Unix for the Minimalist

Duncan A. Buell
Department of Computer Science and Engineering
University of South Carolina
Columbia, South Carolina 20209
buell@cse.sc.edu

December 18, 2003

Abstract

This is intended to be a quick introduction to Unix and its theology and an introductory list of commands and features and the location of further information about them.

1 History

Unix was originally developed at Bell Labs in the early 1970s. Its more recent history is rather contorted. Unix Version 6 was the "established" version about 1976 and was used by both research and "production" types. The research types then refined Version 6 and produced Version 7. Neither system supported virtual memory, and both ran on PDP 11 machines. When the Vax 780 came out, the Computer Science Division at UC Berkeley arranged to be licensed to produce a virtual memory upgrade to Version 7 for the Vax; this version became known as Berkeley Unix. The Berkeley BSD4.2 became a "standard" Unix and is the model for many of the cloned Unix-like systems.

From the Version 6 code at Bell Labs came an upgrade, illogically named System III. This version, which also did not support virtual memory, was available on both PDP 11s and on Vaxen. The virtual memory upgrade to System III was then called System V, available for Vaxen and other machines and then cloned in several commercial ventures. The Cray Unix, for example, followed System V.

Unix from its earliest days was an operating system for computations in research, although some inroads into data processing were made. From the early days on DEC equipment, ports to Sun and other hardware became standard. Meanwhile, Microsoft, due to the great serendipity of its license in the early 1980s of DOS to IBM for "personal computers," came to dominate operating system use on Intel-based PC architectures. In the early 1990s, Linus Torvalds ported a version of Unix to an Intel machine, and Linux was born.

More recently, the difference between the two versions has become part of the litigation between SCO and IBM (and others). SCO claims that its license to Unix, which comes from the Bell license, includes code that made its way into the BSD versions and then into Linux by nefarious means.

At no time have the different versions of Unix had a major impact on their use. For the most part, the differences were of concern only to the systems programmers maintaining the systems, but not to the programmers doing applications that used the systems. This continues to be the case.

2 Files and File System Structure

The file system of Unix is simple, even simplistic. Logically, there is a tree structure of directories, each of which lives inside a parent directory and contains children which can be either directories or files. There are allegedly restraints on the creation of cycles in the directory structure; these restraints may actually work. Physically, the data structure that was the old Unix "file system" was a fixed-length array called an "i-list" of records called "i-nodes." As far as the data actually placed on disk is concerned, both directories and files are in fact files. The distinction between a directory and file is made by the program that accesses the data, since it must first find the i-node in the i-list, and in the i-node is contained the information that the disk data in fact represents a directory and not data itself. The fiel which is a directory contains data, *viz* a pointer to its own i-node, a file labelled "dot" (`.`), a pointer to the parent i-node, labelled "dot dot" (`..`), and then the names and pointers to the files and directories that are its children.

In sharp contrast to the Windows world, there is virtually nothing in Unix that assumes that the file names have any inherent meaning. In Windows, the three letter file extension (such as `doc`) has a meaning, and many programs will refuse to work on files that don't have the politically correct file extension.

Such things in general are not part of the Unix mentality. A file is a collection of bytes and nothing more. There is no inherent "meaning" to a file named `myletter.doc`, for example, that would be any different from the meaning attached to the same file if it happened to be named `The_U_S_Constitution`.

The root of the directory tree is a directory labelled "slash" (a single solidus `/`). Below this directory, the solidus is a delimiter and not taken to be a real character in a file name. File names can contain essentially any bytes at all, printable or unprintable. The length of a file name can be quite long, and the "full path name" is the complete file name from the root directory all the way to the individual file name itself. The full path name, for example, on the departmental system to a student whose login id was `smith` might well be `/acct/s1/smith`.

Every user has a default "home directory" into which he/she is placed on logging in. The usual practice is for this to have the same name as the login id. In most shell programs, the abbreviation for the path to the home directory is the tilde (`~`),

In your home directory there are several files, called "hidden files," that you get to set and that control what your environment looks like. The "dot forward" file `.forward` is the file into which you place the email address to which you want email to be forwarded. For example, if your login was `smith` your default CSE department email address would be `smith@cse.sc.edu` but if you wanted all your mail to be forwarded to `flowerchild@yahoo.com` then you would place that address in your `.forward` file; that is, the `.forward` file would consist of one line that was exactly `flowerchild@yahoo.com`.

A second hidden file that is relevant to your environment is the `.tcshrc` file. The "shell" program is a program called `tcsh`, and the `.tcshrc` file is what configures your shell environment. For example, a line that reads `alias h history` in your `.tcshrc` file sets the single letter `h` as an alias for the `history` command; this command displays for you the most recent commands you have issued to the shell.

Note also that file names can be wildcarded with an asterisk. File `outfile*` refers to any file beginning with the string `outfile`, for example `outfile1`, `outfile2`, `outfile.doc`, `outfile.data`, `outfile.whatever`. The file name `out*.*` refers to any file beginning with the string `out` followed by zero or more characters, followed by a period, followed by zero or more additional characters.

2.1 Windows and Unix File Compatibility

I mentioned above that a Unix file is a collection of bytes. For the most part, textual files (including programs and the Unix analog of the Windows `txt`) files consist of *lines* of ASCII characters terminated with a "newline" character, which is `0A` in hexadecimal, `10` in decimal, `12` in octal. The Windows Notepad and Wordpad and editing programs terminate lines with *both* a newline and a "carriage return" character (`0D` in hexadecimal, `13` in decimal, `15` in octal). If you were to edit a file saved by Wordpad, for example, in the `vi` editor, these extra carriage return characters will show up as "control-M" characters `^M` at the end of every line.

If, for example, if you have a Unix file that consists of nothing more than the single word `test`, then that file will have five bytes in it (the four letters of the word followed by the newline). If you edit that file in a Unix editor, and change the word `test` to the word `tests`, then the saved file will be six bytes long. If you edit the file with Wordpad, the file will be saved as a seven-byte file, because the carriage return will be added.

This is a minor annoyance in going back and forth between Windows and Unix. The Notepad editor will *not* read Unix files correctly, but the Wordpad editor will read them without problems.

It is possible to set your Windows editor to save off without the extra carriage returns, but that may also affect your ability to edit those files later with a different Windows editor.

In Unix, if you have saved off a file with a Windows editor and you wish to strip off the carriage return characters, you can use the utility `scc`. Issuing the command

```
scc file_name
```

will cause the carriage returns to be removed from the file. Note that this changes the original file; it does not create a copy with the characters removed.

3 File Permissions and Access Control

Control of access to files in Unix is at the *user*, *group*, and *world* (or "other") level. You are the user. You are probably in a group. For example, faculty are in the "faculty" group. Students are in the "student" group. You can

be in many groups; one could set up a different group for a specific class or research project. The world is everyone with an account, that is, universal access. Files that are intended to be read by anyone, such as home pages on the Internet, should be set to be world-readable.

Every Unix file contains ten access control bits. These can be seen by issuing an `ls -l` command on the file name. The first bit indicates whether or not the file is actually a directory. Thus

```
ls -l some_directory_name
```

returns

```
drwxr-xr-x    2 buell    faculty       512 Dec 18 10:37 some\_directory\_name/
```

with the first character set to `d` for directory, while

```
ls -l some_file_name
```

returns

```
-rw-r--r--    1 buell    faculty         0 Dec 18 10:36 some\_file\_name
```

that character set to a minus sign. (Well, not really, but that's not something I want to get into right now.)

The next nine bits are the read, write, and execute permissions for each of the user, group, and world, in that order. The string `-rw-r--r--` above for `some_file_name` indicates that the user, the group, and the world can all read the file, but that only the user can write to the file (and thus change it). The string `-rwxr-xr-x` above for `some_directory_name` indicates that the user, the group, and the world can all read and execute the file, but that only the user can write to the file. It is a quirk of the Unix file system that one must be able to "execute" a directory in order to descend into it. Thus all directories by default have the execute permission set for them.

4 Some Useful Commands

4.1 Options on Unix Commands

Nearly all Unix commands permit options. Options to the command, as opposed to arguments to the command, are almost invariably indicated with a minus sign. For example

```
head some_file_name
```

displays the first ten lines of the file `some_file_name`. If what you want is not ten lines, but some different number of lines, then that number is indicated with a minus sign followed by the number of lines. Thus

```
head -17 some_file_name
```

displays the first seventeen lines of the file.

The sort command is another command that has some useful options. A sort command issued in the most vanilla form as `sort this_file_name` to sort a file `this_file_name` and display the sorted result to the window. This sort is done lexicographically (dictionary order) on the file. The result of sorting a file consisting of the lines

```
2
3
4
10
1
```

results in

```
1
10
2
3
4
```

because the sort is done character by character from left to right. If what you want is to sort these lines as numbers, you specify a *numerical sort* with the `-n` option. The command `sort -n this_file_name` will result in the display of

```
1
2
3
4
10
```

when run on the same file.

Some *options* themselves take arguments. For example, one compiles a C program `myprogram.c` by issuing the command

```
cc myprogram.c
```

By default, the C compiler translates `myprogram.c` into object code and then into an executable file and names the executable file `a.out`. If you wish to have the executable file named something other than `a.out`, perhaps `myprogram.exe`, then you could either issue a

```
mv a.out myprogram.exe
```

command or else tell the compiler to do the renaming for you by adding the `-o` (for *output*) option to the compile command. This would be done as

```
cc -o myprogram.exe myprogram.c
```

4.2 File Commands

- `cat` (*catenate a file to the window*): The command

  ```
  cat some_file_name
  ```

  causes the file `some_file_name` to be displayed to the window. This is the standard way of seeing what's in the file, and is the command analogous to double-clicking on a file name in Windows. (See also `head`, `more`, `tail`).

- `chmod` (*change mode (privileges) of a file*): The command

  ```
  chmod whoXwhat some_file_name
  ```

  causes the privileges and use of the file `some_file_name` to be changed. In this, `who` must be some combination of one or all of `u` (for *user*—yourself), `g` (for *group*—those in your group), or `o` (for *other*—everyone else). The `X` part of the string above must be either `+` to *add* the privilege to or `-` to *subtract* the privilege from the file. The `what` must be some combination of one or all of `r` (for *read*), `w` (for *write*), `x` (for *execute*). Thus, for example,

```
chmod go-rwx some_file_name
```

causes all privileges (read, write, and execute) to be removed from the file `some_file_name` for logins in the group and for all other logins. The privileges on the file belonging to the owner (usually you) will be unchanged.

- `cp` (*copy a file*): The syntax is *copy from_name to_name* so that, for example,

```
cp this_file_name that_file_name
```

creates a copy of the file `this_file_name` and names the copy `that_file_name`. (See also `mv`, `rm`, `rmdir`).

- `head` (*display the head of a file*): This works the same as the `cat` command except that only the first ten lines of the file are displayed. The number of lines displayed can be specified in the command line. For example,

```
head -23 this_file_name
```

will display the first 23 lines of the file `this_file_name`. (See also `cat`, `more`, `tail`).

- `ls` (*list files*): The command

```
ls this_file_name
```

simply displays the basic information about a file. If the file is present in the current directory, then the file name is displayed. If the file is not present in the current directory, then the message

```
ls:  this_file_name:  No such file or directory
```

is displayed. Of course, this information isn't entirely useful. What is more useful is the use of this command with some of its options The `-l` option causes the *long form* of the file information to be displayed. This includes the privileges, the owner of the file, the group having

privilege to access the file, the number of bytes of the file, the date and time of the last edit of the file, and the file name. At the time this document is being written, the file name of the file that is this document is `unix_minimalist.tex` and issuing the command the

```
ls -l unix_minimalist.tex
```

causes the information

```
-rw-r--r-- 1 buell buell 12022 Dec 17 21:22 unix_minimalist.tex
```

to be displayed. The command

```
ls -l *
```

causes this information to be displayed for all the files in the current directory.

- `mkdir` (*make (create) a directory*): This is the unix analog of the "New Folder" option of Windows. Creation is done in the current directory unless a full path name is specified.

- `more` (*more*): The command `cat` displays an entire file, `head` displays the first lines, and `tail` displays the last lines of a file. In contrast, `more` displays a file one window full of lines at a time. The program then pauses waiting for your input. Hitting the space bar causes another window full of lines to be displayed. Hitting the carriage return or enter key causes one more line to be displayed. Issuing an interrupt (control-C) kills the `more` and returns control to the command line in the window. (See also `cat`, `head`, `tail`).

- `mv` (*move a file*): This is the renaming command. One does not "rename" files in Unix, one moves them from one name to another. The syntax is like `cp`. If moving is what is desired, remember that the last file name specified is taken to be the "to" name. It has the effect of the Windows cut and paste or the rename command. FEATURE/WARNING: You can list a number of files in one command (analogous to highlighting a number of file names in Windows) and put a directory as the last name in the list on the command line. For example,

```
mv file1 file2 file3 new_directory
```

would move all three files, named `file1`, `file2`, and `file3`, into the directory `new_directory`. If in fact `new_directory` is not a directory but a file, some older versions of Unix will mash each of the files successively onto a file named `new_directory`, so that all but the last file is lost. New versions of Unix usually don't let you do this, since this is likely to be an error on your part. (See also `cp`, `rm`, `rmdir`).

- `rm` (*remove a file*): No need to explain? Except that this command only removes files, not directories. WARNING: If you issue the command `rm *` you are asking for the deletion of *all* the files in the current directory. My suspicion is that `rm *` is a command that should be used only very occasionally. (See also `cp`, `mv`, `rmdir`).

- `rmdir` (*remove a directory*): Now there is in fact no need to explain. (See also `cp`, `mv`, `rm`).

- `tail` (*display the tail a file*): This works the same as `head` except it display the *last* ten (or some other specified number) lines of the file. (See also `cat`, `head`, `more`).

4.3 Directory and Location Commands

- `cd` (*change directory*): This has several variations. The command `cd ..` will change to the parent directory (that is, pop up one level in the tree structure), since `..` is the parent. If the argument is a file name (which must be a directory name in order to make sense), it is assumed to be in the current directory unless you have given a complete directory path beginning with `/` or its equivalent `~`.

- `pushd` (*push to directory*): Sometimes it happens that you are in a particular directory and you wish to change the directory to some other particular directory, but you also want to go back where you came from. The command `pushd new_directory` does a `cd new_directory` but retains on a stack the old directory name, which for argument's sake we will call `old_directory`. If you then finish what you are doing in `new_directory` and then want to return to `old_directory`, this can be done by issuing a `pushd` command with no arguments. A series

of `pushd` commands with no arguments simply toggles back and forth between the two directories. (This can be useful, for example, if you wish to keep each programming assignment in a separate directory. You may, however, want to go back and forth to be able to copy a `makefile` from the previous programming assignment and use it for the new programming assignment, or to check how you did something in the previous assignment and use the same technique in the new assignment.)

- `popd` (*pop from directory*): This command is the "undo" of the `pushd` in that in takes you back to the directory from which you did the `pushd` and gets rid of the memory of where you had been.

- `pwd` (*print working directory*): This simply displays the full path name of the current directory.

4.4 Miscellaneous Commands

- `apropos`: Sometimes it happens that you know that a command exists that does more or less what you want, but you cannot remember the command name, so you don't know how to issue the exact `man` request. For example, you know for sure that there are logarithm functions in the mathematics library, but maybe you are not sure what the name would be for the decimal logarithm function. The `apropos` command can help you here. A command

```
apropos logarithm
```

  returns a list of the commands in the manual pages in which the word "logarithm" appears. Note: this command functions much like `grep` in that if you use the word "logarithm" it will try to match the entire word but if you use the shorter string "log" it will match the shorter string and thus maybe give you more answers (in case the man page you want uses the term "log" but not the longer word "logarithm." (See also `man`).

- `cmp` (*compare*): The `diff` command compares two files line by line and reports lines that differ. This works for files that are essentially just ASCII text like programs and text files, but it does not work

very well for files that have nonprintable characters in them. (Most Windows files, for example, will have embedded nonprintable bytes, and as mentioned above their "lines" are not terminated by the carriage return as are the lines of Unix files.) The command

```
cmp file_one file_two
```

will compare `file_one` and `file_two` character by character and report either that they are byte-by-byte identical or that they differ at some character location. (See also `diff`).

- `diff`: This indispensable command lets you compare two files and display the lines that differ from one file to the next. For example, given a file `file_one` containing

  ```
  one
  two version one
  four
  ```

  and a file `file_one` containing

  ```
  one
  two version two
  three
  four
  ```

  the command

  ```
  diff file_one file_two
  ```

  will result in the display

  ```
  2c2,3
  < two version one
  ---
  > two version two
  > three
  ```

The `2c2,3` means that line 2 of the first file differs from lines 2 and 3 of the second file. Specifically, the line `two version one` of the first file (indicated by the "less than" symbol) differs from the lines `two version two` and `three` of the second file (indicated by the "greater than" symbol).

The major use of this command is to *version control*. If you have two copies of what look like the same program, and you cannot remember which one is the right one, this command will let you see where to look to find the differences.

(See also `cmp`).

- `grep`: This stands for *get regular expression and print*. There are lots of things you can do with this command. Perhaps the most useful is the following. If you issue the command

```
grep grizzle this_file_name
```

will display every line in the file `this_file_name` in which the character string `grizzle` appears. The command

```
grep -v grizzle this_file_name
```

will display every line in the file in which the string does *not* appear. This can be especially useful in debugging programs; putting a keyword in the debugging line allows one to `grep` for it to determine whether error and special case conditions have happened, and to put in multiple keywords to be able to separate the output into various kinds of usable pieces.

- `man` (*manual page*): This is the Unix equivalent of the Windows "help" button. The command

```
man sort
```

will, for example, display the official manual page for the `sort` command. This is useful for, among other things, figuring out what the options on a command are and how to use them. (See also `apropos`).

- `sort`: The command

  ```
  sort this_file_name
  ```

  causes the file `this_file_name` to be sorted lexicographically. The command

  ```
  sort this_file_name >that_file_name
  ```

  causes the file `this_file_name` to be sorted lexicographically and the sorted output to be placed in the file `that_file_name`.

- `wc` (*word count*): This gives word, line, and byte counts of the argument file. Thus,

  ```
  wc unix_minimalist.tex
  ```

  run on the source text of this particular document at the time I am writing this particular section returns

  ```
  861 4037 24937 unix_minimalist.tex
  ```

  indicating that the file has 861 lines, 4037 words, and 24937 bytes in the file at the time the command was issued.

5 Standard Input, Output, Error, and File Redirection

Every process in Unix has a *standard input* file `stdin`, a *standard output* file `stdout`, and a *standard error* file `stderr`. These are normally assigned to the window, but can be redirected, as in the description of `sort` above. That is, `sort` would normally take the input from the window, sort it, and display it back to the window. Adding the `<fromfile` causes it to take its standard input from the file `fromfile`. Adding the `<tofile` causes it to send its standard output to the file `tofile`. Either or both input and output can be redirected. Standard error can also be redirected. Redirecting output as in `sort this_file_name >>that_file_name` with the double `>>` will cause

the output to be *appended* at the end of the file `tofile`. This is especially useful for such things as loging progress of a program or collecting error or special condition information from several locations into one file.

There are also things in Unix called "pipes." The command `grep ERROR filename | wc -l` would cause every line of `filename` in which the string `ERROR` occurred to be sent to standard output. The standard output in this case is piped, however, using the vertical stroke, to the standard input of `wc -l`, so that the result of this combined command is to display in the window the number of lines in the file `filename` in which the string occurred.

6 Make

One of the most useful utilities of Unix is `make`, which makes it easier for programmers to produce large programs of several files and compile and link them all together. The basic purpose of `make` is to decide which files have changed since the last compilation and only to recompile those files, not necessarily all the files within the scope of the `make` command.

An example `makefile` is the following.

```
A = main.o
B = sub1.o sub2.o
C = sub3.o
Aprog: \$A \$B \$C
fc \$A \$B \$C -o Aprog
main.o main.f
fc -c main.f
sub1.o sub1.f
fc -c sub1.f
sub2.o sub2.f
cc -c sub2.c
sub3.o sub3.f
cc -c sub3.c
```

This is interpreted as follows. Symbol `A` is defined to be `main.o`, and symbols `B` and `C` are similarly defined. In order to produce the file `Aprog`, the `make` program must have symbols `A`, `B`, and `C` defined and up to date. If they are up to date, then `Aprog` is produced by executing `fc $A $B $C`

`-o Aprog`, that is, compiling the object codes `main.o`, `sub1.o`, `sub2.o`, and `sub3.o`, and naming the output `Aprog`.

In order to produce the file `main.o`, `make` must have an up to date version of `main.f`. If it does, then `main.o` is produced by executing `fc -c main.f`, that is, compiling but not linking the file `main.f`. And so forth.

In the example above the symbols I have used, such as `A`, are single letter symbols, and when they are referred to later they can be referred to as `$A`. It is perfectly permissible to use multiple-character symbols, such as `MYSYMBOL`, but when they are referenced later one needs to enclose the character string in parentheses, as in `$(MYSYMBOL)`.

In the command portion of a makefile one can have more than one line, and not just the one line per symbol of the sample `makefile` here. Anything that is a legal command is possible, one line after another.

THE CRUCIAL THING TO REMEMBER ABOUT `make` IS THAT THE FIRST CHARACTER IN A COMMAND LINE (THE LINES IN THE EXAMPLE ABOVE THAT ISSUE THE `fc` OR `cc` COMMANDS) ABSOLUTELY MUST BE A TAB CHARACTER, AND NOT SPACES. Using blank lines will result in the rather obscure error message

```
*** missing separator.  Stop.
```