Unix for TOPS-20 Users

Nelson H.F. Beebe
Center for Scientific Computing
Department of Mathematics
University of Utah
Salt Lake City, Utah 84112
USA
Tel: (801) 581-5254

[24-Jun-87]

Contents

1 Introduction

This document has been written to provide a starting point for TOPS-20 users who will be working on UNIX systems, particularly the SUN workstations. It is *not* a small book about UNIX; the last section gives pointers to several books that should prove useful. Rather, it is intended to provide the reader with connections between familiar TOPS-20 concepts and their analogues on UNIX.

UNIX has often be criticized for terseness, lack of mnemonic commands, and command inconsistency, plus for over-emphasis of interactive computer use as opposed to batch "number crunching". To a considerable extent, these criticisms are grounded in fact. However, anyone who uses UNIX for an extended period and learns how to make good use of the multitude of tools available almost without exception becomes a convert. Despite its warts, UNIX is in many ways a very beautiful operating system which has had, and continues to have, enormous influence on operating systems design, programming languages, software development environments, and even on hardware of several new architectures. Its clean, sleek, design can be attributed largely to the small number of very talented researchers at AT&T Bell Laboratories who developed it for the first several years of its first existence on a PDP-11, with subsequent ports to IBM 370 and Honeywell GCOS systems. Several of them deserve mention here, for you will see their names mentioned frequently in UNIX books, literature, and documentation:

- Ken Thompson for the operating system kernel and file system;
- Dennis Ritchie for the C language;
- Steve Johnson for the Portable C compiler, and the `yacc` and `lex` compiler generator tools;
- Brian Kernighan, Peter Weinberger, and Al Aho for numerous software tools and books about UNIX and C;
- Stu Feldman for the first Fortran 77 compiler anywhere, `f77`, and for `make`, possibly the all-time greatest software tool ever written.

Implementations of UNIX run on machines as small as the IBM PC, and as large as the Cray 2. With only a few exceptions (CDC, Fujitsu, and IBM), nearly every new supercomputer introduced to the market since 1985 has, or will soon have, UNIX. No other operating system in the world runs on so many different hardware architectures, providing, for the first time in computing's forty-year history, the possibility of true programmer and software portability across multiple architectures. From management's point of view, this should make it much easier to chose computers more on the basis of hardware performance, and less on maintaining the status quo that has left mainframe computing environments only superficially different than they were in 1963 when the first multi-model architecture, the IBM 360 series, was introduced.

2 What's in the Name

Ken Thompson chose the name UNIX as a pun on the operating system Multics, from which it borrowed many ideas. The initial implementation on the PDP-11 was in assembly code, but this was soon rewritten in the C language. Dennis Ritchie developed C from B, which in turn came from BCPL, one of the first portable systems programming languages. BCPL and B still exist, and B has produced a descendant, Margay, which is used to write the Waterloo Maple algebra system. C has completely overshadowed them however.

AT&T Bell Laboratories has trademarked the name UNIX, so other vendors who license it from them must usually come up with new names. AT&T has now gone through public releases of Version 6, Version 7, System III, System V, PWB (Programmer's Workbench), and DWB (Documenter's Workbench). Version 8 is in development internally.

Berkeley has had 4.0, 4.1, 4.2, and 4.3 of BSD (Berkeley Standard Distribution) UNIX. Their contributions to UNIX are legion; the most significant are the original port to the DEC VAX architecture, the addition of virtual memory paging support, the fast file system, the C shell (`csh`), and TCP/IP networking support.

Hewlett-Packard calls theirs HPUX; it is based on System V with Berkeley extensions.

Sun's implementation is called UNIX in their documentation; they are apparently licensed to do so. It is based on 4.3 BSD with System V compatibility, and Sun is working together with AT&T to produce a common UNIX merging the features of both these major implementations.

IEEE has published a portable operating system interface definition called POSIX, derived from a subset of Version 7, System III, and System V.

Gould's implementation is UTX-32. The IBM RT version is called AIX. On the Cray 2, it is called UNICOS.

On the IBM PC, we have Venix from VentureCom, and Xenix from Microsoft.

Nobody to my knowledge has yet produced a cleaned-up version called Kleenix.

3 Command Processors

UNIX offers a variety of command processors, called shells, which are analogous to combinations of the TOPS-20 EXEC, MIC, and PCL .

The original one, `sh`, is known as the Bourne shell, after its author, Steve Bourne. The most popular one in Berkeley UNIX environments is Bill Joy's `csh`, the C shell, so called because of its resemblance to the C programming language. `csh` offers better programmability, command history, command aliasing, and importantly, job control allowing starting and stopping foreground and background jobs.

A newer shell is the Korn shell, `ksh`, which combines features of both `sh` and `csh`, and offers command functions and `Emacs`-style command editing. The `fpsh` is a Backus functional-programming style shell. Neither `ksh` nor `fpsh` has received wide distribution, but `ksh` may eventually replace `sh` if it receives strong AT&T backing.

In Sun UNIX, shells and other programs may be conveniently run in a windowing environment; see the `man` pages on `dbxtool`, `shelltool`, `suntools`, and `tektool`.

Unlike the TOPS-20 EXEC, UNIX shells have only a small number of built-in commands, and these are exclusively for control of shell options. What corresponds to most EXEC commands are actual executable programs whose names are not known to the UNIX shells; they must therefore be typed without abbreviation. They are searched for in the file system in a chain of directories defined by the `PATH` environment variable, much like the TOPS-20 `SYS:` logical name. For this reason, and also because UNIX was originally developed on slow teletype printing terminals, UNIX commands tend to be short, with many 2- and 3-letter commands.

The TOPS-20 EXEC has knowledge of the syntax for all of its built-in commands, and PCL gives ordinary users the ability to add new ones which are indistinguishable from EXEC commands as far as their user interface is concerned. When you type something like `COMPILE /?`, the EXEC knows what switches are available, and will display them in response to your query, but no attempt has been made to load any particular compiler into memory. A program whose syntax is not built-in to the EXEC, such as `FORTRA.EXE`, cannot give you help until it is running. That is why `FORTRA /?` does not work, but `FORTRA` followed by a carriage return and `/?` does, since by that time, the program is executing. The particular order of switches and filenames on the command line is up to whoever defined the parsing syntax, and consequently, TOPS-20 commands exhibit a variety of styles, including subcommand modes, such as used by the `BUILD` and `DIRECTORY` commands, which have a large number of options.

In UNIX, the shells do not carry built-in knowledge of individual program options and filename expectations. Instead, they assume that *all* commands are invoked with the syntax `commandverb` {`optional switches`} `filelist`. Switches are conventionally prefixed by a hyphen (though there are exceptions, such as `tar` and `ar`). In any event, everything after the `commandverb` is assumed to be a potential filelist, and the text is scanned for wildcard characters (which cause filename expansion), and the command line is expanded into a list of blank-separated arguments. The main routine in the program itself is then presented with two parameters—an integer count of the number of arguments, and a vector of pointers to each of the arguments.

For many purposes, this is quite adequate. No program has to handle wildcard expansion, since it sees only the final list of command-line arguments.

Option parsing is simple because the initial character of each argument is examined to see whether it is a hyphen or not; if it is, an option switch is assumed, and the remaining characters of the argument are further examined to classify the argument. Otherwise, the argument is assumed to be a filename.

There are several disadvantages of this approach.

- No in-line help is available from the shell, like it is with the TOPS-20 EXEC. This is a serious flaw of essentially every other operating system but TOPS-20, and forces a user who has partially typed a command line and then forgotten the name of a switch to reach for a printed manual, or abort the typein and go search for on-line documentation.
- Invalid syntax in the command line is not detected until the program has already been located in the file system, loaded into memory, and started, which wastes system resources. Since typographical errors are common, this happens frequently.
- The filename expansion done by the shell is limited to the size of the pointer vector buffer, which is unfortunately fixed when the shell is built. Although it is large, it is not large enough to hold expansions of several large directories; for example, `ls -l /usr/man/man?/*` overflows the shell buffer and the command does nothing.
- Because the shells handle wildcard expansion, everyone assumed that would always be sufficient, and no UNIX library that I am aware of provides a user-callable function for wildcard expansion. Thus, a program like UNIX Kermit cannot handle a `get *.*` command, unless that request was on the command line.
- Expanding wildcards implies searching the file system, and the way that the shells, and programs like `tar`, have done this is to read the directory files themselves, making user code unnecessarily knowledgeable about details that really should be known only to the kernel. In Sun's latest release of UNIX, directories can only be read via special library calls, instead of by explicit opens and reads of directory files, indicating that the importance of my criticism is only now beginning to be understood.
- The non-trivial pattern matching code required by most editors and the shells has only recently been added to the standard libraries on some UNIX systems (`regexp(3)`).

The TOPS-20 approach for wildcard expansion permits the user to present a wildcarded file specification to the Monitor through a system call, then to retrieve one matching filename with each subsequent system call. If the program wishes to offer fancier pattern matching, it can easily do so by requesting the most general file specification from the Monitor in the first place, then apply the pattern match filter to each returned name. Space never has to be allocated to store all the file names at once, since the Monitor is able to handle the buffering of filenames; there is no difficulty then in asking for a listing of all the files in the file system, `DIRECTORY PS:<*>*.*.*`. In UNIX, this can only be done by a subterfuge of additional options of some commands, like `ls`. If the user really wants a complete list of all the files in the wildcard expansion, then there is no difficulty in dynamically allocating the space for them as they are retrieved one by one.

In some recent implementations of 4.3BSD, users have modified shells to provide some limited in-line help with filename expansion. On the Sun, if the `csh` environment variable `filec` is set, then if Ctl-D is typed on the command line (*not* at the beginning, because that would be an end-of-file signal), a list of all files which match to that point is displayed. This action is suppressed for the first word on each line (the command verb), so you cannot use it to find, say, all commands that begin with a certain letter. It is unfortunate that UNIX chose the query as a pattern matching character; it should have been reserved for future use as a help character. Ctl-D is much less obvious.

4 Command Correspondence

The following tables give a brief summary of TOPS-20 commands (obtained by typing "?" to the EXEC) and popular programs with UNIX equivalents.

In some cases, there is a close match between them, and in others, only very rough equivalence. For example, although UNIX has `mount` and `umount` commands, these correspond to privileged TOPS-20 `OPR` commands `SET DISK AVAILABLE` and `DISMOUNT`, rather than to the EXEC `ASSIGN`, `DEASSIGN`, `MOUNT`, and `DISMOUNT` commands. There is no way for a UNIX user to request that the operating system `ASSIGN` or `MOUNT` a device, such as a tape drive; the operating system provides no such control. Instead, the user just references the device name and hopes no one else had the same idea at the same time (e.g. `tar -cv *` writes on `/dev/rmt0` without any volume verification).

Private TOPS-20 Commands	Unix Equivalents
ALERT	leave
ANOTHERDIRECTORY	ls
BELL	echo -n Ctl-G
CD	cd, pwd
DSKUSE	du, df
FILE-ACCESS-COUNT	-n/a-
FINGER	finger
KEYLOAD	cat
LPQ	lpr
MKDIR	mkdir
PCL	-n/a-
PQ	lpr
PSYS	finger, last, ps, rwho, who
PURGE	-n/a-
QSPELL	spell
REPEAT	for
STCOPY	cpio, cp, tar
STRSF3	strsf3
TDELETE	find | rm
TEST	test
TYPE	cat
UPDATE-ALERT	leave
WAIT	sleep, wait
WHAT	env, printenv
XDIRECTORY	csh, ksh, sh

Standard TOPS-20 Commands	Unix Equivalents
?	apropos, man, whatis, whereis, which
ACCESS	-n/a-
ADVISE	-n/a-
AGAIN	!!, !#
APPEND	cat >>
ARCHIVE	ar, tar
ASSIGN	-n/a-
ATTACH	-n/a-
BACKSPACE	dd, mt
BLANK	clear
BREAK	*Ctl-D after* write
BUILD	mkdir, rmdir
CANCEL	atrm, lprm
CLOSE	-n/a-
COMPILE	cc, f77, make, pc, pi, pix, px
CONNECT	cd, pwd
CONTINUE	csh: fg
COPY	cp
CREATE	ed, emacs, ex, vi
CREF	cxref, ctags, pxref
CSAVE	strip, csh: *Ctl*-\; see also COMPILE
DAYTIME	date
DDT	adb, dbx, dbxtool, sdb
DEASSIGN	-n/a-
DEBUG	cc -g, f77 -g, pc -g *plus* adb, dbx, dbxtool, sdb
DECLARE	-n/a-
DEFINE	csh: setenv, sh: export NAME=
DELETE	rm
DEPOSIT	-n/a-
DETACH	-n/a-
DIRECTORY	ls
DISABLE	*Ctl-D from* su
DISCARD	-n/a-
DISMOUNT	-n/a-
DO	*shell commandfile*
EDIT	ed, emacs, ex, sed, textedit, vi
ENABLE	su
END-ACCESS	-n/a-
EOF	-n/a-
ERUN	-n/a-
EXAMINE	-n/a-

Standard TOPS-20 Commands	Unix Equivalents
EXECUTE	*commandname*
EXPUNGE	-n/a-
FDIRECTORY	ls -l
FORK	csh: bg, fg; switcher
FREEZE	csh: fg *plus Ctl-Z*
GET	-n/a-
HELP	apropos, man, emacs *Ctl-H I*
HISTORY	history
INFORMATION	atq, env, lpq, printenv
KEEP	csh: bg
KMIC	-n/a-
LOAD	cc, f77, ld, make, pc, pi, pix, px
LOGOUT	logout *or Ctl-D*
MERGE	-n/a-
MODIFY	-n/a-
MOUNT	-n/a-
NAME	-n/a-
ORIGINAL	-n/a-
POP	*Ctl-D*
PRESERVE	-n/a-
PRINT	lpr
PUNCH	-n/a-
PUSH	csh, sh
QD	-n/a-
R	-n/a-
RDIRECTORY	ls -ltu
RECEIVE	mesg
REENTER	-n/a-
REFUSE	mesg
REMARK	cat >/dev/null
RENAME	mv
RESET	csh: kill -9 %#
RETRIEVE	-n/a-
REWIND	dd, mt
RUN	*commandname*
SAVE	csh: *Ctl-\*
SET	leave, csh: set verbose, passwd
SKIP	dd, mt
START	csh: fg
SUBMIT	at
SYSTAT	finger, last, ps, rwho, w, who

Standard TOPS-20 Commands	Unix Equivalents
TAKE	csh file, sh file, csh: source file, sh: . file
TALK	mesg, write
TDIRECTORY	ls -lt
TERMINAL	reset, stty, tset
TRANSLATE	-n/a-
TYPE	cat, less, more, view
UNATTACH	-n/a-
UNDECLARE	-n/a-
UNDELETE	-n/a-
UNKEEP	-n/a-
UNLOAD	dd, mt
VDIRECTORY	ls -l
WDIRECTORY	ls -lt

Common TOPS-20 Programs	Unix Equivalents
AMSTEX	amstex
AWK	awk
BBOARD	rn
BOLDx	tgrind, vgrind
CB	cb, indent
Ctl-T	Ctl-T, time
DETAB	expand, unexpand
DIFF	diff, diff3, sdiff
DOCUMENT	format, roff, nroff, troff
DOWNLD	cat, setkeys
ECHO	echo
EGREP	egrep
FAIL	as
FGREP	fgrep
FILCOM	diff, diff3, sdiff
FINGER	finger
FORPRT	fpr
FORPTX	fpr
FTP	ftp, rcp, tftp, uucp, uusend
GETSPD	stty
GREP	grep
HEAD	head
INDENT	cb, indent
INFSCR	du
KERMIT	kermit
LATEX	latex
LIBREF	libref, nm
LPTOPS	devps, lptops, transcript
MACRO	as
MAKE	make
MAKFIL	fsplit, makfil, split
MAKLIB	ar, ranlib
MAPLE	maple
MIDAS	as
MLTCOL	pr
MM	mail
NETSTAT	finger, netstat, ruptime, rwho, traffic

Common TOPS-20 Programs	Unix Equivalents
PASTOC	pastoc
PCHIST	gprof, monitor, prof, tcov
PCLOOK	gprof, monitor, prof, tcov
PFORM	-n/a-
PFORT	pfort
PHOTO	script
PRETTY	pretty
QSPELL	spell
RATFOR	efl, f77, ratfor
REDUCE	reduce
RUNOFF	format, roff, nroff, troff
SCRIBE	scribe
SED	sed
SEDIT	ed, ex, sed
SETSPD	stty
SF3	sf3
SNOBOL	icon, snobol
SORT	sort, tsort
SRCCOM	diff, diff3, sdiff
TAIL	tail
TELNET	telnet, rlogin, rsh, uux
TEX	tex
TMACRO	cpp, m4, tmacro
TNET	tip
TPUTIL	ansitape, dd, mt, tar
TRIM72	awk
TSTAGE	find
UHELP	uhelp
UNITS	units
USRLST	ls /usr
UUDECODE	uudecode
UUENCODE	uuencode
VAXTAP	ansitape
VMSHELP	vmshelp
XREF	cxref, pxref
XSEARCH	egrep, fgrep, grep, ngrep

5 File Tree Organization

Tops-20 and Unix both offer what appears to the user to be a tree-structured file system made up of ordinary data files and directory files; the directory files contain lists of files, including possibly directory files. Tops-20 file names are case insensitive, and conventionally displayed in upper-case. UNIX file names are *case sensitive*: files `foo`, `FOO`, and `Foo` are distinct. Since ASCII upper-case letters collate before lower-case letters, UNIX users traditionally spell all file names in lower-case, except those few special files that they want to appear at the start of a directory listing, such as `Makefile`, `README`, and `TODO`, where they are more likely to be noticed.

Here is a comparison of these file tree structures.

Directory Object	TOPS-20
Directory file	name.DIRECTORY.1
Root directory	device:<ROOT-DIRECTORY>
Top-level directory name	device:<ROOT-DIRECTORY>FOO.DIRECTORY.1
Top-level directory	device:<FOO>
Subdirectory	device:<FOO.BAR>
Ordinary file	device:<FOO.BAR>file.ext.gen

Directory Object	UNIX
Directory file	anyname[1]
Root directory	/device
Top-level directory name	/device/foo
Top-level directory	/device/foo
Subdirectory	/device/foo/bar
Ordinary file	/device/foo/bar/file.ext[2]

There are both good and bad points of these file systems.

Both have the concept of a default login directory and a "current default directory", so frequently, only the file name, not its full directory path, has to be specified. UNIX supports relative directory paths, and TOPS-20 will too with the next version of the Monitor to be installed in late spring 1987. This reduces the amount of typing users have to do, and makes it possible to define command files which never have to name a full directory path explicitly, allowing directory trees to be moved from machine to machine without having to edit their command files. The <PLOT79> UNIX implementation makes extensive use of this capability.

On UNIX, files in the current directory can be named without an explicit path, or prefixed by a relative path `./`, since `.` is a shorthand for the current directory. The parent directory shorthand is `..`, so a file in that directory could be referred to as `../filename`. This notation can be repeated:

[1]Unix has no special directory name format; the directory file attribute is stored with the file protection bits

[2]Unix does not have file generations, although multiple periods are permitted in names, so they can be simulated—`file.ext.1` is a valid file name.

`../../../filename` is a file in the great grandparent directory. You rarely need to use the `./` notation, but it is sometimes handy to disambiguate a filename with an initial hyphen from an option switch, which is always begun by a hyphen in UNIX; for example, `rm -foo` raises an error message *rm: unknown option*, but `rm ./-foo` deletes the file successfully.

Another useful shorthand supported by `csh` and several utilities, but not by `sh`, is tilde to represent the login directory: `/.cshrc` is a file in that directory. For `sh`, you must use a standard environment variable, `HOME`: `$HOME/.cshrc`; this works with `csh` too.

On both systems, directory files are special—they may be readable by a user program (on TOPS-20, only by making system calls, for reasons made evident below), but only the operating system kernel is ever permitted to write them, in order to maintain file system integrity.

The UNIX format is simple and consistent, but one cannot tell from the name if a file represents a directory file or not. This often does not matter, for if you ask for a directory listing of a file which is a directory, you get a list of the files in that directory, but if it is a normal data file, you just get the name back; both cases are considered legitimate, and no error is raised for either.

The UNIX file naming syntax and operating system support for it make devices equivalent to files from the point of view of the programmer. Here are some examples:

`/dev/tty`	user terminal
`/dev/mt`	tape drive
`/dev/lpr`	line printer spooler
`/dev/null`	null device
`/u/ma/jones/foo.bar`	ordinary file

The null device is always empty for input, and never full for output; it is primarily useful for providing dummy input and output files.

The TOPS-20 format is more complex, but clearly distinguishes directory files from ordinary data files. Device names can be used in place of file names, but the directory and file specification are just ignored. Thus, for the above UNIX examples, TOPS-20 has

`TTY:`	user terminal
`MT:`	tape drive
`LPT:`	line printer spooler
`NUL:`	null device
`APS:<U.JONES>FOO.BAR.3`	ordinary file

There is, however, a significant distinction in the implementation of the directory system. A TOPS-20 directory contains, among other things:

- file names;

- file attributes (byte count, byte size, page count, protection, account, login or files-only, owner name, last writer name, reference counts, temporary or permanent, deleted but not expunged, archived, visible/invisible, . . .);
- times of creation, last read, last write, last update, last backup, and expiration;
- for archived files, names of two separate tapes on which the file contents reside off-line;
- logical disk address pointers to the parent directory;
- logical disk address pointers to each file's contents;
- disk quotas;
- login passwords.

Since the Tops-20 operating system caches active directories in memory, finding a file in the current directory normally does not require any disk accesses. The pointers to parents and children embedded in the directory file mean that it cannot be renamed or moved to a new disk structure without being reorganized and rewritten. There is no effective limit to the number of files in the entire file system; files can continue to be created until there is no more disk space left in the file system. If a directory file block becomes corrupted, it is still usually possible to reconstruct and recover virtually all of the whole file system by virtue of the forward and backward pointers of the directory blocks (each file has an index block which points to its owner directory). This is the job of the `CHECKD` utility which is run whenever the DEC-20/60 is rebooted after a hardware or electrical failure which potentially could corrupt the file system.

Under Unix, directory files contain essentially only two things:

- file names;
- for each file, its index into a master table of all files in the system (the Unix *inode* table);

A Unix directory file always contains an entry for its parent directory and itself, and so, like a Tops-20 directory, cannot be moved without internal reorganization. On the other hand, finding the parent directory from a subdirectory can be done efficiently from information in the directory file itself.

Unix stores a file reference count in its master inode table, which allows a file to have multiple names. Deleting a file removes its name from its owning directory and decrements the reference count; only when the count reaches zero is the file space actually freed. One important use of these "linked files" is replication of data files to large groups of users, such as students in a class—they can each have their own copies of a problem test file, but only one copy is ever stored on disk. Rewriting the file breaks the links to the other copies, so one user cannot change another's files this way. It also makes it easy to create aliases for files or for file trees.

The master file table is indexed by the inode number from the directory, and the entry it selects is a structure containing file attributes (ownership, protection, etc.) and physical disk address. The weak point here is this single master file table:

- if it is lost, the whole file system is lost;
- it is a sparse linear table of perhaps 100,000 to 500,000 entries in a large system, so access can be slow;
- it is not extendable; once it fills up, the file system is full, even if lots of disk space is still empty.

These issues have received considerable attention from UNIX developers. Duplicate copies of the master file table and frequent updates to disk, plus increasingly reliable disk media and intelligent external disk controllers, have made the frequent file system losses of the past rare today. Internal memory caching of large parts of the master table improves the lookup performance. Neither system has yet been able to deal with the third problem; when it strikes, the only recourse is a complete rolloff of the file system to tape, then a complete system rebuild with an enlarged file table. On a large system, this could easily be an all-day job for the operations staff. To reduce the likelihood of a system-wide inode table overflow, and to allow for selective file system backup, UNIX disks are normally configured into multiple logical volumes, each with their own inode table. If one of these logical volumes suffers overflow, as long as it is not the root volume, it can be unmounted and rebuilt without halting the system.

Disk quotas in UNIX are maintained on the basis of file ownership from limits set in the user authorization files, not by directories as they are on TOPS-20. The UNIX approach is usually more convenient for a user, since the quota is independent of the number of subdirectories, but it also makes it impossible to regulate disk usage by directory as can be done on TOPS-20.

UNIX file systems tend to make heavy use of directories, which can be confusing for new users. Most, however, adhere to certain common conventions for naming of system directories. See the `man` page on `hier(7)` for an outline of the important ones.

6 User, Directory, and File Names

On TOPS-20, a username is used to form the name of the login directory; user SMITH has a login directory `PS:<SMITH>`. Under UNIX, the username is determined from a system authorization file,[3]and need have no particular connection with the login directory name.

A UNIX login name is limited to 8 characters in length; TOPS-20 usernames can be up to 39 characters long.

Filename length limits vary with the version of UNIX. Originally, filename components (the parts between slashes in the full name) could not exceed 14 characters in length; the limit on the full filename including the path specification was unclear. With 4.3BSD, any filename component may be up to 256 characters long, and the entire filename with complete path specification may not exceed 1024 characters in length.

Filename components may contain any characters except NUL and slash; some versions of UNIX even permit non-printable characters in filenames. In general, use of non-printable and special characters is likely to interfere with shell wildcard processing and parsing syntax, so the recommended practice is to restrict filename characters to letters, digits, hyphen, and underscore. Tilde and sharp are used by `sccs` and some editors to mark backup copies. It is perfectly acceptable to have multiple periods in a name; for example, `yacc` creates a file named `y.tab.h`. However, to ease file portability to other operating systems, you should limit yourself to one period per filename, and avoid mixed case and special characters.

Most UNIX language compilers require fixed filename suffixes, like `.c`, `.f`, `.o`, `.p` and `.s`. Executable programs and shell scripts conventionally have *no* filename suffix, making them indistinguishable from one another; the shells actually read the first few characters of the file to figure out which of the two file types it is.

Filenames beginning with a leading period are treated as *hidden* files. Shell wildcard expansion does not include them unless a pattern beginning with a period is given. The command `ls -a` will show all files, including the hidden ones. Typically, such files are used for providing program default initialization data, such as for login, and for shells and editors.

7 Device Names

One of the great strengths of UNIX is that the operating system provides a uniform view of devices as files, allowing standard I/O functions to be used to read, write, and control both without regard to the sometimes substantial differences between them. Kernel device drivers are provided for each of the devices on the system; device `xxx` appears to the user as a file `/dev/xxx`.

Doing `ls /dev` will list all of the system devices. `/dev/tty` is the standard name for the job's controlling terminal. `/dev/null` is the null device—reading

[3]On Unix, the normal text file `/etc/passwd` is maintained by the system manager. These files list authorized users, their encrypted passwords, group and user numbers, login directory, and default command interpreter.

from it always returns immediate end-of-file, and writing to it discards the output. On a Sun workstation, `/dev/fb` is a graphics frame buffer.

`/dev/mt` is the standard magnetic tape drive. Several points should be noted about it:

- Tapes are usually available as a “raw” device as well, named `/dev/rmt`.
- Multiple drives are identified by trailing digits `/dev/mt0`, `/dev/mt1`, ..., `/dev/mt15`.
- The drive *number* is frequently keyed to a default tape density, as well as to an auto-rewind-after-write feature. Consult `man` section `mtio(4)` for details. This bizarre behavior is occasioned by lack of support in the kernel for tape functions which could permit separate control of tape formats and positioning.

8 Logical Names

UNIX has shell and environment variables which are somewhat analogous to TOPS-20 logical names. They differ in two important aspects. First, their values are arbitrary text strings, instead of being restricted to filename strings. Second, they are handled by the shells, and *not* by the UNIX kernel.

This means that they can be, and commonly are, used for communicating arbitrary strings to programs at runtime, which can obtain their values with the `getenv()` system call.

A TOPS-20 program can open a file named `MUNG:FOO.BAR`, and the Monitor will look up the definition of the logical name `MUNG:`, which might be a chain of directory names, and prefix each one in turn to the filename `FOO.BAR` until it finds a valid file specification. In UNIX, every program which wants this feature must handle it explicitly, and there is not even a standard library function to do the job. The shells automatically search the directory list in the `PATH` variable to find a file to be executed.

With `csh`, you set a local shell variable by `set NAME=value`, and view the list of current variables by `set`. With `sh`, the syntax is `NAME=value`, and the `set` command again displays the current list. Such variables can be used for later string substitution in shell commands; wherever the string `$NAME` is found on the command line, its current value will be substituted.

To make variables available to programs which are run from the shell, they must be put into the environment. With `csh`, this is done by `setenv NAME value`, and with `sh`, by `export NAME=value`, or for the duration of a single command, by `NAME=value; commandname`. With both shells, `printenv` will display the current variable list.

9 Pipes, I/O Redirection, and Background Jobs

Unix introduced three enormously valuable notions which had previously not existed in commercial operating systems. They are the subject of this section.

The first of these is that every job has associated with it three standard sequential I/O streams: `stdin`, `stdout`, and `stderr`. These are intended for normal input, normal output, and abnormal output, and all three are automatically opened and available for use when any job begins execution, *independent* of the programming language it is written in.

The shells handle assignment of files to these three streams; if you do not specify otherwise, they default to the controlling terminal. To redirect them, you just add a phrase anywhere on the command line:

Stream	Redirection
`stdin`	`<infilename`
`stdout`	`>outfilename` (overwrites)
`stdout`	`>>outfilename` (appends)
`stderr`	csh: `&filename` (overwrites)
`stderr`	csh: `&&filename` (appends)
`stderr`	sh: `2>filename` (overwrites)
`stderr`	sh: `2>>filename` (appends)

It is frequently desirable to merge the output of `stderr` and `stdout`. With `csh`, you use `>&`; with `sh`, you use `2>&1`. The peculiar numbers used in the `sh` syntax are an obvious poor design; they reflect the fact that the operating system guarantees that these will be represented in system calls by the integer file descriptors 0 (`stdin`), 1 (`stdout`), and 2 (`stderr`).

The second important concept is piping—`stdout` from one process can be "piped" into `stdin` of another by using the syntax `prog1 | prog2`; the vertical bar is the pipe symbol. Pipes are one-way sequential data streams which for efficiency are *not* materialized in disk files. When the first process fills up the pipe, it is suspended until the second process empties the pipe. Typical pipe buffer sizes are about 4K bytes, although this may depend on the particular implementation. Besides saving on disk storage, pipes make it possible for a single process to communicate to another an amount of data larger than the file system could contain, and they permit simple simultaneous processing. Programs that transform their single input to produce a single output stream are known as *filters*, and most Unix tools can be used that way. As an example, a simple spelling checker could be implemented by a program which broke a document into a stream of words, with its output piped into a dictionary lookup program which in turn echoed exceptions to its output. This could start producing exceptions soon after startup, instead of waiting until a possibly very long document was broken into words. Instruction pipelining is one of the important ways to improve hardware performance, and it helps in software performance too.

Sometimes you want to trap the data flowing through a pipe, perhaps to view error messages on the terminal, or collect them in a file. All that is required is a program that copies its `stdin` to its `stdout`, and simultaneously makes a copy in a file. The `tee` program (named for a T-joint in a plumbing pipe) does this: `prog1 | tee filename | prog2`.

The third important concept is background job processing. When the shell starts another process running through the `fork()` and `exec()` system calls, it normally waits for the process to complete before reading more of its own input. However, if the command line is terminated by an ampersand (&), the shell does not wait, and the process runs in the background, usually at a lower priority, while the shell immediately resumes its input processing. This is an extremely convenient feature, because it allows the user to avoid having the terminal tied up while a long running process, such as a text search or a compilation, is in progress.

You can even start a background job and logout, letting it run to completion, providing you protect it from the hangup signal issued at logout which normally terminates all running processes belonging to that shell. The `nohup` (no hangup) command takes care of this; type `nohup somecommand` and then `logout`.

The `csh` even allows you to suspend and resume processes; the lack of this job control feature in AT&T System III and System V UNIX is one of its more serious flaws in comparison to Berkeley UNIX.

10 Terminal Support in Unix

Terminal control has been a perennial problem in every computer system, because few terminal vendors have been able to agree on what command sequences should be used to do the same thing, and even when they claim to agree, in practice, deviations are often found. Many computer vendors therefore tend to only offer support for their own terminals, which naturally cost more than competing products.

The Berkeley UNIX developers had no such hardware bias, and chose the "correct" way to handle terminal support—they defined a terminal capability database, `termcap`, and a set of utility programs to support it. `termcap` on most UNIX systems has entries for between 400 and 500 terminal types.

Programs, such as screen editors and spreadsheets, that use `termcap` need not contain any terminal-specific code, and require only that the user make the terminal type known to them. This is done with the `TERM` environment variable, which is conventionally set at login time by commands in the `.login` file, possibly after prompting the user to supply a type. For example, I am writing this paragraph on a VT100 compatible terminal, so I just need to type `setenv TERM vt100` if I am using `csh`, or `TERM=vt100; export TERM` if I am using `sh`.

By default, the database file is stored in `/etc/termcap` which is not writable by ordinary users. To allow testing a terminal description, you can put it in a local file, say `mytermcap`, then define an environment variable to point to it: `setenv TERMCAP /u/logindir/mytermcap`; an absolute pathname is *required* here.

Since the `termcap` file is reasonably large and therefore time consuming to read for every command that has to use `termcap`, the convention has been adopted that the database entry for the terminal type can be given instead in the `TERMCAP` variable itself. This is a long ugly string which you would never type by hand; the `tset` program will do it for you. The leading slash in the absolute path name for a private `termcap` file is used to disambiguate it from an actual `termcap` entry.

11 EOF and Logout

The end-of-file signal in UNIX is *Ctl-D*, rather than the Ctl-Z you are used to in TOPS-20 and other operating systems. To be recognized as such, it must be typed at the *beginning* of a line. The shells are just ordinary programs without any special privileges, and they too accept an end-of-file signal as part of the normal course of events, and when they get it, they terminate. This means if you type a Ctl-D to your login shell, you are logged out. If you intended to do that, well, it certainly is quick to type a single control character and be logged out. But suppose you thought you were executing a nested shell, or perhaps you typed ahead input to your own program, and then couldn't remember whether you gave it the end-of-file signal. Getting unexpectedly logged off is a nuisance in that you lose your command history and any environment changes, and if you logged in via a dialup or network, you have also lost that connection too.

Fortunately, the `csh` offers a way to prevent this unhappy accident; you just set the shell variable `ignoreeof` to an arbitrary value (e.g. `set ignoreeof`). The shell will then respond with "`Use "logout" to logout.`" if you type in a Ctl-D.

12 Command Aliases

The Bourne shell, `sh`, has no command alias facility; if you want to make an alias for some command, then you must create a command script file for that purpose.

`csh` has the `alias` command; you could type `alias tdir ls -lt` to make a command `tdir` which works like the TOPS-20 `TDIRECTORY` command. This is much faster than creating a separate command file, since no file system access is required. `csh` users tend to heavily customize their environments through the `alias` command. A bare `alias` will display the current list of command aliases.

13 Environment Customization

Many programs in UNIX permit the user to provide default startup options in a hidden file (one beginning with a leading period), usually in the login directory.

For example, the `login` program reads the `.login` file when you login to the system, just like the TOPS-20 `LOGIN.CMD` file is processed. Similarly, the `.logout` file is read and executed when you logout. Regrettably, UNIX has no concept of group-wide or system-wide login command files; this makes it necessary for each user's `.login` and `.profile` files to reference any group or system files explicitly.

`csh` reads startup information from the file `.cshrc`; this makes it analogous to the TOPS-20 `COMMAND.CMD` file. The corresponding startup file for `sh` is `.profile`. These files are the appropriate places to insert your personalized aliases and environment variables. Do not put them in the `.login` file, because that file is not read by spawned shells.

`Emacs` reads startup information from `.emacs`.

The `mail` program has `.mailrc` as its startup file, and will automatically forward mail sent to you to one or more addresses listed in the file `.forward`. Since `MM` on TOPS-20 is vastly superior to UNIX `mail`, I set my `.forward` file to cause all my mail on UNIX systems to be sent to TOPS-20.

In order to use `rcp`, `rlogin`, and `rsh` between different UNIX systems, you must establish a `.rhosts` file in your directory with the proper contents. See `man rlogin` for details.

14 Getting Help On-line

All but the smallest UNIX systems tend to have substantial on-line documentation, mostly in the form of manual pages. The original UNIX documentation consulted largely of Bell Laboratories reports, papers published in the Bell System Technical Journal, and short command descriptions formatted in a uniform syntax for the printed manual.

The printed manual was divided into 8 sections, with section 1 describing the main user commands, and section 3 the operating system interface; the remaining sections were devoted to more obscure commands, and system management tools. Newer releases of UNIX may have more manual sections, and sometimes these are further subdivided; e.g. section 3F describes the Fortran-callable operating system interface. UNIX documentation generally refers to manual pages by name and section, such as `ls(1)` and `hier(8)`.

The files that produce the printed manual are stored in directories `/usr/man/man1` ... `/usr/man/man8` as files suitable for input to the `troff` typesetter program, as well as for the `nroff` typewriter text formatter program. The `man` command searches these directories for the requested file, runs the required formatter on it, and displays the output text. For efficiency, the output is also captured and stored in a corresponding directory `/usr/man/cat1` ... `/usr/-man/cat8`; `man` actually searches these directories first in an attempt to avoid the formatting step.

`man`'s search order is by increasing section, and it stops with the first match it finds. Thus, if you type `man tty`, it will display the section 1 entry. There is also an entry in section 4 for this topic; to see it, you must type `man 4 tty`.

Sometimes there is a `SEE ALSO` entry on a manual page which points you to extended documentation in another section, but often there is not.

To find out what manual pages might be of interest, you can type `apropos keyword`, or `man -k keyword`; this will display the title lines of all files in the manual directories that contain the requested keyword. This information is actually stored separately in the file `/usr/man/whatis` to avoid having to search a large number of files.

`man man` will display `man`'s own documentation. `man 4 intro` shows the introductory page for section 4, which contains a useful summary of the section contents.

In desperation, you can resort to searching an entire manual page directory, e.g. `egrep "foo|bar" /usr/man/man1/*`.

15 Operators, Wheels, and Super-Users

Every operating system has some critical functions which only certain privileged users are permitted to execute. Obvious ones are shutting down the system, and accessing protected files. On TOPS-20, these are associated with OPERATOR and WHEEL capabilities normally restricted to systems personnel. In addition, they are not in effect until explicitly requested by the `ENABLE` command, and they are turned off by the `DISABLE` command. Also, more than one user can have such privileges, which is important when the staff exceeds one person.

UNIX reserves special privileges to a single login name, `root`, and that user is called the *super-user*. Because operating staff may have frequent need for super-user privileges, the `su` command is provided to allow spawning a new command shell for `root` from a running job; anyone who knows the password can issue this command. When the user logs out from this job, control returns to the original shell. This organization is unfortunate, because it makes it necessary for several people to know the same password, and it does not provide for a distribution of privileges across several levels of personnel.

16 Important Unix Tools

UNIX offers a bewildering variety of software tools, and it is hard for a novice to wade through their `man` pages and determine which are the most valuable ones to learn first. As a little experiment, I ran two commands on several different

UNIX systems. The first counts the number of commands in the three standard directories where system commands reside:

System	Command	File Count
Gould UTX-32	ls /bin /usr/bin /usr/ucb | wc -w	284
HP9000 HPUX	ls /bin /usr/bin /usr/ucb | wc -w	281
ISC 4.2BSD	ls /bin /usr/bin /usr/ucb | wc -w	295
Sun 3	ls /bin /usr/bin /usr/ucb | wc -w	302
VAX 4.3BSD	ls /bin /usr/bin /usr/ucb | wc -w	287

The second counts the number of commands in the standard local additions directory; most systems will have more than one of these.

System	Command	File Count
Gould UTX-32	ls /usr/local | wc -w	107
HP9000 HPUX	ls /usr/local | wc -w	6
ISC 4.2BSD	ls /usr/local | wc -w	42
Sun 3	ls /usr/local | wc -w	60
VAX 4.3BSD	ls /usr/local | wc -w	162

In summary, there are 300 to 450 different commands on a typical UNIX system, not counting the few dozen built-in shell commands.

Here then is a list of the *Top Twenty* commands you should learn first; those near the start of the list have higher priority.

- `man` and `apropos` — get help
- `csh` or `sh` — command shell
- `more` or `less` — page through command output (automatic on newer UNIX systems)
- `ls` — directory listing
- `cd` and `pwd` — change/print current directory
- `cat` — type or copy file
- `cp` — copy file or directory
- `mv` — move (rename) file or directory
- `mkdir` — make a directory
- `echo` — check shell command line expansion
- `emacs` or `vi` — screen editors
- `make` — build software

- `grep`, `egrep`, `fgrep`, `ngrep` — string search
- `sed` — automated text editing
- `awk` — text processing language
- `ar` and `ranlib` — source/object library utilities
- `diff` — source comparison
- `script` — log terminal session
- `rm` — remove (delete) file
- `rmdir` — remove (delete) directory

With this repertoire of a score of commands, you should be able to work quite successfully. Note that not a single compiler is listed here; `make` can handle that for you for simpler cases. Seasoned UNIX users value `make` highly, and ask it to direct much of their routine work.

17 Further Reading

Sun provides a number of Beginner's Guides which are useful for new users:

- Getting Started with UNIX
- Setting Up Your UNIX Environment
- Self Help with Problems
- Windows and Window Based Tools
- Mail and Messages
- Doing More with UNIX
- Using the Network
- Games, Demos, and Other Pursuits

There are now a great many introductory books about UNIX on the market, and you may wish to peruse local bookstore shelves, or come and examine my personal library. Here are some which I have found particularly valuable:

- Maurice J. Bach, *The Design of the Unix Operating System*, Prentice-Hall (1986) [advanced definitive treatise on the internals of UNIX]
- Brian W. Kernighan and Rob Pike, *The Unix Programming Environment*, Prentice-Hall (1984) [a *must* for every serious UNIX user]

- Marc J. Rochkind, *Advanced Unix Programming*, Prentice-Hall (1985) [a *must* for every systems programmer, or person writing code which must run in a variety of UNIX implementations]
- Kaare Christian, *The Unix Operating System*, Wiley (1983) [an intermediate treatment of UNIX tools and internals, without the detail in Bach's book]
- AT&T, *Unix System Readings and Applications*, Volumes 1 and 2, Prentice-Hall (1987) [a collection of reprints from the Bell System Technical Journal of key papers on the development and evolution of UNIX]

Index