The UNIX Time-Sharing System

D. M. Ritchie

1. Introduction

UNIX is a general-purpose, multi-user time sharing system implemented on several Digital Equipment Corporation PDP series machines.

UNIX was written by K. L. Thompson, who also wrote many of the command programs. The author of this memorandum contributed several of the major commands, including the assembler and the debugger. The file system was originally designed by Thompson, the author, and R. H. Canaday.

There are two versions of UNIX. The first, which has been in existence about a year, runs on the PDP-7 and -9 computers; a more modern version, a few months old, uses the PDP-11. This document describes UNIX-11, since it is more modern and many of the differences between it and UNIX-7 result from redesign of features found to be deficient or lacking in the earlier system.

Although the PDP-7 and PDP-11 are both small computers, the design of UNIX is amenable to expansion for use on more powerful machines. Indeed, UNIX contains a number of features very seldom offered even by larger systems, including

1. A versatile, convenient file system with complete integration between disk files and I/O devices;
2. The ability to initiate asynchronously running processes.

It must be said, however, that the most important features of UNIX are its simplicity, elegance, and ease of use.

Besides the system proper, the major programs available under UNIX are an assembler, a text editor based on QED, a symbolic debugger for examining and patching faulty programs, and B, a higher level language resembling BCPL. UNIX-7 also has a version of the compiler writing language TMGL contributed by M. D. Mcllroy, and besides its own assembler, there is a PDP-11 assembler which was used to write UNIX-11. On the PDP-11 there is a version of BASIC [reference] adapted from the one supplied by DEC [reference]. All but the last of these programs were written locally, and except for the very first versions of the editor and assembler, using UNIX itself.

2. Hardware

The PDP-11 on which UNIX is implemented is a 16-bit 12K computer, and UNIX occupies 8K words. More than half of this space, however, is utilized for a variable number of disk buffers; with some loss of speed the number of buffers could be cut significantly.

The PDP-11 has a 256K word disk, almost all of which is used for file system storage. It is equipped with DECTAPE, a variety of magnetic tape facility in which individual records may be addressed and rewritten at will. Also available are a high-speed papertape reader and punch. Besides the standard Teletype, there are several variable-speed communications interfaces.

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 there, for example symbolic or binary (object) programs. No particular structuring is expected by the system. Files of text ordinarily 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 generate and expect files with more structure; for example, the assembler generates, and the debugger expects, a name list file in a particular format; however, the structure of files is controlled solely by the programs which use them, not by the system.

3.2. Directories

Directories (sometimes, "catalogs"), 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 is exactly like an ordinary file except that it cannot be written on by user 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, which contains an entry for each user's master directory. Another system directory contains all the programs provided as part of the system; that is, all the commands (elsewhere, "subsystems"). As will be seen, however, it is by no means necessary that a program reside in this directory for it to be used as a command.

Files and directories are named by sequences of eight 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 `/a/b/c` causes the system to search the root for directory `a`; then to search `a` for `b`, and then to find `c` in `b`. `c` may be an ordinary file, a directory, or a special file. As a limiting case, the name `/` refers to the root itself.

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.

When a user logs into UNIX, he is assigned a default current directory, but he may change to any directory readable by him. A path name not starting with `/` causes the system to begin the search in the user's current directory. Thus, the name `a/b` specifies the file named `b` in directory `a`, which is found in the current working directory. The simplest kind of name, for example `a`, refers to a file which itself is found in the working directory.

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 actual path name. The name `..` by convention refers to the parent of the directory in which it appears; that is, the directory in

which it was first 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 special file. Special files are read and written just like ordinary disk files, but the result is activation of the associated device. Entries for all special files reside in the root directory, so they may all be referred to by `/` followed by the appropriate name.

The special files are discussed further in section 6 below.

3.4. Protection

The protection scheme in UNIX is quite simple. Each user of the system is assigned a unique user number. When a file is created, it is marked with the number of its creator. Also given for new files is a set of protection bits. Four of these specify independently permission to read or write for the owner of the file and for all other users. A fifth bit indicates permission to execute the file as a program. If the sixth 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 feature provides for privileged programs which may use files which should neither be read nor changed by other users. If the set-user-identification bit is on for a program, the accounting file may be accessed during the program's execution but not otherwise.

3.5. System 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 or physical record size imposed by the system. The size of a file on the disk is determined by the location of the last piece of information written on it; no predetermination of the size of a file is necessary. In UNIX-11, the unit of information is the 8-bit byte, since the PDP-11 is a byte-oriented machine.

To illustrate the essentials of I/O in UNIX, the basic calls are summarized below in an anonymous higher level language which will indicate the needed parameters without getting into the complexities of machine language programming. (All system calls are also described in Appendix 1 in their actual form.) Each call to the system may potentially result in an error return, which for simplicity is not represented in the calling sequence.

3.5.1. Open

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 or written. If the file is to be updated, that is read and written simultaneously, it may be opened twice, once for reading and once for writing.

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

There are no 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 one may imagine situations in which this fact is unfortunate, in practice difficulties are quite rare.

3.5.2. Create

To create a new file, the following call is used.

```
filep = create(name, mode)
```

Here `filep` and `name` are as before. If the file already existed, it is truncated to zero length. Creation of a file implies opening for writing as well. The `mode` argument indicates the permissions which are to be placed on the file by the protection mechanism. To create a file, the user must have write permission in the directory in which the file is being created.

3.5.3. Write

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 always indicates the next byte to be read or written. If n bytes are read, the pointer advances by `n` bytes.

Once a file is open for writing, the following call may be used.

```
nwritten = write(filep, buffer, count)
```

`buffer` is the address of `count` sequentially stored bytes (words in UNIX-7) which will be written onto the file. `nwritten` is the number of bytes actually written; except in rare cases it is the same as `count`. Occasionally, an error may be indicated; for example if paper tape is being written, an error occurs if the tape runs out.

For disk files which already existed (that is, were opened by `open`, not `create`) the bytes written affect only those implied by the position of the write pointer and the number of bytes written; no other part of the file is changed.

3.5.4. Read

To read, the call is

```
nread = read(filep, buffer, count)
```

Up to `count` bytes are read from the file into `buffer`. The number actually read is returned as `nread`. Every program must beprepared for the possibility that `nread` is 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. Furthermore, devices like the typewriters work in units of lines. Suppose, for example, that before anything has been typed a program tries to read 128 characters from the console. This forces the program to wait, since nothing has been typed. The user now types a line consisting, say, of 10 characters and hits the "new line" key. At this point the read call would return indicating 11 characters read (including the new line). On the other hand, it is permissible to read fewer characters than were typed without losing information; for example bytes may be picked up one at a time.

When the read call returns with `nread` 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.

3.5.5. Seek

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

```
seek(filep, base, offset)
```

The read pointer (respectively write pointer) associated with `filep` is moved to a position `offset` words from the beginning, from the current position of the pointer, or from the end of the file, depending on whether `base` is 0, 1, or 2. `offset` may be negative to move the pointer backwards. For some devices (e.g. paper tape and typewriters) seek calls are meaningless and are ignored.

3.5.6. Tell

The current position of the pointer may be discovered as follows:

```
offset = tell(filep, base)
```

As with seek, `filep` is the file descriptor for an open file, and `base` specifies whether the desired offset is to be measured from the beginning of the file, from the current position of the pointer, or from the end. In the second case, of course, the result is always zero.

4. Implementation of the File system

As mentioned in section 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 identification number) of the file. When the file is accessed, its i-number is looked up in a system table stored in a known part of the disk. 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 addresses for the file contents;
4. its size;
5. times of creation and last modification;
6. the number of links to the file; that is, the number of times it appears in a directory;
7. bits indicating whether the file is a directory and whether it is special (in which case the size and disk addresses are meaningless);
8. a bit indicating whether the file is "large" or "small".

There is space in each i-node for eight disk addresses. A file which fits into eight or fewer 64-word (128-byte) blocks is considered small; in this case the addresses of the blocks themselves are stored. For large files, each of the eight disk addresses may point to an indirect block of 64 words containing the addresses of the blocks constituting the file itself. Thus files may be as large as 8*64*128, or 65,536 bytes.

When the number of links to a file drops to zero, its contents are freed and its i-node is marked unused.

To the user, both reading and writing of files appears to be synchronous and unbuffered. That is, immediately after return from a `read` call the data is available, and conversely after a `write` the user's workspace may be reused. In fact the system maintains, unseen by the user, a rather complicated buffering mechanism. Suppose a `write` call is made specifying transmission of a single byte. UNIX will search its own buffers to see whether the affected disk block currently resides in its own buffers; if not, it will be read in from the disk. Then the affected byte is replaced in the buffer and an entry is made in a list of blocks to be written on the disk. 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 disk 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. Because sequential reading of a file is so common, UNIX attempts to optimize this situation by prereading the disk block following the one in which the requested byte is found. This strategy tends to minimize and in some cases eliminate disk latency delays.

A program which reads or writes files in units of 128 bytes has an advantage over a program which reads or writes a single byte at a time, but the gain is not immense. As an example, the editor ed (8.9 and A2.4 below) was originally written, for simplicity, to do I/O one character at a time; it increased its speed by a factor of about two when it was rewritten to use 128-byte units. Because the system attempts to retain copies of the most recently used disk blocks in core, the speed gain in dealing with large units comes principally from elimination of system overhead, not from latency delays.

5. The Shell

5.1. General

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` $\texttt{arg}_1$`,` $\texttt{arg}_2$ `...` $\texttt{arg}_n$

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 the 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 provided by the system itself.

5.2. Standard I/O

The discussion of I/O given 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. In fact, this is not quite true. There are two files always accessible to every program without an explicit `open` or `create`; they have file descriptors 0 and 1. As 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 >files
```

creates a file called `files` and places the listing there. Thus the argument `> files` means, 'place output on `files`'. 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 is needed within each command; the command need merely use the standard file descriptors 0 and 1 where appropriate.

5.3. Command Separators

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 returning with its signal `@`; 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`; however, no matter how long the assembly takes, the Shell returns immediately. 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 all 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 (Incidentally, the spaces before and after the `&` in the examples above are not necessary.)

5.4. The Shell as a Command

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 named `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 interpret parameters to filed command sequences; see section 8.18.)

When the user types the `&` character as part of a command line, he is explicitly invoking the multitasking facilities of UNIX. That is, he is creating a process which runs asynchronously from his normal command stream. Although this ability is quite convenient for the user directly, it is even more useful to UNIX itself.

5.5. Processes and forking

A *process* in UNIX is the execution of a program. The evidence of the existence of a process is a *core image*. While the processor is executing on behalf of a process, the core image, quite naturally, resides in the core memory of the computer; during the execution of other processes, a core image is kept on the disk. In order to provide fast response to user's requests, UNIX, like most time-sharing systems, swaps the core images of processes between core and the disk.

Except while UNIX is bootstrapping itself into operation, a new process can come into existence in only one way: 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 core images which are copies of each other, but they are not precisely equivalent: one of them is considered the parent process. In the parent, control does not return directly from the `fork`, but instead passes to location `label`; in the child process, there is a normal return. The `processid` returned by the `fork` call is the identification of the other, offspring process.

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

5.6. Execution of programs

Another system primitive on which the Shell depends heavily is invoked by

```
status = execute(file, arg1, arg2, ..., argn)
```

which requests the system to read in and execute the program named by `file`, passing it arguments `arg1`, `arg2`, ..., `argn`. Ordinarily, `arg1` should be the same string as `file`. If this call is successful, control never returns to the program which uses it. That is, the image of the named file replaces the current program. 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.

The third and last process control system call used by the Shell is

```
processid, status = wait()
```

This primitive causes its caller to suspend execution until one of its children has completed execution. Then `wait` returns the `processid` of the terminated process and a `status` value indicating how the process died. (Processes which are never waited for die unnoticed and presumably unmourned.)

5.7. Operation of the Shell

The outline of the operation of the Shell can now be understood. Most of the the time, the Shell is waiting for the user to type a command. When the new line character 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, then 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, `waits` for the child process to die. When this happens, the Shell knows the command is finished, so it types out `@` and reads the typewriter to obtain another command.

Given this framework, the implementation of background processes is trivial; whenever a command line terminates with `&`, 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 signal `@`, 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`, closes file 0 or 1 respectively and opens 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 O 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.

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.

The instances of the Shell to which each UNIX user types 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 code for `init` is kept in a file, like every other command. Its role 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 O 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 is found, and if he is able to supply the correct password, `init` changes to the user's default current directory, sets the user number 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 reopens the appropriate input and output files and types another log-in message. Thus a user may log out simply by typing the end-of-file sequence in place of a command to the Shell.

6. Census of Special Files

Here is a list of the special files currently implemented. Since an entry for each resides in the root directory, the file `xyz` may be referred to by `/xyz`. Alternatively, one may link to any of these files under any name desired.

6.1. ppt

When read, `ppt` refers to the paper tape reader; when written, to the punch. Null characters are ignored for both reading and writing, so `ppt` is suitable only for ASCII (not binary) information; on the other hand, the program need not take account of the leader or trailer. End of file occurs during a read when the end of the tape passes through the sensors.

6.2. bppt

`bppt` also refers to paper tape. The tape is in a blocked format with checksums. Completely arbitrary information may be written and recovered unchanged in this mode.

6.3. rppt

This is raw input and output for paper tape. Every character is passed to the program, including nulls, so that the program must know when the leader ends and information begins during a read. On the other hand, this mode is suitable when tapes of unusual format must be read.

6.4. tty

This is the console typewriter. Null characters are ignored for both reading and writing. For reading, the line is a unit of information; a program reading `tty` will wait until a whole line has been typed, and at most one line will be passed back to the program. However, characters may be read one at a time from the line.

On input, erase and kill processing are performed: `#` will erase the last character typed; `@` kills the entire line.

The ASCII character `EOT` signals an end of file to the program. The ASCII "new line" character is the standard means of ending an input line. On the Teletype models 33 and 35 and some other terminals UNIX must simulate this function by echoing a "return" character when it receives a "line space" (whose code corresponds to the ASCII "new line.")

The name `tty` refers to the user's own typewriter, no matter which physical channel he may be using. There are also special files for each typewriter. They have the names `ctty` (for the central site terminal), and $\texttt{tty}_1$, $\texttt{tty}_2$, ... $\texttt{tty}_n$ (for user's typewriters).

6.5. rtty

This is "raw" typewriter I/O. It is identical to `tty` for output, but on input the program waits only until at least one character has been typed before a return from the read occurs. No erase or kill processing is done.

6.6. tap0; tap1

These files refer to DECTAPE logical units 0 and 1. When they are opened, the program waits until a tape is mounted on the appropriate drive.

6.7. disk

This file refers to the entire disk in a way independent of the file system; it reads or writes the physical block corresponding to the current file pointer.

One use of this file demonstrates convincingly the versatility of the special file concept. There is a program called `check` which scrutinizes the entire file system to determine its consistency and the number of disk blocks used for various purposes. This program is in no sense part of the system; it is an ordinary command invokable by any user. `check` operates by reading the file `disk`. In this way it is able to examine the list of i-nodes (cf. section 4) which define files without depending on "*ad hoc*" system calls to obtain its information.

6.8. system

This special file causes the area of core memory occupied by the system to be treated as a file. Thus the system can be examined and patched during operation by use of the ordinary debugger `db` discussed below.

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, the system writes the user's core image on file "core" in the current working directory. Because of the way the hardware and the system operate, the contents of all the program-accessible registers are stored within this core image file. Thus, the debugger `db` discussed below can be used to determine the state of the program at the time of the fault.

The user may also force the program to stop and a core image file to be written by sending an interrupt signal. Currently this signal is generated by typing the ASCII FS character (control \ on model 37 Teletypes). Thus programs which are looping or about which the user has second thoughts may be halted.

If the user has several processes in execution simultaneously (because he used the `&` facility of the Shell) only one of these processes is stopped, and there is no control over which one; it depends on which is currently in execution or executes next. Clearly this situation leaves much to be desired, for several reasons:

1. when the user has several processes he cannot interrupt with any selectivity;
2. in all cases the rather large core image is produced, when the user may merely wished, for example, to stop a long printout;
3. it is often useful to send an asynchronous signal to a process without stopping it (for example by causing a trap to an agreed-upon location within the process's core image.)

Doubtless, therefore, the interrupt facility will be reworked in the future. Unfortunately, there are not only implementation problems but even conceptual ones-- e.g. how does specify a process which may have started an arbitrary time ago?

8. Some Commands

This section summarizes several of the commands available in UNIX. The list is not exhaustive, but it covers those most frequently used. The assembler `as`, the debugger `db`, the editor `ed` and DECTAPE manipulator `tap` are documented in more detail in Appendix 2.

Where an argument like `name` is given, a file name is meant. In every case, an arbitrary path name may be used to specify any file in the system, subject to the constraints imposed by the protection system.

Arguments enclosed in square brackets are optional.

8.1. as -- assemble

`as` is the assembler for the PDP-11. It is called as follows:

$$\texttt{as name}_1\texttt{, name}_2\texttt{, ... name}_n$$

The concatenation of the files $\texttt{name}_1$ `...` $\texttt{name}_n$ is assembled. The resulting binary output is placed in file `a.out` in the current working directory; a copy of the name list from the assembly is placed on `n.out`. See Appendix 2 for more information.

8.2. b -- compile B program

B [reference] is a new higher-level language with implementations on the PDP-7, PDP-11, and Honeywell 635. To compile several B programs,

```
b −name1 ... −namen
```

Notice that each `name` should be preceded by a − (not part of the file name). Also, `b` supplies the conventional suffix `.b`. For example, to concatenate and compile `abc.b` and `def.b`, type

```
b −abc −def
```

The binary output is left on `a.out` and the name list on `n.out` (just like the assembler). See the B reference manual [reference] for more information.

8.3. cat -- concatenate files

The cat command concatenates several files and copies the result onto the standard output file.

```
cat name1 ... namen
```

Notice that > may be used: `cat a b >/ppt` punches the concatenation of `a` and `b`. `cat x` simply lists `x` on the typewriter.

8.4. chdir -- change directories

To change the current directory, use

```
chdir dirname
```

This command is the only one that does not reside in directory `/bin`; instead it is part of the Shell. The reason is interesting. Recall that each ordinary command is executed as a separate process created by the Shell. If the system's `chdir` primitive were executed in such a process, it would have essentially no effect, since the process would terminate instantly without affecting the current directory of the Shell process and its subsequent offspring. The Shell itself recognizes the `chdir` command and calls the system to change directories without creating a new process.

8.5. chmod -- change mode of file

To change the protection bits for a set of files,

```
chmod mode name1 ... namen
```

The modes of $name_1$, ..., $name_n$ are set to mode. ' Mode is an octal number whose bits in the binary representation have the following meanings:

1 write, non-owner

2 read, non-owner

4 write, owner

10 read, owner

20 execute

40 set user ID on execution

See also section 3.4 on the protection system. Most command programs create files with mode 17; the assembler's `a.out` file has mode 37.

8.6. chown -- change owner of files

To change the owner of a sequence of files,

```
chown owner name1 ... namen
```

`owner` is a user number assigned by the system administrators. Only the owner of a file may donate the file to another user. Notice that `chown` does not change the directory in which the link to the file exists.

8.7. cp -- copy file

To make a copy of a file,

```
cp name1 name2
```

Either $\texttt{name}_1$ or $\texttt{name}_2$ may be special files.

8.8. db -- debug

To examine or patch a (usually binary) file,

```
db [ name [ namelist ] ]
```

The first argument is the file to be examined. The second is a name list file produced as `n.out` when `name` was assembled. The brackets indicate that both arguments are optional. If the first argument is not given, `n.out` is assumed. (Of course, the first argument alone cannot be omitted.)

`db` is discussed in complete detail in Appendix 2.

8.9. ed -- edit

ED is the editor. It is essentially a subset of QED [references]; see Appendix 2 for the differences.

8.10. ln -- link

To create a link,

```
ln name1 [ name2 ]
```

A link to file $\texttt{name}_1$, is created. If $\texttt{name}_2$ is given, the link has name $\texttt{name}_2$, otherwise it has the (last component of) $\texttt{name}_1$. For example, `ln /a/b /c/d` creates a link named `d` in directory `/c` to file `/a/b`. The user must have permission to write in directory `/c`.

8.11. ls -- list directory

To list the names of the files in a directory,

```
ls [ name ]
```

If `name` is not given, the contents of the current directory are listed.

8.12. mkdir -- make directory

To create a directory,

```
mkdir name
```

8.13. mv -- move file

To move or rename a file,

```
mv name1, name2 ...
```

This command does not copy the file. It operates by linking to $name_1$ by the name $name_2$, then unlinking $name_1$. mv is often used to rename a file.

If $name_2$ is a directory, $name_2$ is moved into that directory under the name which is the last component of $name_1$. For example,

```
mv x /dirname
```

moves x to /dirname/x.

8.14. nm -- get namelist

To get a printed listing of the symbol table (name list) from an assembly,

```
nm [ name ]
```

where name is the n.out file from some assembly. If name is not given, n.out is listed.

8.15. pr -- print

The command

```
pr name1 name2 ... namen
```

prints the contents of the named files. The output is separated into pages headed by the file name, the time and date, and the page number.

8.16. rm -- remove file

To unlink one or more files (remove them from directories),

```
rm name1 ...
```

Recall that removing the last link to a file causes it to go away.

8.17. roff -- run off (format)

roff is a program similar to the one under GE-TSS which formats text files under the control of commands embedded in the text. The command

```
roff name1 ... namen
```

will run off the concatenation of $name_1$, ... $name_n$. UNIX roff supports all the features of TSS roff except "merge", tabs and footnotes. See [reference] for details.

8.18. sh -- Shell

To invoke the Shell,

```
sh [ name ]
```

name is interpreted as a file of commands. name need not be given, in which case the Shell will read its standard input file. When called with an argument, the Shell refrains from typing its prompt character @. See section 5 above. The Shell has several features besides those mentioned in section 5.

1. Arguments or parts of arguments to commands enclosed in single ' or double " quotes are taken literally, so that arbitrary character strings can be passed (including spaces, < or > etc.).

2. The character \ serves to quote the next character. In this way a single command may extend over several lines, since a new line preceded by \ is treated like a space.

3. When the Shell is invoked as a command, the character sequences `$0`, `$1`, `...` `$9` are treated as parameters. `$0` is replaced by the name of the file being interpreted; `$1` through `$9` are replaced by the first through ninth argument following the file name. For example, when

```
sh runcom arg1 arg2 arg3
```

is typed, `$0` inside of `runcom` is replaced by `runcom`, `$1` is replaced by arg_1, etc.

8.19. stat -- get file status

To discover interesting information about one or more files,

```
stat name1 ...
```

`stat` gives the i-number, the mode, the owner, the size, and the times of creation and last modification for each of $name_1$, `...`

8.20. tap -- manipulate DECTAPE

`tap` is used to load and dump portions of the hierarchy onto DECTAPE. See Appendix 2 for details.

8.21. tm -- time

To discover various information connected with time, the `tm` command can be used:

```
tm [ command arg1 ... argn ]
```

If called without arguments, `tm` prints out the time of day and the total times accumulated in several categories:

1. Processor time charged to the user.
2. System overhead time.
3. Time spent waiting for the disk.
4. Idle time.
5. Time spent in the interrupt routines.

Without an argument, `tm` gives these values both in absolute form (i.e., totals since creation of the system) and as changes since the last time `tm` was called. When called with one or more arguments,the arguments are assumed to constitute a command to be timed. `tm` executes the given command and prints the times required for the command in each of the above categories.

8.22. un -- undefined symbols

It is sometimes useful to know the names of all the undefined symbols in a given assembly. The command

```
un [ name ]
```

searches the (name list) file `name` and prints all the symbols undefined therein. If `name` is omitted, `n.out` is used.

APPENDIX 1

This appendix summarizes all the system calls. To understand the calls to UNIX, it is fortunately necessary to know only very little of the structure of the PDP-11. The machine contains several general registers, of which only two are used for arguments to the calls, namely RO and R1. There is also a condition register, one of whose bits records a carry occurring during an arithmetic operation. To indicate an error the system sets this carry bit; it is cleared for successful calls. There is a conditional branch instruction to test the state of the bit. All registers not used to communicate explicit arguments are unchanged by calls to the system.

The instruction used to call the system is known to the assembler as `sys`; when the processor executes this instruction it is trapped to a specific location inside UNIX. The address field of `sys` contains a number indicating which system call is desired.

The arguments for a call are placed either in a register or immediately following the `sys` instruction.

A number of the calls, principally those dealing with the file system, take strings as arguments. There is a standard format for such a string: it consists of a sequence of bytes ending in a null character. The `open` call below, contains a complete example of how to write such a string.

A1.1 exit

`exit` is used to terminate a process as follows:

```
sys exit
```

There are no arguments, nor is there ever any return from this call.

A1.2 fork

This is the primitive used to generate new processes.

```
sys fork
(old process return)
(new process return)
```

There are no input arguments. The error bit is set if no space is available to create a new process, and control returns only to the old process. `R0` contains the process identification of the new process. See also section 5.

The parent process returns immediately after the `sys` call; the new process skips one word. (The label argument mentioned in the discussion of `fork` in section 5.5 was a white lie.)

A1.3 read

To read an open file whose file descriptor is `filep`, load `filep` into `R0` and

```
sys read
buffer
nchars
```

`nchars` is the maximum number of characters desired. (The actual number, not its address.) The number of characters actually read returns in `R0`. If `R0` is zero, the end of the file has been reached. The error bit may be set if, for example, the file is a tape file and there was a permanent read error, or if an attempt was made to read into an area not part of the user's core image.

A1.4 write

To write an open file with file descriptor `filep`, load `filep` into `R0` and

```
sys write
buffer
nchars
```

where `buffer` and `nchars` are the same as for read. The number of characters actually written returns in `R0` (ordinarily it is the same as `nchars`) and errors are indicated by the error bit.

A1.5 open

To open an already existing file,

```
sys open
name
mode
name:<pathname\0>
```

`name` is the address of a string of characters constituting a path name. The name is terminated by a null (all zero) character, which is indicated by `\0`; the characters < and > are string quotes. `mode` is `0` or `1` to indicate reading or writing respectively. The file descriptor returns in `R0`. If the file cannot be found or if permission is not granted the error bit is set.

A1.6 creat

To create or recreate a file,

```
sys creat
name
mode
```

`name` is the same as for `open`. `mode` is a number encoding the protection bits as specified under the `chmod` command below (A2.9). `creat`ion of a file implies opening for writing; the file descriptor is returned in `R0`.

A1.7 close

To close a file, move the file descriptor into `R0` and

```
sys close
```

A program may have only a limited number of files open at one time (currently, 10). Closing a file allows another file to be opened in its place. Closing is otherwise unnecessary, for an automatic close on all files is performed when the process terminates.

A1.8 wait

To wait for a child process to terminate,

```
sys wait
```

The identification of the terminated process is returned in `R0`. At the present time no further information is returned; in the future a means of determining the fate of the process will be provided.

If the process executing a `wait` has no living children, an error is returned.

A1.9 link

To create a link in an arbitrary directory,

```
sys link
name1
name2
```

$name_1$ and $name_2$ are pointers to names as in `open`. File $name_1$ is linked to, and the link has name $name_2$. An error is indicated if $name_1$ does not exist or if $name_2$ does exist.

A1.10 unlink

To remove the name of a file from a directory,

```
sys unlink
name
```

`name` is a pointer to a name. The specified entry is removed from its directory; if this was the last entry (link) pointing to the file, the file is destroyed.

A1.11 exec

To cause execution of a file as a program,

```
sys exec
name
argp
...
argp:arg1
arg2
.
.
.
0
```

The first argument is the address of a file name. The second argument is the address of a list of argument pointers terminated by a zero pointer. Each argument pointer is the address of a string to be passed to the command or other program. The first argument pointer arg_1 is, by convention, the name of the file being invoked. When a program is executed by the Shell, it can determine the name by which it was called. Thus one may write a single program with several names which takes various actions according to the name used.

A file invoked by `exec` begins execution at its relative location 0. At the start, its stack pointer (one of the general registers) points to a list of its own arguments as follows:

```
count
arg1
arg2
.
.
.
```

where there are `count` arguments in the list. Each arg_i points to a standard format string. The arg_i are the same as those specified to the `exec` call.

A1.12 chdir

To change the current directory,

```
sys chdir
dirname
```

`dirname` points to the standard format string describing a directory.

A1.13 time

The call

```
sys time
```

returns in the `AC` and `MQ` registers the number of sixtieths of a second since the start of the current year.

A1.14 mkdir

The call

```
sys mkdir
name
```

creates the file whose name is pointed to by `name` and marks it as a directory.

A1.15 chmod

The mode of a file is changed by

```
sys chmod
name
newmode
```

See the `chmod` command (section 8.5) for the interpretation of the mode. Only the owner of a file may change the mode.

A1.16 chown

To change the owner of a file,

```
sys chown
name
newowner
```

Only the owner of a file may change its owner.

A1.17 break

To save time, UNIX does not swap all of the 4K user core area when exchanging core images. The locations swapped are those from the beginning of the core image to the initial program break, and from the top of user core down to the stack pointer. The initial program break is determined by the size of the file containing the program. The system's idea of how much to swap may be altered by using this call:

```
sys break
newbreak
```

`newbreak` becomes the first location not swapped. If it points beyond the stack, or to the verify first word in the core image, the entire core image is swapped.

A1.18 stat

The user may obtain a copy of the i-node for a named file:

```
sys stat
name
buffer
```

`name` is the name of a file, and `buffer` is the address of 34 sequential bytes into which information concerning the file is placed. See section 4 for what information is passed; consult a UNIX programming councilor for its format.

A1.19 seek

To move the read or write pointer associated with the open file with file descriptor `filep`, load `filep` into `R0` and

```
sys seek
base
offset
```

See also section 3.5.5.

A1.20 tell

To discover the position of the read or write pointer associated with the open file with file descriptor `filep`, move `filep` into `R0` and

```
sys tell
base
offset
```

The result returns in `R0`. See also section 3.5.6.

A1.21 (unassigned)

A1.22 intr

To control the handling of "break" signals sent by the user,

```
sys intr
```

If `R0` is zero on entry, interrupts are disabled; if `R0` is non-zero, they are enabled.

APPENDIX 2

This Appendix discusses in more detail the usage of the assembler, the editor, the debugger, and the DEC-TAPE manipulation command.

A2.1 as

As is based on the DEC-provided assembler PAL-11 [references], although it was coded locally. Therefore, only the differences will be recorded.

Character changes are:

```
for use
 @    *
 #    $
 ;    /
```

In `as`, the character `;` is a logical new line; several operations may appear on one line if separated by `;`. Several new expression operators have been provided:

```
> right shift (logical)
< left shift
* multiplication
\/ division
\ remainder
! one's complement (unary)
[] parentheses for grouping
```

There is a conditional assembly operation code:

```
.if expression
...
.endif
```

If the `expression` evaluates to non-zero, the section of code between the `.if` and the `.endif` is assembled; otherwise it is ignored. `.if`s may be nested.

Temporary labels like those introduced by Knuth [reference] may be employed. A temporary label is defined as follows:

```
n:
```

where `n` is a digit `0 ... 9`. Symbols of the form `nf` refer to the first label `n:` following the use of the symbol; those of the form `nb` refer to the last `n:`. The same `n` may be used many times. Labels of this form are less taxing on both the imagination of the programmer and on the symbol table space of the assembler.

The PAL-11 opcodes `.eot` and `.end` are redundant and are omitted.

The symbols

```
r0 ... r5
sp
pc
ac
mq
div
mul
lsh
ash
nor
csw
```

are predefined with appropriate values.

The new opcode `sys` is used to specify system calls. Names for system calls are predefined. See Appendix 1 for the list of calls.

Strings of characters may be assembled in a way more convenient than PAL-11's `.ascii` operation (which is, therefore, omitted). Strings are included between the string quote < and >:

```
<here is a string>
```

Escape sequences exist to enter non graphic and other difficult characters. These sequences are also effective in single and double character constants introduced by single and double quotes respectively. Respectively:

use	*for*
`\n`	`newline (012)`
`\0`	`NULL (000)`
`\>`	`>`
`\t`	`TAB (011)`

When errors occur, a single-character diagnostic is typed out together with the line number and the file name in which it occurred. Errors in pass 1 cause cancellation of pass 2. The possible errors are:

```
) parentheses error
] parentheses error
* Indirection (*) used illegally
. . (the location counter) has become undefined
A error in Address
B Branch instruction has too remote an address
E error in Expression
F error in local (F or b) type symbol
G Garbage (unknown) character
M Multiply defined symbol as label
O Odd-- word quantity assembled at odd address
P Phase error-- "." different in pass 2 from pass 1 value
R Relocation error
U Undefined symbol
X syntaX error
```

The binary output of the assembler is placed on the file `a.out` in the current directory. The assembler also generates a file `n.out` which is a copy of the name list from the assembly, that is, a table of the names of symbols used and their values. `n.out` is used by the `db`, `nm` and `un` commands.

The assembler does not produce a listing of the source program. This is not a serious drawback; the

debugger db discussed below is sufficiently powerful to render a printed octal translation of the source unnecessary.

A2.3 db

Unlike many debugging packages (including DEC's ODT, on which db is loosely based) db is not loaded as part of the core image which it is used to examine; instead it examines files. Typically, the file will be either a core image produced after a fault (see section 7) or the binary output of the assembler. db is called as follows:

```
db [ name [ namelist ] ]
```

name is the file being debugged; if omitted core is assumed. namelist is the n.out file produced when name was assembled; if omitted, n.out is assumed. If no appropriate name list file can be found, db can still be used but some of its symbolic facilities become unavailable.

The format for most db requests is an address followed by a one character command.

Addresses are expressions built up as follows:

1. A name has the value assigned to it when the input file was assembled. It may be relocatable or not depending on the use of the name during the assembly.
2. An octal number is an absolute quantity with the appropriate value.
3. An octal number immediately followed by r is a relocatable quantity with the appropriate value.
4. The symbol . indicates the current pointer of db. The current pointer is set by many db requests.
5. Expressions separated by + or " " (blank) are expressions with value equal to the sum of the components. At most one of the components may be relocatable.
6. Expressions separated by − form an expression with value equal to the difference to the components. If the right component is relocatable, the left component must be relocatable.
7. Expressions are evaluated left to right.

If no address is given for a command, the current address (also specified by .) is assumed. In general, . points to the last word or byte printed by db.

There are db commands for examining locations interpreted as octal numbers, machine instructions, ASCII characters, and addresses. For numbers and characters, either bytes or words may be examined. The following commands are used to examine the specified file.

/ - The addressed word is printed in octal.

\ - The addressed byte is printed in octal.

" - The addressed word is printed as two ASCII characters.

´ - The addressed byte is printed as an ASCII character.

? - The addressed word is interpreted as a machine instruction and a symbolic form of the instruction, including symbolic addresses, is printed. Usually, the result will appear exactly as it was written in the source program.

& - The addressed word is interpreted as a symbolic address and is printed as the name of symbol whose value is closest to the addressed word, possibly followed by a signed offset.

<nl> - (i. e., the character "new line") This command advances the current location counter . and prints the resulting location in the mode last specified by one of the above requests.

^ - This character decrements . and prints the resulting location in the mode last selected one of the above requests. It is a converse to <nl>.

It is illegal for the word-oriented commands to have odd addresses. The incrementing and decrementing of "." done by the <nl> and ^ requests is by one or two depending on whether the last command was word or byte oriented.

The address portion of any of the above commands may be followed by a comma and then by an expression. In this case that number of sequential words or bytes specified by the expression is printed. . is advanced so that it points at the last thing printed.

There are two commands to interpret the value of expressions.

= - When preceded by an expression, the value of the expression typewriter (EOT character).

A2.4 ed

ed is nearly a subset of QED [reference]. When called by

```
ed name
```

ed performs an automatic r (read) command on file name. The major differences between ed and QED are:

1. There is no \f character; input mode is left by typing . alone on a line.

2. There are no buffers and hence no \b stream directive.

3. The commands are limited to:

```
a c d i p q r s w = !
```

4. The only special characters in regular expressions are:

```
* ^ $ [ .
```

which have the usual meanings. However, "^" and "$" are only effective if they are the first or last character respectively of the regular expression. Otherwise suppression of special meaning is done by preceding the character by "\", which is not otherwise special.

5. In the substitute command, only the leftmost occurrence of the matched regular expression is substituted.

A2.5 tap

The *tap* command is used as follows:

`tap [01] [crxdt] [s] [v]` $\texttt{name}_1$ `...` $\texttt{name}_n$ $\texttt{-name'}_1$ `...` $\texttt{-name'}_n$

The first argument consists of characters which indicate what is to be done. Subsequent arguments specify a set of files.

A digit (0 or 1) in the first argument indicates the logical unit number on which the tape is mounted. *C, r, x, d,* and *t* are mutually exclusive:

> `c` indicates the creation of a new tape. Files $\texttt{name}_1$, ... $\texttt{name}_n$ are placed on the tape. If any of these are directories, all files and subdirectories therein are placed on the tape as well. Arguments preceded by − indicate files or directories which are not to be placed on the tape even though implied by one of the other arguments.
>
> `r` indicates that the files specified (exactly as for `c`) will be added to the tape. If there was a file of the same name as one of the specified files already on the tape, it will be replaced.
>
> `x` indicates that the specified files are to be extracted from the tape and copied onto the disk. If any directory needed does not exist, it will be created.
>
> `d` indicates that the specified files are to be deleted from the tape.
>
> `T` causes a partial table of contents of the tape to be produced, including all files implied by the following arguments. (E.g., `tap t /dmr` gives the names of all files on the tape in directory `/dmr`.)

Argument `v` (for "verify") may be used in addition to the preceding arguments. Before each file is dealt with as indicated by one of the preceding arguments, the `v` option causes `tap` to pause, type the name of the affected file, and request the user to decide whether the file should be treated. The reply `y` means "yes"; an empty line means "no"; a `q` means "no, and exit from the `tap` command. For example, by the use of `xv`, files can be selectively restored.

Argument `s` may be used alone or in addition to one of `c, r, x, t, d`. It causes `tap` to examine the tape, verify that it can be read properly, and produce statistics on the contents of the tape.