-

The UNIX Time-Sharing System

Dennis M. Ritchie
Ken Thompson

Bell Laboratories
Murray Hill, N. J. 07974

ABSTRACT

UNIX is a general-purpose, multi-user, interactive operating system for the Digital Equipment Corporation PDP-11/40, 11/45 and 11/70 computers. It offers a number of features seldom found even in larger operating systems, including

1. A hierarchical file system incorporating demountable volumes,
2. Compatible file, device, and inter-process I/O,
3. The ability to initiate asynchronous processes,
4. System command language selectable on a per-user basis,
5. Over 100 subsystems including a dozen languages.

This paper discusses the nature and implementation of the file system and of the user command interface.

1. Introduction

There have been three versions of UNIX. The earliest version (circa 1969-70) ran on the Digital Equipment Corporation PDP-7 and -9 computers. The second version ran on the unprotected PDP-11/20 computer. This paper describes only the PDP-11/40, /45 and /70[1] system, since it is more modern and many of the differences between it and older UNIX systems result from redesign of features found to be deficient or lacking.

Since PDP-11 UNIX became operational in February, 1971, about 100 installations have been put into service; they are generally smaller than the system described here. Most of them are engaged in applications such as the preparation and formatting of patent applications and other textual material, the collection and processing of trouble data from various switching machines within the Bell System, and recording and checking telephone service orders. Our own installation is used mainly for research in operating systems, languages, computer networks, and other topics in computer science, and also for document preparation.

This is a revised version of an article appearing in the Communications of the ACM, Volume 17, Number 7 (July 1974) pp. 365-375. That article is a revised version of a paper presented at the Fourth ACM Symposium on Operating Systems Principles, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, October 15-17, 1973.

-

Perhaps the most important achievement of UNIX is to demonstrate that a powerful operating system for interactive use need not be expensive either in equipment or in human effort: UNIX can run on hardware costing as little as $40,000, and less than two man-years were spent on the main system software. Yet UNIX contains a number of features seldom offered even in much larger systems. Hopefully, however, the users of UNIX will find that the most important characteristics of the system are its simplicity, elegance, and ease of use.

Besides the system proper, the major programs available under UNIX are

assembler,
text editor based on QED[2],
linking loader,
symbolic debugger,
compiler for a language resembling BCPL[3] with types and structures (C),
interpreter for a dialect of BASIC,
phototypesetting and equation setting programs
Fortran compiler,
Snobol interpreter,
top-down compiler-compiler (TMG[4]),
bottom-up compiler-compiler (YACC),
form letter generator,
macro processor (M6[5]),
permuted index program.

There is also a host of maintenance, utility, recreation and novelty programs. All of these programs were written locally. It is worth noting that the system is totally self-supporting. All UNIX software is maintained under UNIX; likewise, this paper and all other UNIX documents were generated and formatted by the UNIX editor and text formatting program.

2. Hardware and software environment

The PDP-11/45 on which our UNIX installation is implemented is a 16-bit word (8-bit byte) computer with 112K bytes of core memory; UNIX occupies 53K bytes. This system, however, includes a very large number of device drivers and enjoys a generous allotment of space for I/O buffers and system tables; a minimal system capable of running the software mentioned above can require as little as 64K bytes of core altogether.

Our PDP-11 has a 1M byte fixed-head disk, used for file system storage and swapping, four moving-head disk drives which each provide 2.5M bytes on removable disk cartridges, and a single moving-head disk drive which uses removable 40M byte disk packs. There are also a high-speed paper tape reader-punch, nine-track magnetic tape, and DECtape (a variety of magnetic tape facility in which individual records may be addressed and rewritten). Besides the console typewriter, there are 30 variable-speed communications interfaces attached to 100-series datasets and a 201 dataset interface used primarily for spooling printout to a communal line printer. There are also several one-of-a-kind devices including a Picturephone® interface, a voice response unit, a voice synthesizer, a phototypesetter, a digital switching network, and a satellite PDP-11/20 which generates vectors, curves, and characters on a Tektronix 611 storage-tube display.

The greater part of UNIX software is written in the above-mentioned C language[6]. Early versions of the operating system were written in assembly language, but during the summer of 1973, it was rewritten in C. The size of the new system is about one third greater than the old. Since the new system is not only much easier to understand and to modify but also includes many functional improvements, including multiprogramming and the ability to share reentrant code among several user programs, we considered this increase in size quite acceptable.

3. The File system

The most important role of UNIX is to provide a file system. From the point of view of the user, there are three kinds of files: ordinary disk files, directories, and special files.

3.1 Ordinary files

A file contains whatever information the user places on it, for example symbolic or binary (object) programs. No particular structuring is expected by the system. Files of text consist simply of a string of characters, with lines demarcated by the new-line character. Binary programs are sequences of words as they will appear in core memory when the program starts executing. A few user programs manipulate files with more structure; for example, the assembler generates, and the loader expects, an object file in a particular format. However, the structure of files is controlled by the programs which use them, not by the system.

3.2 Directories

Directories provide the mapping between the names of files and the files themselves, and thus induce a structure on the file system as a whole. Each user has a directory of his own files; he may also create subdirectories to contain groups of files conveniently treated together. A directory behaves exactly like an ordinary file except that it cannot be written on by unprivileged programs, so that the system controls the contents of directories. However, anyone with appropriate permission may read a directory just like any other file.

The system maintains several directories for its own use. One of these is the *root* directory. All files in the system can be found by tracing a path through a chain of directories until the desired file is reached. The starting point for such searches is often the root. Another system directory contains all the programs provided for general use; that is, all the *commands*. As will be seen, however, it is by no means necessary that a program reside in this directory for it to be executed.

Files are named by sequences of 14 or fewer characters. When the name of a file is specified to the system, it may be in the form of a *path name,* which is a sequence of directory names separated by slashes "/" and ending in a file name. If the sequence begins with a slash, the search begins in the root directory. The name */alpha/beta/gamma* causes the system to search the root for directory *alpha,* then to search *alpha* for *beta,* finally to find *gamma* in *beta*. *Gamma* may be an ordinary file, a directory, or a special file. As a limiting case, the name "/" refers to the root itself.

A path name not starting with "/" causes the system to begin the search in the user's current directory. Thus, the name *alpha/beta* specifies the file named *beta* in subdirectory *alpha* of the current directory. The simplest kind of name, for example *alpha*, refers to a file which itself is found in the current directory. As another limiting case, the null file name refers to the current directory.

The same non-directory file may appear in several directories under possibly different names. This feature is called *linking;* a directory entry for a file is sometimes called a link. UNIX differs from other systems in which linking is permitted in that all links to a file have equal status. That is, a file does not exist within a particular directory; the directory entry for a file consists merely of its name and a pointer to the information actually describing the file. Thus a file exists independently of any directory entry, although in practice a file is made to disappear along with the last link to it.

Each directory always has at least two entries. The name "**.**" in each directory refers to the directory itself. Thus a program may read the current directory under the name "**.**" without knowing its complete path name. The name "**..**" by convention refers to the parent of the directory in which it appears, that is, to the directory in which it was created.

The directory structure is constrained to have the form of a rooted tree. Except for the special entries "**.**" and "**..**", each directory must appear as an entry in exactly one other, which is its parent. The reason for this is to simplify the writing of programs which visit subtrees of the directory structure, and more important, to avoid the separation of portions of the hierarchy. If arbitrary links to directories were permitted, it would be quite difficult to detect when the last connection from the root to a directory was severed.

3.3 Special files

Special files constitute the most unusual feature of the UNIX file system. Each I/O device supported by UNIX is associated with at least one such file. Special files are read and written just like ordinary disk files, but requests to read or write result in activation of the associated device. An entry for each special file resides in directory */dev,* although a link may be made to one of these files just like an ordinary file. Thus, for example, to punch paper tape, one may write on the file */dev/ppt*. Special files exist for each communication line, each disk, each tape drive, and for physical core memory. Of course, the active disks and the core special file are protected from indiscriminate access.

There is a threefold advantage in treating I/O devices this way: file and device I/O are as similar as possible; file and device names have the same syntax and meaning, so that a program expecting a file name as a parameter can be passed a device name; finally, special files are subject to the same protection mechanism as regular files.

3.4 Removable file systems

Although the root of the file system is always stored on the same device, it is not necessary that the entire file system hierarchy reside on this device. There is a *mount* system request which has two arguments: the name of an existing ordinary file, and the name of a special file whose associated storage volume (e. g. disk pack) should have the structure of an independent file system containing its own directory hierarchy. The effect of *mount* is to cause references to the heretofore ordinary file to refer instead to the root directory of the file system on the removable volume. In effect, *mount* replaces a leaf of the hierarchy tree (the ordinary file) by a whole new subtree (the hierarchy stored on the removable volume). After the *mount*, there is virtually no distinction between files on the removable volume and those in the permanent file system. In our installation, for example, the root directory resides on the fixed-head disk, and the large disk drive, which contains user's files, is mounted by the system initialization program; the four smaller disk drives are available to users for mounting their own disk packs. A mountable file system is generated by writing on its corresponding special file. A utility program is available to create an empty file system, or one may simply copy an existing file system.

There is only one exception to the rule of identical treatment of files on different devices: no link may exist between one file system hierarchy and another. This restriction is enforced so as to avoid the elaborate bookkeeping which would otherwise be required to assure removal of the links when the removable volume is finally dismounted. In particular, in the root directories of all file systems, removable or not, the name “`..`” refers to the directory itself instead of to its parent.

3.5 Protection

Although the access control scheme in UNIX is quite simple, it has some unusual features. Each user of the system is assigned a unique user identification number. When a file is created, it is marked with the user ID of its owner. Also given for new files is a set of seven protection bits. Six of these specify independently read, write, and execute permission for the owner of the file and for all other users.

If the seventh bit is on, the system will temporarily change the user identification of the current user to that of the creator of the file whenever the file is executed as a program. This change in user ID is effective only during the execution of the program which calls for it. The set-user-ID feature provides for privileged programs which may use files inaccessible to other users. For example, a program may keep an accounting file which should neither be read nor changed except by the program itself. If the set-user-identification bit is on for the program, it may access the file although this access might be forbidden to other programs invoked by the given program's user. Since the actual user ID of the invoker of any program is always available, set-user-ID programs may take any measures desired to satisfy themselves as to their invoker's credentials. This mechanism is used to allow users to execute the carefully-written commands which call privileged system entries. For example, there is a system entry invokable only by the “super-user” (below) which creates an empty directory. As indicated above, directories are expected to have entries for “`.`” and “`..`”. The command which creates a directory is owned by the super-user and has the set-user-ID bit set. After it checks its invoker's authorization to create the specified directory, it creates it and makes the entries for “`.`” and “`..`”.

Since anyone may set the set-user-ID bit on one of his own files, this mechanism is generally available without administrative intervention. For example, this protection scheme easily solves the MOO accounting problem posed in

[7].

The system recognizes one particular user ID (that of the ‘‘super-user’’) as exempt from the usual constraints on file access; thus (for example) programs may be written to dump and reload the file system without unwanted interference from the protection system.

3.6 I/O calls

The system calls to do I/O are designed to eliminate the differences between the various devices and styles of access. There is no distinction between ‘‘random’’ and ‘‘sequential’’ I/O, nor is any logical record size imposed by the system. The size of an ordinary file is determined by the highest byte written on it; no predetermination of the size of a file is necessary or possible.

To illustrate the essentials of I/O in UNIX, some of the basic calls are summarized below in an anonymous language which will indicate the required parameters without getting into the complexities of machine language programming. Each call to the system may potentially result in an error return, which for simplicity is not represented in the calling sequence.

To read or write a file assumed to exist already, it must be opened by the following call:

filep = open (name, flag)

Name indicates the name of the file. An arbitrary path name may be given. The *flag* argument indicates whether the file is to be read, written, or ‘‘updated,’’ that is read and written simultaneously.

The returned value *filep* is called a *file descriptor.* It is a small integer used to identify the file in subsequent calls to read, write or otherwise manipulate the file.

To create a new file or completely rewrite an old one, there is a *create* system call which creates the given file if it does not exist, or truncates it to zero length if it does exist. *Create* also opens the new file for writing and, like *open,* returns a file descriptor.

There are no user-visible locks in the file system, nor is there any restriction on the number of users who may have a file open for reading or writing. Although it is possible for the contents of a file to become scrambled when two users write on it simultaneously, in practice difficulties do not arise. We take the view that locks are neither necessary nor sufficient, in our environment, to prevent interference between users of the same file. They are unnecessary because we are not faced with large, single-file data bases maintained by independent processes. They are insufficient because locks in the ordinary sense, whereby one user is prevented from writing on a file which another user is reading, cannot prevent confusion when, for example, both users are editing a file with an editor which makes a copy of the file being edited.

It should be said that the system has sufficient internal interlocks to maintain the logical consistency of the file system when two users engage simultaneously in such inconvenient activities as writing on the same file, creating files in the same directory, or deleting each other’s open files.

Except as indicated below, reading and writing are sequential. This means that if a particular byte in the file was the last byte written (or read), the next I/O call implicitly refers to the first following byte. For each open file there is a pointer, maintained by the system, which indicates the next byte to be read or written. If *n* bytes are read or written, the pointer advances by *n* bytes.

Once a file is open, the following calls may be used.

n = read (filep, buffer, count)

n = write (filep, buffer, count)

Up to *count* bytes are transmitted between the file specified by *filep* and the byte array specified by *buffer.* The returned value *n* is the number of bytes actually transmitted. In the *write* case, *n* is the same as *count* except under exceptional conditions like I/O errors or end of physical medium on special files; in a *read,* however, *n* may without error be less than *count.* If the read pointer is so near the end of the file that reading *count* characters would cause reading beyond the end, only sufficient bytes are transmitted to reach the end of the file; also, typewriter-like devices

-

never return more than one line of input. When a *read* call returns with *n* equal to zero, it indicates the end of the file. For disk files this occurs when the read pointer becomes equal to the current size of the file. It is possible to generate an end-of-file from a typewriter by use of an escape sequence which depends on the device used.

Bytes written on a file affect only those implied by the position of the write pointer and the count; no other part of the file is changed. If the last byte lies beyond the end of the file, the file is grown as needed.

To do random (direct access) I/O it is only necessary to move the read or write pointer to the appropriate location in the file.

location = seek (filep, offset, base)

The pointer associated with *filep* is moved to a position *offset* bytes from the beginning of the file, from the current position of the pointer, or from the end of the file, depending on *base. Offset* may be negative. For some devices (e.g. paper tape and typewriters) seek calls are ignored. The actual offset from the beginning of the file to which the pointer was moved is returned in *location*.

3.6.1 Other I/O calls

There are several additional system entries having to do with I/O and with the file system which will not be discussed. For example: close a file, get the status of a file, change the protection mode or the owner of a file, create a directory, make a link to an existing file, delete a file.

4. Implementation of the file system

As mentioned in §3.2 above, a directory entry contains only a name for the associated file and a pointer to the file itself. This pointer is an integer called the *i-number* (for index number) of the file. When the file is accessed, its i-number is used as an index into a system table (the *i-list*) stored in a known part of the device on which the directory resides. The entry thereby found (the file’s *i-node*) contains the description of the file:

1. its owner;
2. its protection bits;
3. the physical disk or tape addresses for the file contents;
4. its size;
5. time of last modification;
6. the number of links to the file; that is, the number of times it appears in a directory;
7. a bit indicating whether the file is a directory;
8. a bit indicating whether the file is a special file;
9. a bit indicating whether the file is ‘‘large’’ or ‘‘small.’’

The purpose of an *open* or *create* system call is to turn the path name given by the user into an i-number by searching the explicitly or implicitly named directories. Once a file is open, its device, i-number, and read/write pointer are stored in a system table indexed by the file descriptor returned by the *open* or *create.* Thus the file descriptor supplied during a subsequent call to read or write the file may be easily related to the information necessary to access the file.

When a new file is created, an i-node is allocated for it and a directory entry is made which contains the name of the file and the i-node number. Making a link to an existing file involves creating a directory entry with the new name, copying the i-number from the original file entry, and incrementing the link-count field of the i-node. Removing (deleting) a file is done by decrementing the link-count of the i-node specified by its directory entry and erasing the directory entry. If the link-count drops to 0, any disk blocks in the file are freed and the i-node is deallocated.

The space on all fixed or removable disks which contain a file system is divided into a number of 512-byte blocks logically addressed from 0 up to a limit which depends on the device. There is space in the i-node of each file for eight device addresses. A *small* (non-special) file fits into eight or fewer blocks; in this case the addresses of the blocks themselves are stored. For *large* (non-special) files, seven of the eight device addresses may point to indirect blocks each containing 256 addresses for the data blocks of the file. If required, the eighth word is the address of a double-indirect block containing 256 more addresses of indirect blocks. Thus files may conceptually grow to (7+256)·256·512 bytes; actually they are restricted to 16,777,216 (2^{24}) bytes. Once opened, a small file (size 1 to 8 blocks) can be accessed directly. A large file (size 9 to 32768 blocks) requires one additional access to read below logical block 1792 (7·256) and two additional references above 1792.

The foregoing discussion applies to ordinary files. When an I/O request is made to a file whose i-node indicates that it is special, the last seven device address words are immaterial, and the first is interpreted as a pair of bytes which constitute an internal *device name.* These bytes specify respectively a device type and subdevice number. The device type indicates which system routine will deal with I/O on that device; the subdevice number selects, for example, a disk drive attached to a particular controller or one of several similar typewriter interfaces.

In this environment, the implementation of the *mount* system call (§3.4) is quite straightforward. *Mount* maintains a system table whose argument is the i-number and device name of the ordinary file specified during the *mount,* and whose corresponding value is the device name of the indicated special file. This table is searched for each (i-number, device)-pair which turns up while a path name is being scanned during an *open* or *create;* if a match is found, the i-number is replaced by 1 (which is the i-number of the root directory on all file systems), and the device name is replaced by the table value.

To the user, both reading and writing of files appear to be synchronous and unbuffered. That is, immediately after return from a *read* call the data are available, and conversely after a *write* the user's workspace may be reused. In fact the system maintains a rather complicated buffering mechanism which reduces greatly the number of I/O operations required to access a file. Suppose a *write* call is made specifying transmission of a single byte. UNIX will search its buffers to see whether the affected disk block currently resides in core memory; if not, it will be read in from the device. Then the affected byte is replaced in the buffer and an entry is made in a list of blocks to be written. The return from the *write* call may then take place, although the actual I/O may not be completed until a later time. Conversely, if a single byte is read, the system determines whether the secondary storage block in which the byte is located is already in one of the system's buffers; if so, the byte can be returned immediately. If not, the block is read into a buffer and the byte picked out.

The system recognizes when a program has made accesses to sequential blocks of a file, and asynchronously pre-reads the next block. This significantly reduces the running time of most programs while adding little to system overhead.

A program which reads or writes files in units of 512 bytes has an advantage over a program which reads or writes a single byte at a time, but the gain is not immense; it comes mainly from the avoidance of system overhead. A program which is used rarely or which does no great volume of I/O may quite reasonably read and write in units as small as it wishes.

The notion of the i-list is an unusual feature of UNIX. In practice, this method of organizing the file system has proved quite reliable and easy to deal with. To the system itself, one of its strengths is the fact that each file has a short, unambiguous name which is related in a simple way to the protection, addressing, and other information needed to access the file. It also permits a quite simple and rapid algorithm for checking the consistency of a file system, for example verification that the portions of each device containing useful information and those free to be allocated are disjoint and together exhaust the space on the device. This algorithm is independent of the directory hierarchy, since it need only scan the linearly-organized i-list. At the same time the notion of the i-list induces certain peculiarities not found in other file system organizations. For example, there is the question of who is to be charged for the space a file occupies, since all directory entries for a file have equal status. Charging the owner of a file is unfair in general, since one user may create a file, another may link to it, and the first user may delete the file. The first user is still the owner of the file, but it should be charged to the second user. The simplest reasonably fair algorithm seems to be to spread the charges equally among users who have links to a file. The current version of UNIX avoids the issue by not charging any fees at all.

-

4.1 Efficiency of the file system

To provide an indication of the overall efficiency of UNIX and of the file system in particular, timings were made of the assembly of a 8848-line program. The assembly was run alone on the machine; the total clock time was 32 seconds, for a rate of 276 lines per second. The time was divided as follows: 66% assembler execution time, 21% system overhead, 13% disk wait time. We will not attempt any interpretation of these figures nor any comparison with other systems, but merely note that we are generally satisfied with the overall performance of the system.

5. Processes and images

An *image* is a computer execution environment. It includes a core image, general register values, status of open files, current directory and the like. An image is the current state of a pseudo-computer.

A *process* is the execution of an image. While the processor is executing on behalf of a process, the image must reside in core; during the execution of other processes it remains in core unless the appearance of an active, higher-priority process forces it to be swapped out to the fixed-head disk.

The user-core part of an image is divided into three logical segments. The program text segment begins at location 0 in the virtual address space. During execution, this segment is write-protected and a single copy of it is shared among all processes executing the same program. At the first 8K byte boundary above the program text segment in the virtual address space begins a non-shared, writable data segment, the size of which may be extended by a system call. Starting at the highest address in the virtual address space is a stack segment, which automatically grows downward as the hardware's stack pointer fluctuates.

5.1 Processes

Except while UNIX is bootstrapping itself into operation, a new process can come into existence only by use of the *fork* system call:

processid = fork (label)

When *fork* is executed by a process, it splits into two independently executing processes. The two processes have independent copies of the original core image, and share any open files. The new processes differ only in that one is considered the parent process: in the parent, control returns directly from the *fork*, while in the child, control is passed to location *label.* The *processid* returned by the *fork* call is the identification of the other process.

Because the return points in the parent and child process are not the same, each image existing after a *fork* may determine whether it is the parent or child process.

5.2 Pipes

Processes may communicate with related processes using the same system *read* and *write* calls that are used for file system I/O. The call

filep = pipe ( )

returns a file descriptor *filep* and creates an inter-process channel called a *pipe*. This channel, like other open files, is passed from parent to child process in the image by the *fork* call. A *read* using a pipe file descriptor waits until another process writes using the file descriptor for the same pipe. At this point, data are passed between the images of the two processes. Neither process need know that a pipe, rather than an ordinary file, is involved.

Although inter-process communication via pipes is a quite valuable tool (see §6.2), it is not a completely general mechanism, since the pipe must be set up by a common ancestor of the processes involved.

5.3 Execution of programs

Another major system primitive is invoked by

execute (file, arg_1, arg_2, . . . , arg_n)

which requests the system to read in and execute the program named by *file*, passing it string arguments arg_1, arg_2,

. . . , arg_n. All the code and data in the process using *execute* is replaced from the *file*, but open files, current directory, and inter-process relationships are unaltered. Only if the call fails, for example because *file* could not be found or because its execute-permission bit was not set, does a return take place from the *execute* primitive; it resembles a ''jump'' machine instruction rather than a subroutine call.

5.4 Process synchronization

Another process control system call

processid = wait ( )

causes its caller to suspend execution until one of its children has completed execution. Then *wait* returns the *processid* of the terminated process. An error return is taken if the calling process has no descendants. Certain status from the child process is also available.

5.5 Termination

Lastly,

exit (status)

terminates a process, destroys its image, closes its open files, and generally obliterates it. When the parent is notified through the *wait* primitive, the indicated *status* is available to the parent. Processes may also terminate as a result of various illegal actions or user-generated signals (§7 below).

6. The Shell

For most users, communication with UNIX is carried on with the aid of a program called the Shell. The Shell is a command line interpreter: it reads lines typed by the user and interprets them as requests to execute other programs. In simplest form, a command line consists of the command name followed by arguments to the command, all separated by spaces:

command arg_1 arg_2 . . . arg_n

The Shell splits up the command name and the arguments into separate strings. Then a file with name *command* is sought; *command* may be a path name including the ''/'' character to specify any file in the system. If *command* is found, it is brought into core and executed. The arguments collected by the Shell are accessible to the command. When the command is finished, the Shell resumes its own execution, and indicates its readiness to accept another command by typing a prompt character.

If file *command* cannot be found, the Shell prefixes the string */bin/* to *command* and attempts again to find the file. Directory */bin* contains all the commands intended to be generally used.

6.1 Standard I/O

The discussion of I/O in §3 above seems to imply that every file used by a program must be opened or created by the program in order to get a file descriptor for the file. Programs executed by the Shell, however, start off with two open files which have file descriptors 0 and 1. As such a program begins execution, file 1 is open for writing, and is best understood as the standard output file. Except under circumstances indicated below, this file is the user's typewriter. Thus programs which wish to write informative or diagnostic information ordinarily use file descriptor 1. Conversely, file 0 starts off open for reading, and programs which wish to read messages typed by the user usually read this file.

The Shell is able to change the standard assignments of these file descriptors from the user's typewriter printer and keyboard. If one of the arguments to a command is prefixed by ''>'', file descriptor 1 will, for the duration of the command, refer to the file named after the ''>''. For example,

ls

ordinarily lists, on the typewriter, the names of the files in the current directory. The command ls >there

creates a file called *there* and places the listing there. Thus the argument ‘‘>there’’ means, ‘‘place output on *there*.’’ On the other hand,

ed

ordinarily enters the editor, which takes requests from the user via his typewriter. The command

ed <script

interprets *script* as a file of editor commands; thus ‘‘<script’’ means, ‘‘take input from *script*.’’

Although the file name following ‘‘<’’ or ‘‘>’’ appears to be an argument to the command, in fact it is interpreted completely by the Shell and is not passed to the command at all. Thus no special coding to handle I/O redirection is needed within each command; the command need merely use the standard file descriptors 0 and 1 where appropriate.

6.2 Filters

An extension of the standard I/O notion is used to direct output from one command to the input of another. A sequence of commands separated by vertical bars causes the Shell to execute all the commands simultaneously and to arrange that the standard output of each command be delivered to the standard input of the next command in the sequence. Thus in the command line

ls | pr −2 | opr

ls lists the names of the files in the current directory; its output is passed to *pr*, which paginates its input with dated headings. The argument ‘‘−2’’ means double column. Likewise the output from *pr* is input to *opr*. This command spools its input onto a file for off-line printing.

This procedure could have been carried out more clumsily by

ls >temp1
pr −2 <temp1 >temp2
opr <temp2

followed by removal of the temporary files. In the absence of the ability to redirect output and input, a still clumsier method would have been to require the *ls* command to accept user requests to paginate its output, to print in multi-column format, and to arrange that its output be delivered off-line. Actually it would be surprising, and in fact unwise for efficiency reasons, to expect authors of commands such as *ls* to provide such a wide variety of output options.

A program such as *pr* which copies its standard input to its standard output (with processing) is called a *filter*. Some filters which we have found useful perform character transliteration, sorting of the input, and encryption and decryption.

6.3 Command Separators; Multitasking

Another feature provided by the Shell is relatively straightforward. Commands need not be on different lines; instead they may be separated by semicolons.

ls; ed

will first list the contents of the current directory, then enter the editor.

A related feature is more interesting. If a command is followed by ‘‘&’’, the Shell will not wait for the command to finish before prompting again; instead, it is ready immediately to accept a new command. For example,

as source >output &

causes *source* to be assembled, with diagnostic output going to *output;* no matter how long the assembly takes, the Shell returns immediately. When the Shell does not wait for the completion of a command, the identification of the process running that command is printed. This identification may be used to wait for the completion of the command or to terminate it. The “&” may be used several times in a line:

as source >output & ls >files &

does both the assembly and the listing in the background. In the examples above using “&”, an output file other than the typewriter was provided; if this had not been done, the outputs of the various commands would have been intermingled.

The Shell also allows parentheses in the above operations. For example

(date; ls) >x &

prints the current date and time followed by a list of the current directory onto the file *x.* The Shell also returns immediately for another request.

6.4 The Shell as a Command; Command Files

The Shell is itself a command, and may be called recursively. Suppose file *tryout* contains the lines

as source
mv a.out testprog
testprog

The *mv* command causes the file *a.out* to be renamed *testprog. A.out* is the (binary) output of the assembler, ready to be executed. Thus if the three lines above were typed on the console, *source* would be assembled, the resulting program renamed *testprog*, and *testprog* executed. When the lines are in *tryout*, the command

sh <tryout

would cause the Shell *sh* to execute the commands sequentially.

The Shell has further capabilities, including the ability to substitute parameters and to construct argument lists from a specified subset of the file names in a directory. It is also possible to execute commands conditionally on character string comparisons or on existence of given files and to perform transfers of control within filed command sequences.

6.5 Implementation of the Shell

The outline of the operation of the Shell can now be understood. Most of the time, the Shell is waiting for the user to type a command. When the new-line character ending the line is typed, the Shell’s *read* call returns. The Shell analyzes the command line, putting the arguments in a form appropriate for *execute*. Then *fork* is called. The child process, whose code of course is still that of the Shell, attempts to perform an *execute* with the appropriate arguments. If successful, this will bring in and start execution of the program whose name was given. Meanwhile, the other process resulting from the *fork*, which is the parent process, *wait*s for the child process to die. When this happens, the Shell knows the command is finished, so it types its prompt and reads the typewriter to obtain another command.

Given this framework, the implementation of background processes is trivial; whenever a command line contains “&”, the Shell merely refrains from waiting for the process which it created to execute the command.

Happily, all of this mechanism meshes very nicely with the notion of standard input and output files. When a process is created by the *fork* primitive, it inherits not only the core image of its parent but also all the files currently open in its parent, including those with file descriptors 0 and 1. The Shell, of course, uses these files to read command lines and to write its prompts and diagnostics, and in the ordinary case its children_the command programs_inherit them automatically. When an argument with “<” or “>” is given however, the offspring process, just before it performs *execute,* makes the standard I/O file descriptor 0 or 1 respectively refer to the named file. This is easy because, by agreement, the smallest unused file descriptor is assigned when a new file is *open*ed (or *create*d); it is only necessary to close file 0 (or 1) and open the named file. Because the process in which the command program runs simply terminates when it is through, the association between a file specified after “<” or “>” and file descriptor 0 or 1 is ended automatically when the process dies. Therefore the Shell need not know the actual names of the files which are its own standard input and output, since it need never reopen them.

Filters are straightforward extensions of standard I/O redirection with pipes used instead of files.

In ordinary circumstances, the main loop of the Shell never terminates. (The main loop includes that branch of the return from *fork* belonging to the parent process; that is, the branch which does a *wait*, then reads another command line.) The one thing which causes the Shell to terminate is discovering an end-of-file condition on its input file. Thus, when the Shell is executed as a command with a given input file, as in

sh <comfile

the commands in *comfile* will be executed until the end of *comfile* is reached; then the instance of the Shell invoked by *sh* will terminate. Since this Shell process is the child of another instance of the Shell, the *wait* executed in the latter will return, and another command may be processed.

6.6 Initialization

The instances of the Shell to which users type commands are themselves children of another process. The last step in the initialization of UNIX is the creation of a single process and the invocation (via *execute*) of a program called *init*. The role of *init* is to create one process for each typewriter channel which may be dialed up by a user. The various subinstances of *init* open the appropriate typewriters for input and output. Since when *init* was invoked there were no files open, in each process the typewriter keyboard will receive file descriptor 0 and the printer file descriptor 1. Each process types out a message requesting that the user log in and waits, reading the typewriter, for a reply. At the outset, no one is logged in, so each process simply hangs. Finally someone types his name or other identification. The appropriate instance of *init* wakes up, receives the log-in line, and reads a password file. If the user name is found, and if he is able to supply the correct password, *init* changes to the user's default current directory, sets the process's user ID to that of the person logging in, and performs an *execute* of the Shell. At this point the Shell is ready to receive commands and the logging-in protocol is complete.

Meanwhile, the mainstream path of *init* (the parent of all the subinstances of itself which will later become Shells) does a *wait*. If one of the child processes terminates, either because a Shell found an end of file or because a user typed an incorrect name or password, this path of *init* simply recreates the defunct process, which in turn reopens the appropriate input and output files and types another login message. Thus a user may log out simply by typing the end-of-file sequence in place of a command to the Shell.

6.7 Other programs as Shell

The Shell as described above is designed to allow users full access to the facilities of the system, since it will invoke the execution of any program with appropriate protection mode. Sometimes, however, a different interface to the system is desirable, and this feature is easily arranged.

Recall that after a user has successfully logged in by supplying his name and password, *init* ordinarily invokes the Shell to interpret command lines. The user's entry in the password file may contain the name of a program to be invoked after login instead of the Shell. This program is free to interpret the user's messages in any way it wishes.

For example, the password file entries for users of a secretarial editing system specify that the editor *ed* is to be used instead of the Shell. Thus when editing system users log in, they are inside the editor and can begin work immediately; also, they can be prevented from invoking UNIX programs not intended for their use. In practice, it has proved desirable to allow a temporary escape from the editor to execute the formatting program and other utilities.

Several of the games (e.g., chess, blackjack, 3D tic-tac-toe) available on UNIX illustrate a much more severely restricted environment. For each of these an entry exists in the password file specifying that the appropriate game-playing program is to be invoked instead of the Shell. People who log in as a player of one of the games find themselves limited to the game and unable to investigate the presumably more interesting offerings of UNIX as a whole.

7. Traps

The PDP-11 hardware detects a number of program faults, such as references to non-existent memory, unimplemented instructions, and odd addresses used where an even address is required. Such faults cause the processor to trap to a system routine. When an illegal action is caught, unless other arrangements have been made, the system terminates the process and writes the user's image on file *core* in the current directory. A debugger can be used to determine the state of the program at the time of the fault.

Programs which are looping, which produce unwanted output, or about which the user has second thoughts may be halted by the use of the *interrupt* signal, which is generated by typing the ‘‘delete’’ character. Unless special action has been taken, this signal simply causes the program to cease execution without producing a core image file.

There is also a *quit* signal which is used to force a core image to be produced. Thus programs which loop unexpectedly may be halted and the core image examined without prearrangement.

The hardware-generated faults and the interrupt and quit signals can, by request, be either ignored or caught by the process. For example, the Shell ignores quits to prevent a quit from logging the user out. The editor catches interrupts and returns to its command level. This is useful for stopping long printouts without losing work in progress (the editor manipulates a copy of the file it is editing). In systems without floating point hardware, unimplemented instructions are caught and floating point instructions are interpreted.

8. Perspective

Perhaps paradoxically, the success of UNIX is largely due to the fact that it was not designed to meet any predefined objectives. The first version was written when one of us (Thompson), dissatisfied with the available computer facilities, discovered a little-used PDP-7 and set out to create a more hospitable environment. This essentially personal effort was sufficiently successful to gain the interest of the remaining author and others, and later to justify the acquisition of the PDP-11/20, specifically to support a text editing and formatting system. When in turn the 11/20 was outgrown, UNIX had proved useful enough to persuade management to invest in the PDP-11/45. Our goals throughout the effort, when articulated at all, have always concerned themselves with building a comfortable relationship with the machine and with exploring ideas and inventions in operating systems. We have not been faced with the need to satisfy someone else's requirements, and for this freedom we are grateful.

Three considerations which influenced the design of UNIX are visible in retrospect.

First: since we are programmers, we naturally designed the system to make it easy to write, test, and run programs. The most important expression of our desire for programming convenience was that the system was arranged for interactive use, even though the original version only supported one user. We believe that a properly-designed interactive system is much more productive and satisfying to use than a ‘‘batch’’ system. Moreover such a system is rather easily adaptable to non-interactive use, while the converse is not true.

Second: there have always been fairly severe size constraints on the system and its software. Given the partially antagonistic desires for reasonable efficiency and expressive power, the size constraint has encouraged not only economy but a certain elegance of design. This may be a thinly disguised version of the ‘‘salvation through suffering’’ philosophy, but in our case it worked.

Third: nearly from the start, the system was able to, and did, maintain itself. This fact is more important than it might seem. If designers of a system are forced to use that system they quickly become aware of its functional and superficial deficiencies and are strongly motivated to correct them before it is too late. Since all source programs were always available and easily modified on-line, we were willing to revise and rewrite the system and its software when new ideas were invented, discovered, or suggested by others.

The aspects of UNIX discussed in this paper exhibit clearly at least the first two of these design considerations. The interface to the file system, for example, is extremely convenient from a programming standpoint. The lowest possible interface level is designed to eliminate distinctions between the various devices and files and between direct and sequential access. No large ‘‘access method’’ routines are required to insulate the programmer from the system calls; in fact all user programs either call the system directly or use a small library program, only tens of instructions long, which buffers a number of characters and reads or writes them all at once.

Another important aspect of programming convenience is that there are no ‘‘control blocks’’ with a complicated structure partially maintained by and depended on by the file system or other system calls. Generally speaking, the contents of a program’s address space are the property of the program, and we have tried to avoid placing restrictions on the data structures within that address space.

Given the requirement that all programs should be usable with any file or device as input or output, it is also desirable from a space-efficiency standpoint to push device-dependent considerations into the operating system itself. The only alternatives seem to be to load routines for dealing with each device with all programs, which is expensive in space, or to depend on some means of dynamically linking to the routine appropriate to each device when it is actually needed, which is expensive either in overhead or in hardware.

Likewise, the process control scheme and command interface have proved both convenient and efficient. Since the Shell operates as an ordinary, swappable user program, it consumes no wired-down space in the system proper, and it may be made as powerful as desired at little cost. In particular, given the framework in which the Shell executes as a process which spawns other processes to perform commands, the notions of I/O redirection, background processes, command files, and user-selectable system interfaces all become essentially trivial to implement.

8.1 Influences

The success of UNIX lies not so much in new inventions but rather in the full exploitation of a carefully selected set of fertile ideas, and especially in showing that they can be keys to the implementation of a small yet powerful operating system.

The *fork* operation, essentially as we implemented it, was present in the Berkeley time sharing system[8]. On a number of points we were influenced by Multics, which suggested the particular form of the I/O system calls[9] and both the name of the Shell and its general functions. The notion that the Shell should create a process for each command was also suggested to us by the early design of Multics, although in that system it was later dropped for efficiency reasons. A similar scheme is used by TENEX[10].

9. Statistics

The following numbers are presented to suggest the scale of our operation. Those of our users not involved in document preparation tend to use the system for program development, especially language work. There are few important ‘‘applications’’ programs.

Overall, we have

100	user population
14	maximum simultaneous users
380	directories
4800	files
66300	512-byte secondary storage blocks used

There is a ‘‘background’’ process that runs at the lowest possible priority; it is used to soak up any idle CPU time. It has been used to produce a million-digit approximation to the constant $e-2$, and is now solving all rook-and-pawn vs. rook chess endgames. Not counting this background work, we average daily

2400	commands
5.5	CPU hours
100	connect hours
32	different users
100	logins

Acknowledgements. We are grateful to R.H. Canaday, L.L. Cherry, and L.E. McMahon for their contributions to UNIX. We are particularly appreciative of the inventiveness, thoughtful criticism, and constant support of R. Morris, M.D. McIlroy, and J.F. Ossanna.

-

References

1. Digital Equipment Corporation. *PDP-11/40 Processor Handbook* (1972), *PDP-11/45 Processor Handbook* (1971), and *PDP-11/70 Processor Handbook* (1975).
2. Deutsch, L.P., and Lampson, B.W. An online editor. *Comm. ACM 10,* 12 (Dec. 1967), 793-799, 803.
3. Richards, M. BCPL: A tool for compiler writing and system programming. Proc. AFIPS 1969 SJCC, Vol. 34, AFIPS Press, Montvale, N.J., pp. 557-566.
4. McClure, R.M. TMG—A syntax directed compiler. Proc. ACM 20th Nat. Conf., ACM, 1965, New York, pp. 262-274.
5. Hall, A.D. The M6 macroprocessor. Computing Science Tech. Rep. #2, Bell Telephone Laboratories, 1969.
6. Ritchie, D.M. C reference manual. Unpublished memorandum, Bell Telephone Laboratories (1973).
7. Aleph-null. Computer Recreations. *Software Practice and Experience 1,* 2 (Apr.-June 1971), 201-204.
8. Deutch, L.P. and Lampson, B.W. SDS 930 time-sharing system preliminary reference manual. Doc. 30.10.10, Project GENIE, Univ. Cal. at Berkeley (Apr. 1965).
9. Feiertag, R.J., and Organick, E.I. The Multics input-output system. Proc. Third Symposium on Operating Systems Principles. Oct. 18-20, 1971, ACM, New York, pp. 35-41.
10. Bobrow, D.G., Burchfiel, J.D., Murphy, D.L., and Tomlinson, R.S. TENEX, a paged time sharing system for the PDP-10. *Comm. ACM 15.,* 3 (March 1972) 135-143.