight instances, none of which is serious to the vast majority of programs.

Extensions Some of the more important extensions (which, if used, make a program nonportable) include recursive subroutine and function calls, automatically allocated local variables, bit fields and Boolean operators, relaxed input format, and 16-bit integer and double complex data types.

A FORTRAN preprocessor, called RATFOR ("rational FORTRAN") is also available. This preprocessor adds structured programming constructs to FORTRAN, making FORTRAN appear rather similar to C. RATFOR was more important before the advent of FORTRAN 77, which considerably "modernized" the language definition. However, it is still useful to sites that have substantial investments in RATFOR programs.

4.3.2.2 Pascal

pc is the Berkeley Pascal compiler, as enhanced by Sun. This compiler conforms very closely to the ISO Pascal standard. The following summarizes those aspects of Sun Pascal that may differ from other implementations.

Options Run-time checking of subrange and array bounds may be disabled; nonstandard usages can be flagged.

Violations The compiler does not conform to the standard in a few minor instances. For example:

1. Operands of binary set operators are required to have identical types rather than just compatible types.
2. n **mod** m produces an incorrect result for $m < 0$.

Extensions The most important extension is separate compilation of modules with intermodule type checking. Enumerated types may be read and written. Functions are not limited to returning scalars, but can return structures of arbitrary size. Built-in procedures have been added to integrate the language into the UNIX system environment, for example, to pick up command line arguments and to gain access to the system clock. Other extensions support 32- and 64-bit reals and make Pascal more

suitable for system programming.

A utility called **pxref** produces a cross-reference list of program identifiers.

4.3.2.3 C

The C language[5] is closely identified with the UNIX system. Virtually all system programming, and a good deal of application programming, in UNIX systems is done in the C language. C is a somewhat controversial language, with many supporters and a few detractors. To appraise the language in a balanced fashion, it is important to understand what a system programming language must provide, and to compare the language to the alternatives.

A system programming language *must* provide access to basic machine facilities such as addresses and bit fields. It must also provide a way to manipulate data as "just bytes"; that is, as of no particular type. Modern system programming languages, such as Modula-2 and Ada[6] recognize these minimum requirements. So, of course, do assembly languages, and the many system programming languages designed and used internally by computer manufacturers. Standard Pascal,[7] FORTRAN, and most other high-level languages do not.

Compared to assembly language, C is a great advance. C programs are easier to write and debug, and both programs and programmers can move with comparative ease to new machines. C programs are reasonably efficient, although this is highly compiler-, machine-, and application-dependent. Size and speed penalties are typically 20-40% over *good* assembly language code. (At the same time, C also preserves some of the "fun" of assembly language programming, for example, address arithmetic.)

If compared to a more modern system programming language, C may appear deficient; its syntax is somewhat irregular and can lead to cryptic (and compact) programs. Its general lack of restrictions ("permissiveness" or "flexibility," depending on how you look at it) can lead to subtle and hard-to-find bugs. There is no run-time checking (for example, subscript ranges), and exhaustive compile-time checks are delegated to a separate utility that programmers can ignore. Finally, C does not have the kind of constructs (for example, abstract data types and concurrency) that can make some classes of programs easier to write and maintain.

However, no modern system programming language has attained the acceptance of C. Ada, Modula-2, LISP, and so on, will have to gain much wider followings to assert themselves as practical rivals. For the next several years, millions of lines of system code will be written in C and thousands of programmers will learn to

5. ANSI committee X3J11 is drafting a C standard. For the present, compilers are sometimes informally compared to Brian W. Kernighan and Dennis M. Ritchie's book *The C Programming Language*, published by Prentice-Hall in 1978. Sun's C implementation is based closely on this definition, allowing for the book's ambiguities (it was never intended as a standard), and includes several standard extensions (such as **void**) which have appeared since the book was published.

6. Ada is a trademark of the U. S. Department of Defense.

7. Computer manufacturers who use Pascal as their system programming language have all extended the language to make it suitable for the purpose.

write it. C, for all its faults, is a very effective system programming language and is likely to be the dominant such language for at least several years.

Table 4.2 summarizes the language's principal constructs as implemented by Sun. To give some sense of the "flavor" of a C program, Figure 4.3 implements a function that performs a binary search on an array of arbitrary size.[8] Each item in the array consists of an integer key (the search key) followed by a character string.

Table 4.2 *Sun C Language Elements*

Data Types	character, integer (8-, 16-, and 32-bit); unsigned integer (8-, 16- and 32-bit); real (32- and 64-bit); bit field; array; structure (record); enumeration; pointer; union (multiply defined storage area); programmer-defined.
Storage Classes	automatic; static; register.
Initialization	static and automatic data, except for structures.
Operators	assignment; standard arithmetic; increment/decrement; logical; address comparison and arithmetic; bit manipulation; type coercion.
Control	if-else; switch (case); while (top-of-loop test); do (bottom-of-loop test); for (indexed loop); break (escape from loop or switch); continue (skip to loop test); goto.
Subprograms	functions; parameters are passed by value except for arrays which are passed by reference; functions can accept a variable number of arguments and can be called recursively.
I/O Library	read and write variable-size blocks; read and write formatted data, including control characters; reposition in file to effect random access.
Other Libraries	string processing; dynamic storage allocation; mathematical functions.

The C compiler is oriented more toward rapid compilation than to extensive compile-time checking. Another UNIX system utility, called **lint**, is dedicated to "picking bits of fluff" (this is apparently the source of the name) from C programs. **lint** flags constructs that are likely to be unportable, checks parameters and arguments for consistency (even across independently compiled modules), and locates uninitialized variables, unreferenced variables, and unreferenced statements. In effect, it gives C many of the advantages of a modern strongly typed language, but lets programmers choose whether the extra processing time required to make the checks is worth it.

4.3.3 Debuggers

Two source-level debuggers are commonly found on UNIX systems, **sdb** from Bell Labs and **dbx** from Berkeley. Sun supplies the latter because it has better facilities for controlling execution and for displaying complex data structure elements. (With **dbx**'s command aliasing facility, **sdb** users can, if they desire, make its syntax almost identical to **sdb**'s.) Sun has enhanced **dbx** so it works with any of the Sun programming languages and has added a number of useful features to it.

8. This function is from A.R. Feuer and N.H. Gehani, "A Comparison of the Programming Languages C and PASCAL," *Computing Surveys*, Vol. 14., No. 1, March 1982.

Fig. 4.3 *Sample C Program*

```
#include <stdio.h>
#define FAILURE (-1)

typedef struct {
    int     key;
    char   *value;
}               item;

int BinarySearch(A, n, k)
/*return index of item in A with key k,
        return FAILURE if k is not in A*/
item A[];
int n, k;
{
        int low = 0, high = n - 1, mid;
        while(low <= high) {
                mid = (low + high) / 2;
                if (k < A[mid].key)
                        high = mid - 1;
                else if (k > A[mid].key)
                        low = mid + 1;
                else
                        return(mid);
        }
        return(FAILURE);
}
```

dbx is a source-level debugger that can be used to monitor and control any program compiled with the **-g** option (this option directs the compiler to include symbolic information, mainly variable names and types, in the object file). **dbx** includes the facilities usually associated with symbolic debuggers: starting and stopping program execution, setting and removing breakpoints, single-stepping, displaying variable values, tracing statements as they are executed, tracing variable values as they change, and so on. **dbx** can also break when a variable changes value, or takes a particular value, and lets the user call functions and procedures interactively. **dbx** is very adept at displaying data, for example:

```
print x              (* prints elements of x if x is a structure *)
print array[x+y]     (* evaluates x+y and prints the corresponding
                        element in array *)
print ptr^           (* prints what ptr points to *)
```

dbx also provides a set of machine-level facilities that allow displaying memory locations and registers, and tracing, breakpointing, and single-stepping by machine instructions rather than high-level language statements.

Sun also supplies a window/mouse-based interface to **dbx** called **dbxtool**. One in the series of Suntools, **dbxtool** makes **dbx** extremely easy to use. For example, when a program is running under **dbxtool**, its output is displayed in one window, while the source text is scrolled automatically in another as the point of execution changes.

The alternative to **dbx** is **adb**, an assembly-level interactive debugger found in many UNIX systems. Its low level of function and cryptic syntax do not make **adb** the debugger of choice, but it can do some things that **dbx** cannot. First, **adb** does not need a symbol table to run, so it can be used with programs not compiled with a symbol table option. (**adb** *can* use the symbol table produced by the **-go** compiler option.) Second, it can be used to patch code in memory or in a file. Finally, **adb** is a good general tool for interactively examining binary files of any sort.

4.3.4 Profilers

When a program is compiled with the **-pg** option, the compilers add extra code to the object module that "instruments" the program at the procedure/function/subroutine level. Two things happen when such an instrumented program is subsequently executed. Every time a routine is called, a counter, which records the number of times that routine has been called, is incremented. A second counter, which tabulates the number of time this routine has been called by every other routine, is also incremented. Every clock tick (20 ms.) the routine containing the address of the machine's program counter is "charged" for consumption of the previous tick. When the program terminates, the raw data gathered by the instruments is written to a file.

The **gprof** utility tabulates the raw data into a report that lists the time used by the routines in the program. For each routine, **gprof** reports how many times the routine was called and how much processor time it consumed.[9] **gprof** also produces a call graph that identifies the run-time relationships between routines. The information contained in the call graph report is comprehensive and detailed, but it essentially apportions each routine's call count and time consumption data among its callers. The result is particularly illuminating for recursive routines.

A second profiler, provided by Sun and called **tcov**, exposes test case coverage rather than execution characteristics. Compiling a program with the **-a** option invokes a preprocessor that instruments a program for test coverage analysis. When the instrumented program is run, test coverage data is produced. The **tcov** utility analyzes this data and produces reports that show which parts of the program were exercised in the run and which remain to be tested.

9. This is the same information provided by **gprof**'s predecessor **prof**, the execution profiler found in many UNIX systems. **gprof** was developed at Berkeley to both emulate and extend **prof**'s capabilities.

4.3.5 Management

It is good practice to break large programs into modules that are small enough to be individually manageable. Eventually, however, the number of these units and their interrelationships may become another problem, particularly as the program evolves. The UNIX system includes two important utilities for managing the configuration of software products. While these may not be the ideal tools, they are extremely valuable; as one indication, every software group within Sun uses them heavily. The utilities may be applied to documentation projects as well as to software.

The first is called **SCCS** for "Source Code Control System." SCCS is a kind of source code custodian, whose job is to track the successive versions of modules (more generally, of text files). Once a file has been placed in **SCCS**'s custody, it is normally only retrieved, modified, or updated through **SCCS** commands. For each version of each file, **SCCS** maintains change information that indicates what, when, why, and by whom the changes were made. SCCS does not store each version as a separate file, but rather stores a base file and, separately, its changes. This approach conserves storage and still allows **SCCS** to reconstruct any version of a file by simply applying the appropriate sequence of changes to the base file. Authorized users can "check out" a version of a file to update it, and **SCCS** will prevent another user from also checking out the file, eliminating the possibility of multiple uncoordinated updates.

make is a utility that automatically recompiles the modules of a program that are affected by changes. To do this, **make** relies on a file called a **makefile** that describes the interdependencies of modules, and the commands that must be executed to create a consistent version of the program. The UNIX system timestamps each file whenever it is written. When **make** is invoked, it examines these timestamps to determine which files must be recompiled (a file must be recompiled if a file it depends on has a later timestamp). **make** recompiles the minimum number of files and relinks the program. **make** can simplify the management of documentation as well. For example, a manual may have 10 chapters, stored as separate manuscript files. The manual is printed by passing the chapters through a formatting program (such at **troff**, discussed later in this chapter) to produce the files that go to the printer. A **makefile** can be set up to reflect the dependency of each printer file on its corresponding manuscript file. Now suppose changes are made to chapters 2 and 8 only. **make**-ing the manual will run only chapters 2 and 8 through the formatter, because only their printer files are out of date. (This example assumes that the manual has chapter-relative page numbering.)

4.3.6 Compiler Construction

Two UNIX system utilities are of interest to users contemplating the implementation of new languages for UNIX systems. **yacc** ("yet another compiler-compiler") is a utility that generates a parser from a grammatical description of a language. (Many UNIX system utilities, including **dbx** and the C compiler, are based on **yacc**.) Similarly, **lex** generates a lexical analyzer from a description of a

language's lexical rules. Both programs produce C code.

4.4 Filters

A substantial number of UNIX system utilities are classified as **filters**. A filter reads its standard input, transforms the data in some way and writes the result to its standard output. The idea behind the name is clearer when one visualizes how filters are commonly employed. A common construction is several filters joined in a pipeline, as shown below:

```
filter1 < input-file | filter2 | filter3 > output-file
```

Bytes flow from the input file to the output file through the pipe. The filters are interposed in the pipeline; the first filter might translate certain values, the second filter might rearrange the lines (typically by sorting), and the third might eliminate duplicate lines. Thus, each filter performs a relatively small, specialized function. When used in combination they perform a more complex function — and the number of possible combinations is, of course, very large. The last section of this chapter is devoted to examples of filters, and other utilities, combined in various ways.

Some of the minor filters are described briefly in Table 4.3; the remainder of this section covers some of the more interesting examples of the class. Note that each of the utilities described in this section is *significantly* more powerful than the very limited description presented here suggests.

Table 4.3 *Minor Filters*

Filter	Function
crypt	Encrypt/decrypt a stream
compact	Compress stream by Huffman coding
uncompact	Expand Huffman-coded stream
tr	Translate characters
colrm	Remove columns
expand	Expand tabs to spaces
unexpand	Compress spaces to tabs
fold	Fold long lines into fixed length
join	Join lines in two files that have identical fields
rev	Reverse characters in every line
tee	Copy standard input to standard output and additional files
uniq	Count and remove duplicate lines
wc	Count characters, words and lines

4.4.1 grep

grep (the name will be explained shortly) selects lines from a file, based on matching a pattern. The pattern is commonly just a simple string. For example, the following searches all C source files in the current directory for references to the function **BinarySearch**:

```
grep BinarySearch *.c
```

Useful options to **grep** can select lines that do *not* match the pattern, can print the number of each matching line, and can make a lower-case pattern match on uppercase as well.

grep can also search for a pattern specified by a regular expression — and this is the source of the program's name.[10] Regular expressions allow the concise specification of complex matching criteria, such as "lines that begin/end with *pattern*," "a line containing the *n*th occurrence of *pattern*," "lines containing *pattern1* followed by *pattern2*," and so on.

fgrep and **egrep** are similar to **grep**. **fgrep** can search for multiple patterns in parallel, but can only search for fixed strings, not regular expressions. **egrep** is faster than **grep** under some circumstances and generalizes the notion of regular expression. **egrep** can, for example, extract lines containing any of several alternate regular expressions; in other words, the logical OR of several regular expressions.

4.4.2 sed

sed is a stream-oriented version of **ed**. It applies a list of editor commands to each line in one or more files, writing the resulting modified lines to standard output. The lines to be edited can be selected by regular expressions as described above.

sed is a very convenient tool for removing unwanted fields, transforming all occurrences of one string to another string, or adding a new field to each line. As a very simple example, the following shell script indents the lines in a file by inserting a tab at the beginning of each line:

```
sed 's/^/<tab>/' $*
```

(In the above example, the tab character is represented by the string **<tab>** in order to make the tab's location visible on this page.)

4.4.3 awk

awk takes its name from its Bell Labs authors, Aho, Weinberger, and Kernighan. It is a pattern matching *programming language* whose syntax is modeled after C. It is often used as a report generator. Like **grep**, **awk**'s basic action is to select lines from its standard input. Several selection criteria can be specified; all are applied in a single pass over the input stream. However, it has substantially more power than **grep**; this power is a consequence of the *actions* **awk** can perform on selected data. The simplest action is to merely write the selected line to the standard output, like **grep**; however, it is wasteful to use **awk** like this because **grep** can do the same thing faster.

Having selected a line, an **awk** program specifies the action to be performed; each selection criterion may have a different action.

10. The name is actually a contraction of **ed**'s **g**/*regular-expression*/**p** command, which *g*lobally applies a *r*egular *e*xpresson to a file and *p*rints the result, where printing means writing to standard output.

Action statements can operate on fields (by default delimited with spaces and tabs) in the selected line and on variables you declare. Actions include arithmetic, conditional tests, and string processing. Fields, modified fields, constants, and variables can be written to the standard output. Special actions can be taken at the beginning and end of the input stream, for example, to write totals. The power of **awk** is probably underappreciated and underutilized at most UNIX system installations; to fully describe it would require a short book.

4.4.4 sort

The UNIX **sort** utility sorts the lines in a file. The lines may contain any bytes and may be considered as being composed of fields separated by blanks or other characters, for example, semicolons or slashes. Options to **sort** permit sorting on multiple fields, on specific bytes within fields, into ascending or descending order, ignoring case or white space (blanks and tabs), and so on. Besides sorting, **sort** can merge sorted files, optionally eliminating duplicate lines — a handy feature for mailing lists, for example.

4.5 Formatters

From its beginning, the UNIX system has emphasized text formatting tools nearly as much as program development tools. (Of course, many tools, notably the editors and filters, apply equally to both kinds of work.) The key text processing tools are the **nroff** (pronounced "en-roff") and **troff** ("tee-roff") formatters. Note that a formatter is not a word processor — it translates a stream containing ordinary text and commands into a formatted document. The process of creating documents with formatters is essentially analogous to the process of developing programs with compilers.

troff and **nroff** are input-compatible; they differ in the output devices for which they are designed. **troff** generates output for a typesetter or a device, such as a laser printer, that can emulate a typesetter. **nroff** works with simpler output devices such as line printers, daisywheel printers, and terminals. **nroff** achieves compatibility with the more complex **troff** by simply ignoring **troff** commands, such as "change to 12 point type," that are not meaningful for the devices it drives. Therefore, everything said in this section about **troff** applies to **nroff** as well; a document formatted by either of them will produce the same result, within the limitations imposed by the relevant output devices.

troff means roughly "runoff for typesetter," and is one of the many offspring of a fairly ancient and well-known program called "runoff" written at MIT. **troff** processes an input stream containing the text to be formatted, interspersed with commands specifying how it is to be formatted. Figure 4.4 shows an example; lines beginning with periods are **troff** commands, as are strings

beginning with backslashes.

Fig. 4.4

troff - Input -

Example

```
.ce
Chapter 1
.sp 3
.ti +2m
If you really want to hear about it, the
first thing you'll probably want to know is where I was born, and
what my lousy childhood was like, and how my parents were
occupied and all before they had me, and all that David
Copperfield kind of crap, but I don't feel like going into
it.
In the first place, that stuff bores me, and in the second
place my parents would have about two haemorrhages apiece if I
told anything personal about them.
They're quite touchy about
anything like that, especially my father.
They're
.ft I
nice
.ft R
and all\(emI'm not saying that\(embut they're also touchy as
hell.[11]
```

- Output -

Chapter 1

If you really want to hear about it, the first thing you'll probably want to know is where I was born, and what my lousy childhood was like, and how my parents were occupied and all before they had me, and all that David Copperfield kind of crap, but I don't feel like going into it. In the first place, that stuff bores me, and in the second place my parents would have about two haemorrhages apiece if I told anything personal about them. They're quite touchy about anything like that, especially my father. They're *nice* and all—I'm not saying that—but they're also touchy as hell.

troff shows its age; like so much of the UNIX system it reflects an era when the teletypewriter was the principal terminal. Its rigid, abbreviated syntax and low level of function make it effectively an assembly language for a typesetter. Much like assembly language programming, it is possible to do just about any formatting task with **troff** commands, but it is difficult to do even simple things (for example, running headers and footers). Fortunately, one of **troff**'s facilities is the ability to define macro

11. J. D. Salinger, *The Catcher In The Rye*, Hamish Hamilton, Great Britain, 1951

packages; and, in fact, most people rely on a macro package to do most of their formatting, dropping down to direct **troff** commands only when necessary to create a special effect. The most widely used macro package is called **ms**. **ms** understands higher-level document constructs such as chapters, sections, paragraphs, footnotes, and so on. Its basic commands are shown in Table 4.4.

Table 4.4 *Basic* **ms** *Commands*

Command	Function
.B	Start boldface
.I	Start italics
.R	Start roman
.DS/.DE	Start/end display (literal text)
.FS/.FE	Start/end footnote
.KS/.KE	Start/end text to keep together on same page
.KF/.KE	Start/end text to keep together; float to next page permitted
.PP	New paragraph
.LP	New left-justified paragraph
.IP *string*	New indented paragraph with *string* in margin
.NH *n*	New *n*-th level numbered heading

Other, less important, text processing utilities are provided with the UNIX system. **spell** finds spelling errors in a text stream. **deroff** strips **troff** commands from a stream. **man** is a macro package that produces pages formatted like the UNIX system command documentation. It is useful for documenting locally developed utilities in a form recognized throughout the UNIX community.

More interesting are the **tbl** and **eqn** preprocessors. These are formatting languages especially oriented toward producing tabular and mathematical constructs, respectively. Both accept streams containing their own commands interspersed with **troff** and/or **ms** commands. Passing through commands that do not concern them, the preprocessors convert their own commands into low-level **troff** commands and insert the result into the output stream. Thus, to process a complex document containing tables, equations and text, you run it through a pipeline as shown in Figure 4.5. Figures 4.6 and 4.7 show examples of **tbl** and **eqn** input files and formatted results.

Fig. 4.5 *A* **troff** *Pipeline*

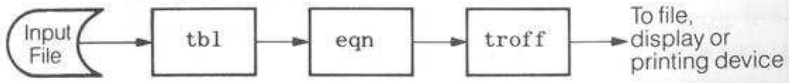


Fig. 4.6 **tbl** *Example*

- Input -

```
.TS
center box;
cB | cB | cB
cB | cB | cB
c  | n  | n.
Partition Contents      Partition Size  Partition Size
                        (42 MB Disk)    (Larger Disks)
=
root filesystem (/)     8 MB            8 MB
_
[swap space]            8 MB            16 MB
_
user filesystem (/usr)  26 MB           40 MB +
.TE
```

- Output -

Partition Contents	Partition Size (42 MB Disk)	Partition Size (Larger Disks)
root filesystem (/)	8 MB	8 MB
[swap space]	8 MB	16 MB
user filesystem (/usr)	26 MB	40 MB +

Fig. 4.7 **eqn** *Example*

- Input -

```
.EQ
f( zeta ) =~ 1 over {2 pi i} int from C
        f(z) over {z - zeta} dz
.EN
```

- Output -

$$f(\varsigma) = \frac{1}{2\pi i}\int_C \frac{f(z)}{z-\varsigma}\,dz$$

4.6 Communication

Included in the UNIX system are utilities for sending electronic mail to users on the same machine, to users on other machines which are members of the UNIX User's Network, called USENET, and to users on other networks, such as ARPANET and CSNET, that are connected to USENET. Berkeley extended the mail facility to apply to machines on the local network as well. The **tip** (terminal interface program) utility provides an alternate means of transfering data between computers.

4.6.1 mail

Mail may be sent to users on the same machine, on different machines in a local network, or on machines connected to USENET. Sending mail is a simple matter of invoking the **mail** utility with an argument designating the address to which the mail is to be sent. The **mail** utility accepts the text of the message, terminated by a **Control-D**, the normal terminal end-of-stream indicator. Having captured the message text, the **mail** utility adds the message to the queue of the local machine's **sendmail** daemon, wakes up the daemon, and terminates.

sendmail's job is to deliver mail. **sendmail** is awakened by **mail** whenever a new message is to be sent. It is also awakened periodically to attempt redelivery of outstanding messages, for example, a message addressed to a machine that was down when delivery was originally attempted.[12] Delivering a message means appending it to a file, called a **mailbox**, associated with the addressee. Delivery can be straightforward or somewhat involved, depending on where the addressee's mailbox is located relative to the sender. There are three basic locations: on the same machine, on the local network, or on the USENET network. Corresponding to these locations are three address formats, examples of which are shown below as arguments to the **mail** command:

```
mail jacque
mail terry@polaris
mail decwrl!sci!rob
```

Common to each of these is the addressee's **login-name** (**jacque**, **terry** and **rob**, in the example). (The mail system's **alias** facility, which permits other forms of address, is discussed shortly.) No other information is necessary when the addressee's mailbox is on the same machine as the sender: **sendmail** looks up (in a file kept for the purpose) the name of the addressee's mailbox file and appends the message to the mailbox. If the **login-name** is invalid, **sendmail** returns the mail, with an explanation, to the *originator's* mailbox.

An address argument such as **terry@polaris** means that **terry**'s mailbox is on a machine called **polaris**. To deliver this mail the **sendmail** daemon on the originating machine needs to forward the message to the **sendmail** daemon on the destination machine. It does this with the aid of a file called **/etc/hosts** and the Sun network communication facilities. **/etc/hosts** contains the name and internet address of every machine (host) on the local network. On each machine is a socket (recall that a socket is an interprocess communication mechanism) that has been established as the mail service connection. The **sendmail** daemon listens on this socket for mail service requests. Having established a connection via the socket, the two daemons conduct a conversation, checking each other's identities and the **login-name** of the addressee. If everything is valid, the message is transmitted over the net and the receiving daemon delivers it just as if it had come from its queue instead of another machine. If the **login-name** is invalid, the originating **sendmail** daemon returns the message to its sender. If the destination host is down, so that the attempt to establish a connection fails, the originating **sendmail** daemon attempts redelivery later; after a configurable number of days of redelivery attempts, it also returns the message to its sender.

12. Implementation note: **sendmail** does not deliver messages directly, but **fork**s an instance of itself to handle each message. This technique insures that the mail system remains operational even if an instance of **sendmail** hangs during its delivery attempt.

It is clearly inconvenient for users to have to remember or look up **login-names** and **hostnames** to send mail over a network that might consist of hundreds of machines and users. The nuisance can be eliminated with a simple convention and the mail system's **alias** facility. The idea is to establish a uniform naming convention for people, and a single machine on which an up-to-date alias file is maintained. For example, suppose people are named by first initial concatenated with last name, and the machine with the alias file is called **po** (for post office). Then, to send mail to Alfred Newman, whose **login-name** and **hostname** are unknown (or may have changed), the following suffices:

```
mail anewman@po
```

The message is delivered first to **po**, where the **sendmail** daemon looks up **anewman**'s **login-name** and **hostname** in the alias file and then forwards the message to its final destination. As it happens, aliases are quite versatile. They can also be used for mail distribution lists, allowing, for example,

```
mail marketing@po
```

to send a message to all persons aliased to **marketing**. An alias need not point to a **login-name** and **hostname** but may instead specify a program designated to print that person's mail in hardcopy form; in this way individuals who do not have terminals can receive mail.

When **sendmail** does not recognize a **hostname**, (for example, a new machine)[13] or an address format (as it would not recognize, for example, **decwrl!sci!rob**), **sendmail** forwards the mail to the system designated as **mailhost** in its **/etc/hosts** file. Besides having the most current version of **/etc/hosts**, the **sendmail** daemon on this machine has access to a file that describes more than just the local network environment. (The information is localized in one machine to simplify maintenance and to establish a control point which can help to prevent reissuing of **login-names**, for example.) In particular, it understands foreign address formats, other networks to which it may serve as a gateway, and machines that are its adjacent USENET nodes. Mail transmission over USENET is the subject of the next section.

So much for sending mail; how do you receive it? When you log into a UNIX system, the system checks your mailbox and notifies you if new mail has arrived since your last login. In addition, the shell can, at your request, automatically check for new mail after executing each command and notify you when new mail arrives. To be notified asynchronously when mail arrives (for

13. Keeping the **/etc/hosts** files of all machines on the network reasonably up to date is simple. One publicly known machine's file (the machine corresponding to **po** would be a logical candidate) is updated manually, and the other machines periodically, say nightly, run a shell script that copies this file.

example, when you are expecting an important message during a long editing session), you can issue the **biff** command. To read mail, you just type **mail** with no address argument. The **mail** utility displays a summary line for each message in your mailbox. These messages may be read, saved, edited, deleted, replied to, and so on, using the **mail** utility's interactive commands.

4.6.2 USENET

Many UNIX system sites (around 2,000 in 1984) are members of a voluntary cooperative network called USENET. Some USENET nodes provide access to other networks such as ARPANET and CSNET. The network is implemented with dial-up telephone lines and a transfer program called **UUCP** (UNIX-to-UNIX **cp**, that is, copy). **UUCP** employs a transfer protocol that is highly optimized for telephone line transmission.

Each node on the network maintains a list of (usually nearby) sites to which it has agreed to periodically transfer data. It also maintains accounts for the sites from which it has agreed to receive data. Data is transferred when one site calls another, logs in, and executes the **UUCP** utility which copies files from one system to the other.

To send mail to a user at another USENET site, you must specify a path to the user's site. For example,

```
mail decwrl!sci!rob
```

means that the message goes first to the site known as **decwrl**, then to the site known as **sci** where **rob** is a user. How long the message takes to traverse the "hops" in its path to its ultimate destination depends on how frequently the sites in the path call each other up. The network is organized with a number of long-haul links that make it possible send a message to most sites in North America, Europe, Australia, and Asia in a half-dozen or fewer hops. Other networks, reachable through sites serving as gateways from USENET, have their own address formats which are interpreted when a message arrives at a gateway.

4.6.3 news

A second class of information, known as **news**, also travels over USENET. News is functionally similar to an electronic bulletin board, but it is implemented in a decentralized fashion. News articles are organized into categories called **newsgroups**. There are presently on the order of 250 newsgroups covering such diverse subjects as workstations, graphics, languages, networks, architecture, bugs, and conference announcements (not to mention wine, music, books, rumors, motorcycles, philosophy, and birds). News moves around the net in basically the same fashion as mail. However, each article is tagged with its newsgroup and a unique identifier (derived from the originating site's name) to obtain a reasonable degree of efficiency. Each site subscribes to a set of newsgroups known to the sites from which it may receive news. When a connection is established, the sending site transfers all articles in the categories the receiving site wishes to receive, and the

receiver discards (using the article identifier) any articles it has already received from another machine.

To submit a new article you run the **postnews** utility. This program prompts for the newsgroup and subject and then **fork**s you into an editor so you can compose your article. When you exit the editor, **postnews** queues the article for transmission onto the net.

The **readnews** utility lets you read articles. You can specify the newsgroups you want to read routinely in a file called **.newsrc**. **readnews** updates this file so that each time you run **readnews** you only see articles that you have not already read. For each article, **readnews** displays a header showing the subject, author, and length of the article. You can then read the article or skip it. There are many other commands as well, one of which allows you to mail a reply to an article's author. Each site sets its own rules for dropping old articles, but they are typically kept for about two weeks.

4.6.4 tip

tip is a terminal interface program that makes a process appear to another computer as a terminal. **tip** supports selected autodialing modems so you can pass a telephone number as an argument and **tip** will make the call for you. **tip** can send and receive ASCII files and can run a program on the other computer, connnecting its output to the communication line. Binary files can be sent or received with the aid of a utility called **uuencode**.

4.7 Putting Utilities Together

As mentioned at the start of this chapter, the UNIX system utilities are individually substantially more powerful than the simplified descriptions provided here. What is more important, however, is to appreciate the really remarkable power of utilities used *in combination*, with shell code (including pipes) binding them together. Given the power of the utilities, it is good practice, when faced with a new problem, to *resist the urge to code*. Rather, you should review your existing tools — utilities supplied with the UNIX system as well as those developed locally — to see if there is a way to look at the problem that will make it yield to a combination of existing, debugged tools.

This section includes a few examples that show how utilities can work together. Underlying all these examples is an approach that sees a problem as being composed of subproblems to which utilities can be fruitfully applied. Then the utilities are connected in a pipeline or shell script to solve the problem as a whole.

4.7.1 Disk Space Consumers

Here is a command that a system administrator might routinely employ to find out which users are consuming the most disk space:

```
alias dhog 'du -s /usr/* | sort -nr | more'
```

It is implemented here as a C-Shell alias, but could as well be a shell script. The **du** (disk usage) utility scans a list of directories, looking at the space occupied by the files in those directories, including files in their subdirectories (and their subdirectories, and so on). The example assumes that each user files are stored in subdirectories of **/usr**; for example, the files of user **gage** are all stored in **/usr/gage** and its subdirectories. The **-s** option tells **du** to prepare only "summary" information — to write one line per directory totaling the kilobytes consumed by that directory and its descendants. Piping **du**'s output to **sort -nr** orders the lines numerically, in reverse order, so the largest consumption figures come out on top. The output is then piped to **more** so it is displayed one screenful at a time. The result looks something like this:

```
14256 /usr/gage
12300 /usr/billc
996 /usr/carl
...
```

4.7.2 Frequently Used Words

Suppose — this example is not so much practical as illustrative — that you want to know the ten most frequently used words in a file or set of files. Think about this problem for a moment; it is just the sort of program that you think you can crank out in half an hour and ends up taking two days. Here is a shell script that does the job.[14]

```
#! /bin/sh
cat $* |
tr -sc A-Za-z '\012' |
sort |
uniq -c |
sort -n |
tail
```

The **cat** step concatenates all the files passed as arguments. The **tr** step eliminates case distinctions and compresses runs of nonletters into newlines, producing one line per word. Then the words are sorted to bring duplicate words together. The **uniq** utility eliminates the duplicates, writing one line per distinct word that includes the word and a count of its occurrences. The result is sorted numerically so the most frequent words come out last; **tail** picks off the last ten words.

14. The example comes from Kernighan and Pike, *The UNIX Programming Environment*, Prentice-Hall, 1984.

4.7.3 A Primitive Rhyming Dictionary

Another example, again more illustrative than practical, produces a simple-minded rhyming dictionary:

```
rev /usr/dict/words | sort | rev
```

(The rhyming dictionary is simple-minded because it equates rhyming with similar spelling — "through" and "rough" will rhyme according to this dictionary.) **/usr/dict/words** is a file distributed with the UNIX system that contains about 25,000 words, arranged one word per line. **rev** is a utility that reverses the order of the characters in a line, thus turning a line containing "tree" into "eert." Sorting **rev**'s output brings together all of the words ending in "a," and so on. To make the list readable, though, the letters in each word need to be reversed again, hence the last step.

4.7.4 Finding CPU Gluttons

A UNIX system will commonly run several, even dozens, of processes concurrently. Figure 4.8 is a shell script that finds the processes that are using the most CPU time.

Fig. 4.8 *CPU Utilization Shell Script*

```
#! /bin/sh
echo -n "                    ";    date;    echo ""
ps axgcu |
awk '($3>=1.0 || $3=="%CPU") \
        && substr($0,44,5)>=" 0:05" \
        && $NF!="CSH"' |
sort -r +.14 -.25
```

The script begins by displaying the system date and time followed by a blank line. It then executes the **ps** (process status) utility which generates one line of data for every running process in the system. The data produced by **ps** includes process id, login name, percentage of CPU time and memory used, and the name of the command that the process is running. To highlight only the large consumers, an **awk** program selects a subset of the lines produced by **ps**, namely those that have used more than 1% of the CPU, have been running for more than five seconds and are not C-Shells. It also selects the column header line produced by **ps**. Finally, the selected lines are sorted in reverse order based on CPU and memory consumption so that the largest consumers are displayed first.

5 *The SunWindows System*

This chapter describes the SunWindows user interface through which Sun workstation users interact with the computer and the network. The discussion is divided into two parts; they describe the SunWindows system from a user's point of view and from a programmer's point of view. Because this report's principal topic is the UNIX system, the SunWindows interface is only introduced here. Readers who take the time to become more fully acquainted with the Sunwindows system will find it richer and more flexible than described here.

The SunWindows interface is based on concepts pioneered and proved at the XEROX Palo Alto Research Center in the 1970s. These concepts have been popularized, in a simplified form, by the Macintosh[1] personal computer. This style of interface requires comparatively abundant CPU cycles, memory, and display bandwidth. A Sun workstation has exactly these attributes, but the minicomputers and terminals on which the UNIX system has been traditionally run do not. This lack of suitable hardware is one reason the often-criticized UNIX system user interface has persisted for so long.

5.1 Using SunWindows

When a Sun workstation is booted, it comes up as described in Chapter 3. The screen image looks like a that of a large character terminal displaying the prompt:

```
login:
```

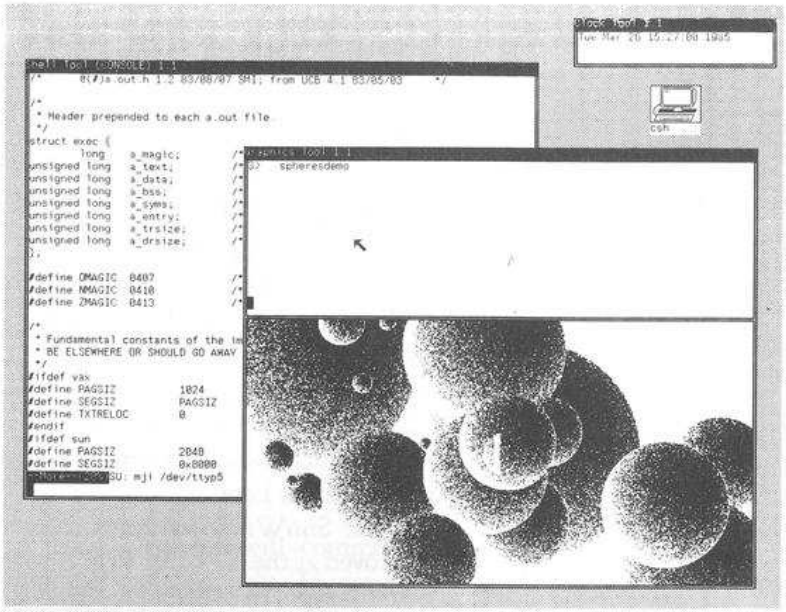
After logging in, one of the first commands most users issue is **suntools**. **suntools** is the utility that starts up the SunWindows user interface. Guided by a file of user-specified parameters, **suntools** radically transforms the appearance of the workstation screen, making it look more or less (depending on user parameters) like Figure 5.1. This section describes the objects shown in that figure and how you can manipulate them.

5.1.1 The Display and the Mouse

The SunWindows system is based on the Sun high-resolution bit-mapped display and the "pointing device" known as the **mouse**. The Sun display is a remarkably versatile medium for presenting information. Software can "paint" arbitrary combinations of text (in various sizes and typefaces), lines, curves, and

1. Macintosh is a trademark of Apple Computer, Inc.

Fig. 5.1 *Typical SunWindows Display*



shading on the Sun display. Because the display is bit-mapped, software can update one part of the display without altering other areas by simply changing the bits in the frame buffer (bit map) that correspond to the screen locations of interest. The color displays of some Sun models are even more versatile and are especially adept at presenting complex information.

Steering a traditional cursor around a large bit-mapped display with "arrow keys" is not a satisfying experience. The cursor can only be moved in two axes, movements are slow, and positioning is crude, with resolution limited to one "standard" character. Accordingly, Sun workstations are supplied with a three-button mouse (see Figure 5.2). Moving the mouse on its pad causes a **mouse cursor**, by default a bold arrow pointing at 300 degrees, to move on the screen. The mouse cursor can be moved very rapidly in any direction and positioned very precisely; many users can pick out a single pixel with it.[2] The mouse buttons are used to select objects on the display and to enter predefined commands, in the manner of conventional function keys. Sun's mouse has three buttons, making it a little harder to learn than a one- or two-button mouse, but ultimately more powerful.

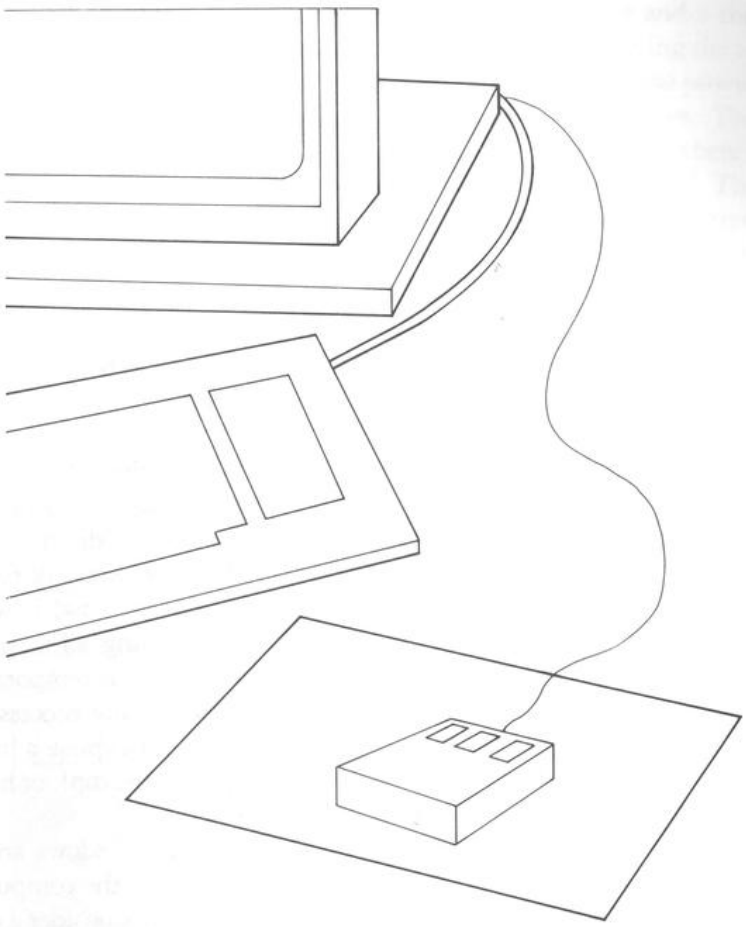
5.1.2 Windows

The Sun display is roughly twice as large as a typical character terminal. This workspace is large enough to be usefully divided into smaller areas, each corresponding to a different work activity. These rectangular areas are called **windows**.

Associated with each window is an application, or **tool**, consisting of one or more UNIX processes. The tool controls the

2. A pixel, or picture element, is a single dot on the screen. For monochrome displays, a pixel is represented by one bit in the system's frame buffer memory. About 70 pixels per inch are displayed on a Sun monitor.

Fig. 5.2 *Mouse Pointing Device*



window, displaying data in it and interpreting mouse and keyboard input directed to it. (Input is directed to a tool when the mouse cursor is in the tool's window.) Figure 5.1 illustrates a few standard tools that are supplied with the SunWindows system:

- **shelltool**, which emulates a character terminal running a UNIX system shell, thereby providing a traditional UNIX system interface in a window;
- **graphicstool**, which allows an existing graphics program to draw inside a window;
- **clocktool**, which simply displays the time of day.

These tools and other standard tools are described more fully later in this section. Users can also write their own tools, as described in the second half of this chapter.

Several windows may be displayed on the screen simultaneously; each window represents an activity in progress. For example, in one **shelltool** window you may be running **tip**, transfering a file over telephone lines to a remote computer; in a second **shelltool** window an editor may be waiting for your next command; in a third **shelltool** window the file you have just changed with the editor may be compiling; finally, in a **graphicstool** window you may be examining a chart drawn by

a graphics program. All of the activities represented by these windows may run concurrently, since they are embodied in the system as different processes.

At any time you may switch your attention from one window to another by moving the mouse cursor. For example, if the compilation mentioned above uncovers a syntax error, you can move to the edit window and make the fix; the editor will be exactly as you left it. After saving the updated file, you can return to the compile window and recompile. (Each **shelltool** runs a separate instance of the shell.) Thus, a window always preserves the context of the tool running in it. When you return to a window, not only is the tool's state exactly as you left it, but you see the evidence of that state.

Using the mouse, there are many ways to arrange and alter windows; for example, they can be moved and resized at any time. Windows need not occupy discrete areas of the screen; they can **overlap** one another like pieces of paper on a desk. For example, when you create a new window, it will typically appear near the center of the screen, **hiding** all or parts of windows in the same vicinity. A hidden window is temporarily removed from view, but is not otherwise affected; any processes running in it continue to run. At any time you can **expose** a hidden window (bring it from the bottom of the pile to the top), or hide the top window to expose the one underneath.

Switching between windows and opening new windows as needed lets you work with the computer the way you might work with papers on your desk. Consider a case in which your writing is interrupted by a phone call, and the phone call is interrupted by a visit from a colleague. You respond by shuffling the papers on your desk, placing items of immediate interest on top, perhaps temporarily hiding your previous work. As you finish with each interruption, you remove some of the top papers exposing your earlier work; eventually the paper you were writing resurfaces.

Besides their resemblance to the arrangement of work on a desk top, windows have analogies in operating systems. For example, the overlapping nature of windows enlarges your effective workspace in somewhat the same way that virtual memory enlarges a process's workspace. You can also think of windows as analogous to processes and yourself as playing the role of the operating system scheduler. Like the operating system you generally have more than one activity underway at a time. You multiplex your attention among the various activities according to their priorities, their "readiness," and in response to external events represented by interrupts. Thus, in addition to being based on multitasking, the SunWindows system brings the underlying operating system's multitasking capabilities out to your work surface, making them directly accessible and visible.

5.1.3 Subwindows

A tool's window is framed by a double-lined **border** and a thick **namestripe** at the top (see Figure 5.3). Besides identifying the tool and visually distinguishing window boundaries, the frame provides a common set of commands that apply to all windows. These commands are invoked by placing the mouse cursor anywhere on the frame and holding down the right mouse button.[3] These commands allow you to move, change the size of, hide, or expose any window, regardless of the tool running in it. The commands are implemented not by the tools themselves, but by a SunWindows facility called the **tool manager**.[4] In effect, the tool manager owns all the window frames, while each tool owns what is inside its frame.

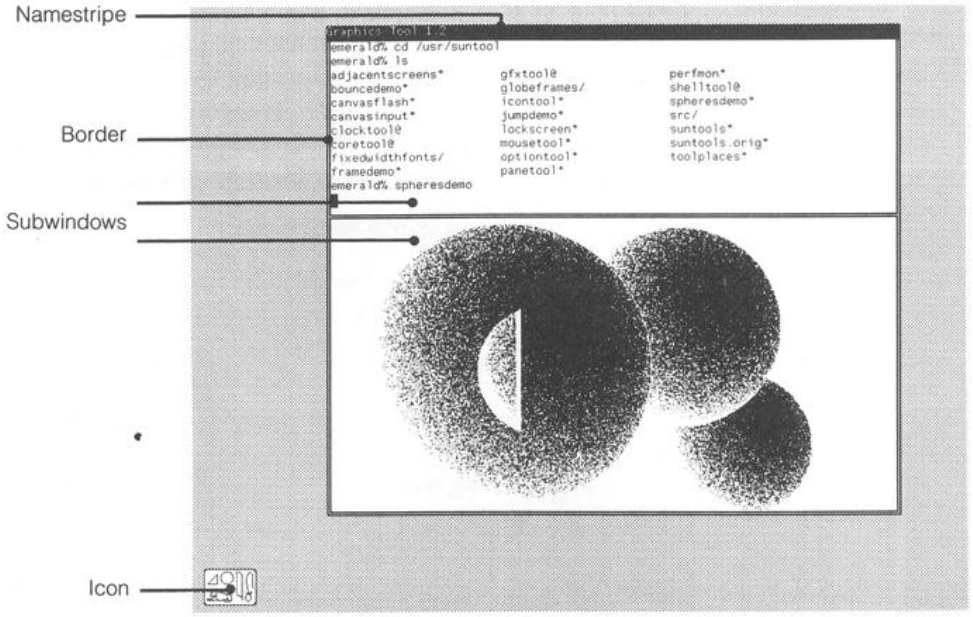


Fig. 5.3 *Anatomy of a Window*

Inside the frame is a collection of one or more **subwindows**; these are where a tool really does its work. Subwindows group related tool functions into visually distinct units. For example, in Figure 5.3, you can type commands to the tool in the upper subwindow, while the lower subwindow is the "canvas" upon which the tool draws pictures. Many other subwindow examples are illustrated later in this section.

Unlike windows, subwindows do not overlap; they are "tiled" over the area of a window. Subwindows move as a unit with their window and, if necessary, adjust their shapes in the window when you change the window's size.

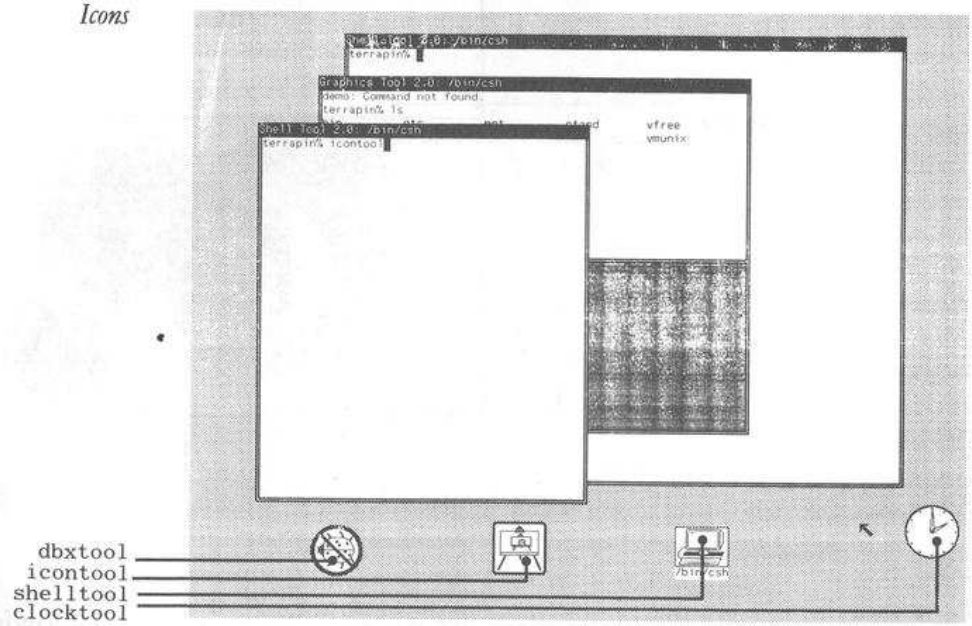
3. Holding down the right mouse button "pops up" a menu of commands; pop-up menus are discussed shortly.

4. The tool manager might be better called the "window manager" since it really manages windows. As will be described in the second half of the chapter, tools manage themselves.

5.1.4 Icons

While their overlapping nature allows you to "pile up" windows to arbitrary depths, you can reduce workscreen clutter by **closing** windows that are temporarily unneeded. A closed window changes into a small **icon**, a picture that symbolizes the tool associated with the window (see Figure 5.4). An icon occupies less space than a window while allowing you to quickly transform it back into a window when you need to. Icons can be moved around just like windows, and, of course, can be **opened**. Both **open** and **close** are tool manager commands. A tool may place its icon wherever it wishes on the screen; by default the tool manager lines up icons row along the bottom of the screen, keeping the center area clear for open windows. Note that any process running in a window continues to run when the window is closed and that a newly opened window looks just as it did when it was closed (allowing for any changes a running process may have made in the interval).

Fig. 5.4 *Icons*



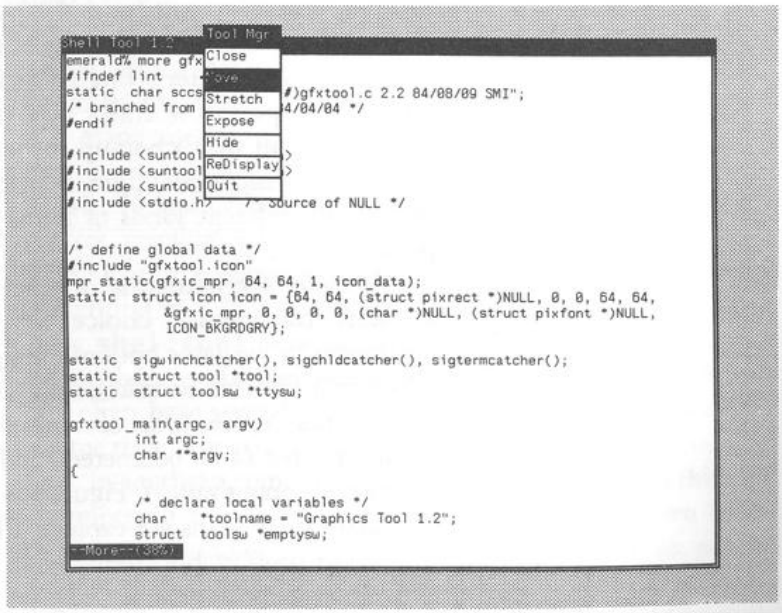
When a tool is no longer needed, you exit the tool via an operation it provides, or you can use the standard **quit** command provided by the tool manager. In either case all resources associated with the tool are released. (By the way, you can *start* a tool in several ways. If you enter the tool's name in a file called **.suntools**, the **suntools** program will start it automatically when it brings up the SunWindows environment. Entering the tool's name in a file called **.rootmenu** allows you to select the tool from the **root menu**, the menu that pops up from the gray background that underlies the windows. You can also start a tool by typing its name to a **shelltool** as you would an ordinary utility.)

5.1.5 Menus

Tools can accept commands in several ways. A tool might, for example, let you type commands into a subwindow as you type commands to the shell. This approach requires you to both remember the commands and be able and willing to type. Alternatively, a tool can provide a **pop-up menu** of commands you can select with the mouse. Thus, instead of having to type a command from memory, you can select an item from a visible array of choices. Pop-up menus have the advantage of occupying no screen space when you are not using them.

Popping up menus and selecting items from them is a natural and fast action. If you position the mouse cursor anywhere on a window frame and hold down the right mouse button, you pop up the tool manager menu shown in Figure 5.5.[5] When you pop up a menu the mouse cursor is automatically re-oriented to point to the first item in the menu, which is also highlighted in reverse video. As you move the mouse cursor down the menu (while still holding down the right button), the next item is highlighted. To select a highlighted item you click (press and release) the left mouse button. When you release the right mouse button the menu disappears and the mouse cursor returns to its former position and orientation. Note that having popped up a menu, you can "choose not to choose" any of the commands displayed. To do so, you simply move the cursor away from the menu and release the right button.

Fig. 5.5 *Tool Manager Menu*



A few more notes on menus:

5. The tool manager menu also pops up when you hold the right button down with the mouse cursor over an icon. The **close** command shown in Figure 5.5 is replaced by an **open** command, however.

- The menu that appears when you press the right mouse button depends on the location of the mouse cursor, that is, whether it is on a window frame, or in a subwindow, or on the gray background. Each subwindow can present a different menu in addition to the standard tool manager menu available in the window frames and the root menu available in the background.
- When a subwindow offers many commands, it can pop up a **menu stack** rather than a single menu. With the mouse you can quickly select one of the menus in the stack and then an item in the menu.
- As you become familiar with a tool, popping up a menu can become a bit of a nuisance. In cases where you know in advance what you want to do, you could save a little time by short-circuiting the menu item selection procedure. **Accelerators** let you do just that; they effectively make mouse buttons behave like conventional function keys. The tool manager provides two accelerators. Clicking the left mouse button exposes a hidden window or opens a closed one. Holding down the middle button moves a window. Subwindows may provide their own accelerators.

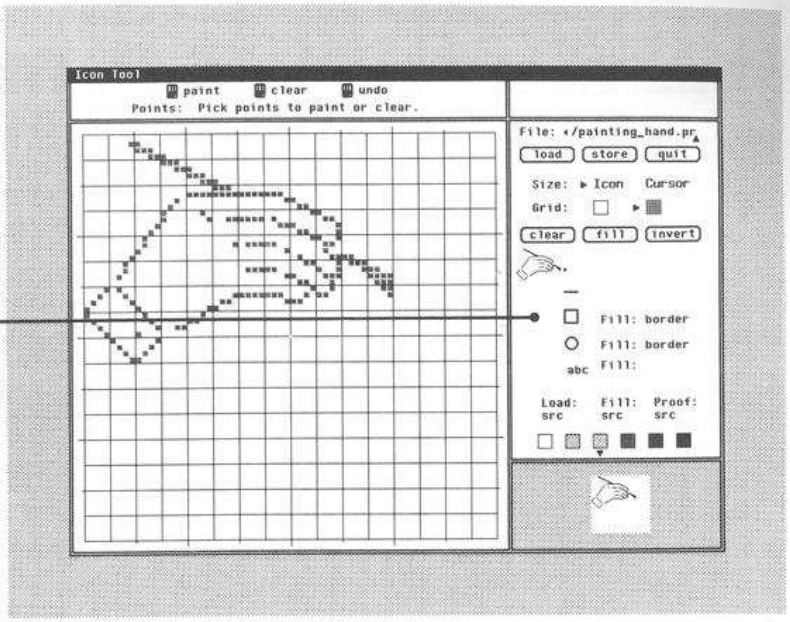
5.1.6 Panels

A **panel** is a permanently visible alternative to a menu. It presents a monitoring/controlling surface like an aircraft or automobile instrument panel, or the front panel of a (perhaps elderly) computer. Like a menu, a panel displays an array of **items**; an item can be labeled with either a character string or a picture. You select an item by positioning the mouse cursor over it and clicking the left mouse button. (Recall that a pop-up menu item is selected in the same way.) This simple paradigm is made extremely versatile by the number of different kinds of items that may be displayed on a panel.

Among the items that a panel can display are **buttons** (for invoking commands), **choices** (typically a list of permissable parameters or available options), **toggles** (a toggle has two states, selecting it switches its state), **text** (for filling in variable data such as file names), **cycles** (a list of values that selection makes "roll over" in the manner of an odometer), and **sliders** (for picking a value from a continuous range). Figure 5.6 shows the panel displayed by **`icontool`**; its items are explained later in this section.

Panels can comfortably coexist with pop-up menus; for example, frequently used commands may be permanently displayed on a panel, while less-used commands are provided in a pop-up menu. Menus may also be used to explain the items in panels. Tool programmers can use both panels and menus according to application needs and preferences.

Fig. 5.6 *A Panel*



5.1.7 Standard Tools

While many users will write their own tools, all will use some of the standard tools that are supplied with the SunWindows system. These standard tools are described in this section.

5.1.7.1 clocktool

The simplest tool, **clocktool** consists of a single subwindow that displays the time of day each minute or each second. When its window is open, **clocktool** mimics a digital watch; conversely, **clocktool**'s icon is an analog dial, with an optional sweep second hand. Most icons are static images; **clocktool**'s, though, is dynamic: the clock hands advance with time. Many users keep a **clocktool** icon in a corner of their screen.

5.1.7.2 shelltool

shelltool consists of a single subwindow that emulates a character terminal running a shell. The emulated terminal is "smart" in that it supports escape sequences for operations such as highlighting text. The keystrokes you type when the mouse cursor is in a **shelltool** window are simply passed to the program running there — the shell or the utility you started with the shell. Users often have several **shelltool** windows on their screen at the same time. For example, in one window you may be running an editor, in another a compile may be under way, while in a third you are composing an electronic mail message. You can even have a **shelltool** window that is remotely logged in to another machine; in this way you can execute a command (**tip**, which needs a modem that your machine may not have, is a good example) on another machine simply by moving the mouse cursor to the window and typing.

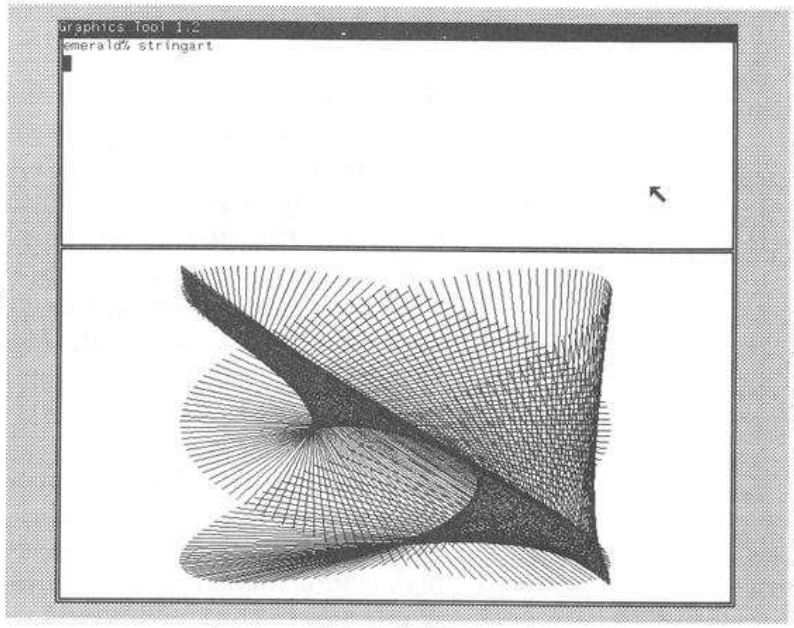
shelltool provides a pop-up menu whose commands are **select** and **stuff**. These commands work like the "cut and paste" operations provided by many editors except that they copy text rather than moving it. They let you, in a limited but useful way, edit a shell session with the mouse. Suppose, for example, you

want to browse through a file with the **more** utility, but you don't remember the exact name of the file. You can get the name of the file displayed on the screen with the **ls** command; then you can type **more**; then you can **select** the file name from screen and **stuff** it next to your **more** command and press the return key. Importantly, you can **select** text from one **shelltool** window and **stuff** the selection into another **shelltool** window.

5.1.7.3 graphicstool

By emulating a character terminal, **shelltool** allows a UNIX application that has been written with no knowledge of the SunWindows system to nevertheless run in a window. **graphicstool** does the same for existing graphics programs. As Figure 5.7 shows, a **graphicstool** window consists of two subwindows. The upper window works like a **shelltool** window; you start the graphics program by typing its name and arguments as a command to the shell. The program's graphical output is automatically directed to the lower window.

Fig. 5.7 **graphicstool** *Window*



5.1.7.4 icontool

By default, a tool's icon is simply a square with the tool's name on it. **icontool** (see Figure 5.8) is a bit map editor that you can use to design a custom icon that symbolizes the tool. **icontool** can also be used to design custom mouse cursors and pictorial labels for panel items.

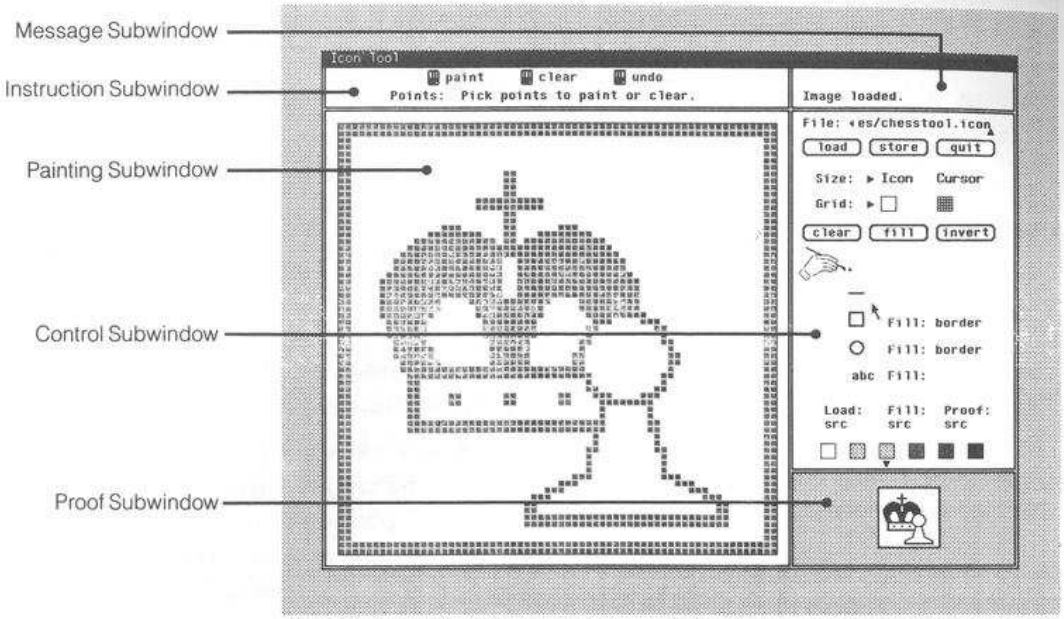
icontool's subwindows are described below:

Instruction Using a combination of graphics and text, the upper left subwindow gives instructions for painting with the mouse in the currently selected the painting mode.

Message The upper right window displays status and error messages; for example, trying to load an non-existent icon file will produce an error message in this window.

Control The control subwindow on the right is a panel that

Fig. 5.8 **icontool** *Window*



you use to select **icontool** options and operations. At the top of the panel is a text item into which you type the name of the file you want to edit. Under this are buttons for loading and storing this file and for quitting the tool. The next two items let you choose the size of the image you wish to paint (either icon size, 64x64 pixels, or cursor size, 16x16 pixels), and whether you want to paint on a plain or gridded canvas. Buttons are provided for clearing, filling, or inverting all the pixels on the canvas. **icontool** has five painting modes: point (pixel), line, box, circle, and text. The painting hand image marks the current choice; selecting a different item moves the hand and displays new instructions in the instruction subwindow. The choices at the bottom of the panel allow you to perform raster operations on your image. (Raster operations are bit-mapped graphics functions that allow you to create new images from the logical combination (for example, AND, OR, or XOR) of other images.) This facility is very useful when you are creating a cursor, since it lets you see how the cursor will look when it is moved over different backgrounds, and which raster operation will give the cursor the best contrast.

Painting To the left of the control subwindow is the painting subwindow in which you change pixels with the mouse. In point mode, for example, you paint a new pixel by clicking the left mouse button and you clear a painted pixel by clicking the middle mouse button. You can undo errors or experiments by clicking the right mouse button.

Proof The painting subwindow magnifies your canvas so you can manipulate individual pixels. The proof subwindow shows the image you are drawing at its actual display size and resolution.

5.1.7.5 fonttool

fonttool is another bit map editor that lets you create your own fixed-width bit-mapped font for display on the Sun screen. Figure 5.9 shows a typical **fonttool** window. Within it are four subwindows, described below:

Message **fonttool** writes prompts, warning messages (such as when you quit the tool without first saving an updated font), and so on, in this window.

Control In this panel, you specify font attributes, the font file you want to edit, the kind of drawing operations you wish to perform, and so on.

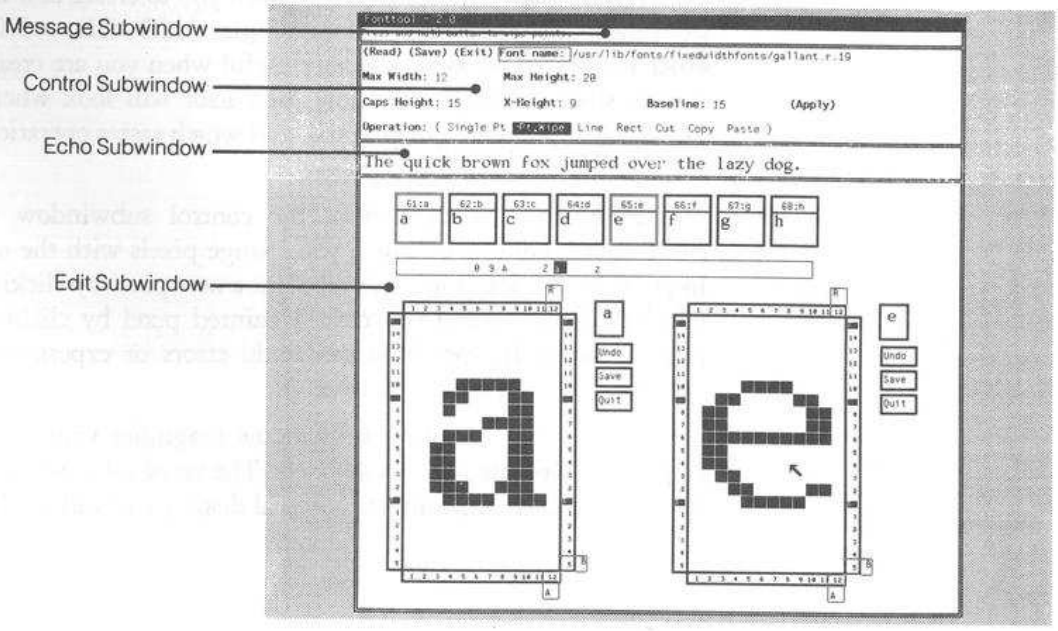
Echo Typing into this subwindow shows how font characters look when strung together.

Edit This is the window where you do your real work. Eight edit buttons show eight adjacent font characters. On each button is displayed the hexadecimal position of the character and its ASCII representation, if there is one (a font may contain up to 255 characters, but there are ASCII representations for only the first 126). Each button also shows the character's bit map displayed at screen resolution.

A different segment of the font can be displayed in the edit buttons by moving (with the mouse) the slider below them. The width of the slider bar represents the complete set of 255 characters; the current position of the slider is shown by the black box. The relative locations of key reference points, namely the numbers and the capital and lower case letters, are shown on the slider bar to make it easier to find the section of the font you want to work with next.

Below the slider bar are the edit pads. You make an edit pad for a character by clicking a mouse button on the corresponding edit button. (In response, the edit button's border is changed from a

Fig. 5.9 **fonttool** *Window*



solid bold line to a double line.) Next to the edit pad are a proof window, which shows the character at screen resolution as you have edited it, and three control buttons. By clicking a mouse button on these you can undo your last operation, save the edited bit map into the font file, and quit editing the character, which makes the edit pad disappear.

In the edit pad, each bit of the character is shown by a black cell. You can paint new black cells and change black cells to white in several modes, as specified in the "operation" section of the control subwindow. You can also change drawing modes by popping up a menu in the edit window; this menu has the same operations shown in the control window. In Figure 5.9 the "point wipe" operation has been selected; within an edit pad pressing a mouse button down will draw a black cell and will draw more black cells as you move the mouse cursor with the button held down.

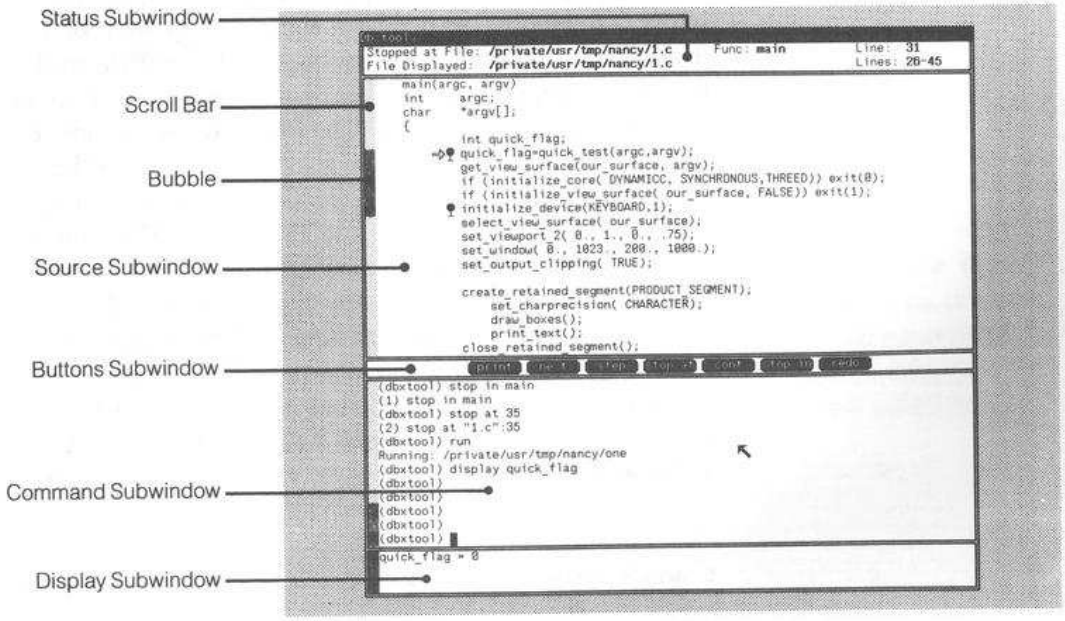
Sliders on the edges of the edit pad labeled "R," "B," and "A," can be moved to change the right edge, bottom edge, and horizontal advance of the character respectively. For fixed-width fonts these are normally identical for every character.

5.1.7.6 dbxtool

dbxtool is a window-based interface to the **dbx** debugger described in the utilities chapter. An outstanding tool in its own right, **dbxtool** also illustrates how the SunWindows system can be used to add new graphical interfaces to existing software.

Figure 5.10 shows a typical **dbxtool** window. It is divided into five functionally specialized subwindows. These subwindows are all adjustable in size. They come up in a default configuration or you can make them come up as you wish by placing instructions in a **.dbxinit** file in your current or home directory. The subwindows are described below.

Fig. 5.10 **dbxtool** *Window*



Status A panel subwindow whose items show the file being debugged, the function in the file that is displayed, the line numbers shown in source subwindow, and so on.

Source Tracks the source code being debugged, showing locations of breakpoints ("stop signs" in Figure 5.10), and the point where execution is stopped (hollow arrow). As the program runs, the text in this window keeps step automatically.

Buttons A panel subwindow whose button items are frequently used **dbx** commands. You can add and delete buttons as you like, according to the commands you use most; **dbxtool** rearranges the buttons automatically in the subwindow. You execute a command by selecting an item in the source subwindow (a line, a variable name, or the like) and then selecting the appropriate command button. For example, suppose you want to see the value of a variable called **index**. To start the selection you move the mouse cursor to the **i** and click the left mouse button. As you move the mouse cursor to the right, **dbxtool** highlights characters showing you what you are selecting. At the **x** you terminate the selection by clicking the middle button.[6] Having selected the text, you then select the "print" button on the screen by placing the mouse cursor over it and clicking the left mouse button.

Command Displays command and program output and supplements the buttons subwindow by accepting all **dbx** commands. Provides a select and stuff menu functionally similar to **shelltool**'s.

Display Displays the values of selected variables whenever execution stops.

The source, command and display subwindows are equipped with **scroll bars**. In each scroll bar is a dark grey "bubble" that shows the relative position and extent of the file segment displayed in the subwindow. For example, in Figure 5.10, the bubble in the source subwindow shows that about one-quarter of the file is displayed, and that the segment displayed is near the middle of the file. You manipulate a scroll bar by placing the mouse cursor over it and clicking a mouse button. The left button scrolls forward in the file; the line next to the mouse cursor is brought to the top of the subwindow. The right button scrolls backwards; it brings the top line down to a position next to the cursor. The middle button scrolls to an absolute location anywhere in the file; it considers the mouse cursor's position as locating a line in the file and it brings that line to the top of the subwindow. For example, to go to the middle of the file, you place the mouse cursor halfway down the scroll bar and click the middle button. Note that for scrolling, the text in the display and command subwindows is treated as if it were a file, so you can review previously entered commands or results at

6. Actually, **dbxtool** lets you be a little "sloppier" when selecting a simple name like this. If you select any two points in the string **dbxtool** will "expand" the selection to the full word.

any time.

5.2 Programming with the SunWindows System

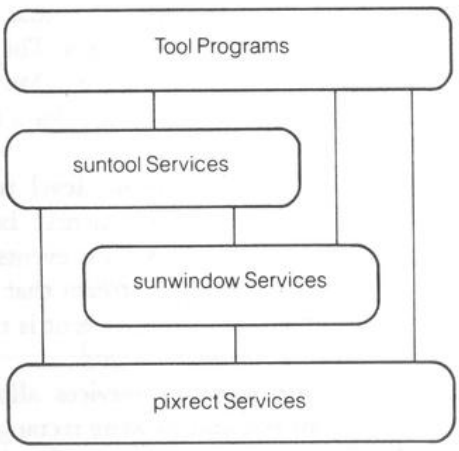
To the programmer, the SunWindows system is a versatile "parts kit" from which many different kinds of user interfaces can be constructed. Most people, though, use the SunWindows system to write tools like those described in the first part of this chapter, so this section describes the SunWindows system as it appears to a tool programmer.

The SunWindows system fully supports the capabilities of color workstations. However, to simplify the discussion, this section only describes the facilities for monochrome displays. In general, tools written for color workstations produce acceptable results on monochrome workstations and vice versa.

5.2.1 SunWindows Structure and Facilities

The SunWindows facilities are embodied in libraries of procedures (technically C functions),[7] plus supporting data structure declarations and macros. The libraries are arranged in three levels as illustrated in Figure 5.11. As shown in the figure, each of the higher levels builds on the facilities provided by the lower levels. A tool is free to use the facilities of all levels, though most of its calls will be to **suntool** procedures.

Fig. 5.11 *SunWindows Facility Structure*



Adding to the flexibility of the system, many of the facilities provided at each level can be extended or replaced. You could, for example, replace the entire **suntool** level with an equivalent that supported tools with tiled, rather than overlapping, windows. You can also integrate your own display or your own pointing device into the SunWindows system.

7. Presently, tools must be written in the C language.

5.2.1.1 pixrects Level

Where the UNIX system provides device-independent I/O operations on streams, the SunWindows **pixrects** level provides device-independent operations on pixels grouped into rectangles called **pixrects**. The notion of a rectangle of pixels is very versatile: a pixrect may represent a single character, an image such as a cursor or an icon, or the entire display screen. Importantly, pixrect operations work identically on all pixrects, whether they are currently displayed or are structures in memory. Similarly, the pixels in a pixrect are addressed in exactly the same way, regardless of the addressing convention employed by the underlying display or memory. For example, display locations are typically addressed as X-Y displacements from an origin, while memory is addressed as a linear array; pixrects hide these differences from your software. The uniformity of pixrects makes it easy to build and save display images in memory and simplifies the job of integrating a new display device into the SunWindows system. Pixrect operations include single pixel access; color map access; drawing of patterns, polygons, curve bounded regions, text and vectors; and basic raster operations involving two and three pixrects. (Raster operations combine pixrects using logical operations such as AND, OR, and XOR.)

5.2.1.2 sunwindow Level

From the simple rectangles supported by the **pixrects** level, the **sunwindow** level builds a system of multiple overlapping windows. These windows may be read and written concurrently by multiple processes; the **sunwindow** level coordinates their interaction so that each process can proceed without interfering with the others. The **sunwindow** level also maintains a record of the areas of a window that are hidden and automatically clips any output destined for a hidden area to prevent overwriting the hiding window.

The **sunwindow** level tracks the mouse with a cursor. It defines mouse movements, button pushes, and keystrokes as **events**. It captures these events as they occur and combines them into a single input stream that reflects the sequence in which the events occurred. Each event is timestamped so a tool can compute the time between events.

sunwindow services allow you to determine a window's current size and to write rectangles, vectors, text, and so on, into a window without concern for clipping. It also lets you determine where the cursor is, move it, and change its image.

5.2.1.3 suntool Level

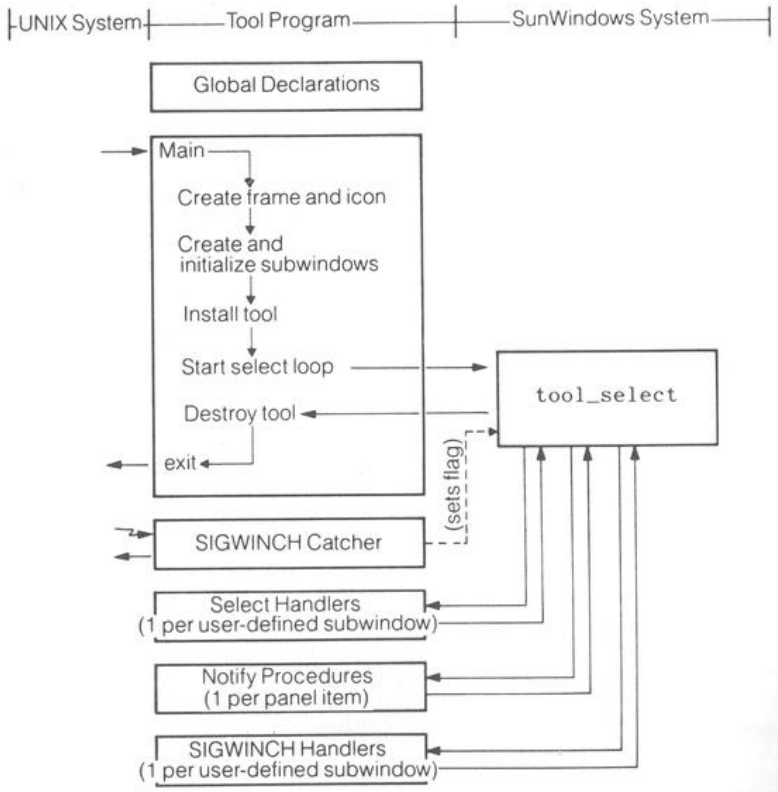
As the **sunwindow** level built windows from pixrects, the **suntool** level, in combination with your code, turns windows into tools — complete applications with window-based user interfaces. In the **suntool** level is the tool manager, the executive framework into which you fit your application. The tool manager provides the border, namestripe, and icon for your tool, lays out subwindows within the frame, displays the tool manager menu on request, and implements its commands. It provides packages you can use to create pop-up menus and standard and user-defined subwindows. You can also use its facilities to obtain access to the

full screen when necessary, and to select data from one window and stuff it into another.

5.2.2 Anatomy of a Tool

Although tools vary widely in their function and their operation, all conform to the basic organization shown in Figure 5.12. A tool is an ordinary program that observes conventions defined by the **suntool** level and calls upon services provided by that level and possibly the lower levels. Within this framework a tool is much like any other UNIX program: it can spawn child processes, pipe data to and from utilities, and so on. However, a tool is inherently dependent on the mouse and keyboard for input and upon the screen for output; therefore, the UNIX system standard I/O conventions do not apply to a tool, (except for writing messages to the standard error if the tool cannot start).

Fig. 5.12 *Tool Program Anatomy and Control Flow*



5.2.2.1 Getting Started

Before a tool is invoked, the **suntools** program, which initializes the SunWindows environment, must be running. As discussed earlier in the chapter, a tool can be invoked automatically by **suntools**, from the root menu, or by typing its name to a **shelltool**. However it is invoked, control enters at the tool's main procedure. Like any utility, the tool can accept command line arguments, can look in a predefined file for startup parameters, and can read shell environment variables.

5.2.2.2 Building the Window Frame

Having completed its startup housekeeping, the tool program proceeds to define the objects that will represent it on the screen. It does so by calling **suntool** procedures (for simplicity these are not shown in Figure 5.12). First the tool defines its window frame and its icon by calling **tool_make**. This procedure takes as parameters a list of attributes that modify the default attributes of a window frame and its icon. Some of these attributes include the size of the frame, the text in the namestripe, and whether the tool is to come up with its window open or closed. For example, if you do not supply an icon for the tool, the default icon (a square with the tool's name inside) is used.

5.2.2.3 Building Subwindows

Next the tool identifies and initializes its subwindows. Supplied with the SunWindows system are standard subwindow packages, described shortly. If a standard subwindow does not provide the functions you need, you can create a user-defined subwindow. To create a user-defined subwindow, you call **tool_createsubwindow**.

To create an instance of a standard subwindow, you call a procedure dedicated to creating that subwindow (for example, **msgsw_createtoolsubwindow** to create a message subwindow). This procedure in turn calls **tool_createsubwindow**.

The most frequently used standard subwindows are described below:

Message A message subwindow simply displays a text string. When you create the subwindow, you specify the font in which to display messages and the initial text string. Later you can write a different string into the window by calling **msgsw_setstring**.

Terminal Emulator This subwindow emulates the basic Sun character terminal. An associated SunWindows service called **ttysw_fork** allows you to fork a process that runs a program in the subwindow, connnecting its standard input, output, and error streams to the subwindow. Often the program run in this fashion is a shell, providing a conventional UNIX system interface inside a subwindow.

Graphics A graphics subwindow provides an environment for a graphics program in somewhat the same way that a terminal emulator provides an environment for a program accustomed to running with a display terminal.

Panel Panels were described earlier in the chapter. For each item you define in a panel, you can name a **notification procedure** (described more fully shortly) that you want called when the user selects the item. The subwindow software takes care of correlating mouse position with item locations and providing appropriate visual feedback for selections.

You can place subwindows where you wish in your tool's window, or you can let the system tile them over the available area. Starting with the first subwindow defined, the tiling algorithm places each successive subwindow as high and as left as it will go. In this way the subwindows are laid out in the customary English reading sequence.

5.2.2.4 Installing the Window

With the window frame and its subwindows defined, the tool is ready to have its window painted on the display and to begin normal processing. To do this the tool calls **tool_install** which adds the window to the database of windows presently being managed. One of the things **tool_install** does is send a SIGWINCH signal (described shortly) to the tool, telling it to draw itself on the screen. When **tool_install** returns, the tool is ready to begin normal processing.

5.2.2.5 Responding to Events

Normal tool operation commences when the tool's main procedure calls **tool_select**. This procedure takes its name from the 4.2BSD **select** system call described earlier in this report. It is **tool_select** and the **select** call that make a tool **event-driven**; that is, the tool is quiescent until the user directs input to it with the mouse or keyboard.[8] The tool responds to the event and then returns to its dormant state.

tool_select essentially runs in a loop that does not terminate until the user quits the tool, typically via the tool manager's **quit** command. This loop uses the **select** call to wait for an event directed to the tool's window. If no event is available, the routine (and thus the process running the tool) blocks until input becomes available. A tool can set masks to filter out events of no interest; for example, a tool that does not use the keyboard can cause all keystrokes made in its window to be ignored. When **select** returns, **tool_select** identifies the subwindow in which the event occurred and calls the **select handler** you named when you created the subwindow. Standard subwindows have their own select handlers so you need to write select handlers only for user-defined subwindows.

Of course, what a select handler does when invoked is entirely application-specific. In general, it will call **input_readevent** to get the event that caused it to be called and based on this event take the appropriate action and return to **tool_select**.

Many SunWindows facilities are provided to assist select handlers. One of them is the pop-up menu package. To define a pop-up menu for a subwindow, the select handler declares a structure containing the menu items. Having been selected, the handler calls **input_readevent**. If the event returned is "right mouse button down," the select handler calls **menu_display**, passing the menu structure. This procedure pops up the menu on the screen, and highlights items as the user moves the mouse cursor up or down the menu. When the user releases the right button, **menu_display** makes the menu disappear and returns, identify-

8. The expiration of a timer can also be designated as an event.

ing the item selected, if any. Now the handler can execute the user's command and return to **tool_select**.

5.2.2.6 Handling Panel Notifications

A panel item's notify procedure is similar to a select handler, but simpler because of the work done by the panel software. At entry to your notification procedure you know that the user has selected a panel item, and you have only to provide the appropriate response. If, instead of using a panel, you build an equivalent with a user-defined subwindow, your select handler must query the mouse position, compute the item selected, highlight the selection, and so on.

5.2.2.7 Handling Window Changes

At any time a user may stretch the tool's window, open its icon, or expose a portion of the window that had been hidden by an overlapping window. In each of these cases, some portion of the window's image must be regenerated and redisplayed. Such an area is called **damage**. Damage does not occur when the user hides all or part of a window. The SunWindows system automatically clips the window image so that only the unhidden portion is displayed.

Damage inherently occurs asynchronously to the execution of the tool's process, since it is instigated by the user, whose actions are unpredictable. Since damage occurs asynchronously, the SunWindows system notifies a tool asynchronously, by sending the tool a UNIX signal, called SIGWINCH (window change).

As with any signal, the tool catches a SIGWINCH with a signal handler (or catcher), a procedure the operating system invokes in the manner of an interrupt handler. While the tool must eventually repair the damage to its window, the SIGWINCH catcher is not the place to do it. Because the SIGWINCH catcher is invoked asynchronously, it cannot, as a rule, assume that the window is in a consistent state. For example, the tool's own code may have been updating the window when the damage was inflicted. Therefore the SIGWINCH handler does nothing but set a flag in **tool_select** by calling **tool_sigwinched**.

Besides invoking the tool's SIGWINCH catcher, occurrence of a SIGWINCH terminates **tool_select**'s **select** call if it is blocked on such a call. At that time **tool_select** notices that its **sigwinchpending** flag is set. In response, it calls all the subwindow SIGWINCH handlers; it is these procedures that actually effect the repairs. If, when a SIGWINCH occurs, **tool_select** is not blocked on a **select** (typically meaning that one of the select handlers is running) it acts on the flag before it issues the next **select**. The result is that asynchronously occurring damage is repaired synchronously with normal processing.

SIGWINCH handlers are supplied with the standard subwindows, making them self-repairing.[9] You provide a SIGWINCH handler for any user-defined subwindows you create. A typical SIGWINCH handler makes calls to the **sunwindow** level to set clipping information and regenerates the full subwindow area from the current values of its variables. The **sunwindow** level redisplays only the damaged area to minimize display flash.

5.2.2.8 Cleaning Up

Normal operation ends when the user quits the tool, typically by selecting the **quit** command from the tool manager menu. When this happens, **tool_select** returns to the tool's main procedure. The tool then cleans up, calls **tool_destroy** to remove its window from the display and free the associated resources, and finally exits in the normal way.

9. A graphics subwindow is a special case. Like a user-defined subwindow, the SunWindows system has no idea of what should be displayed in a graphics window, so it cannot repair the subwindow. However, the tool may ask that an exact image of the graphics subwindow be continuously retained and updated in memory, allowing damage to be repaired by simply calling **pw_repairretained**.

Sun Microsystems, Inc.

Reader Comments

We at Sun Microsystems wish to provide the best possible information on our UNIX environment. We therefore encourage your comments and suggestions by asking you to complete the questions below.

1 **Content**

Were your expectations met? If not, please indicate what you think should be added or deleted.

2 **Layout and Style**

Did you find the information well-organized and easy to follow? If not, please indicate how it may be improved.

3 **Technical Errors**

Please list errors of fact by page number and actual text of the error.

4 **Typographical Errors**

Please list typographical errors by page number and actual text of the error.

Name _____ Date _____

Title _____ Dept. _____

Company _____ Division _____

Address _____ M/S _____

City, State _____ Zip _____

Telephone ( ) _____ Ext. _____