A Brief Introduction to Unix

With Emphasis on the Unix Philosophy

And How to Apply it to Do Your Work

by

Corey Satten
corey@cac.washington.edu
Networks and Distributed Computing, CAC
University of Washington, HG–45
Seattle, Washington 98195

Overview

Unlike a traditional introduction to Unix, the emphasis of this one is on philosophy and brevity. When you understand how the creators of Unix intended you to use it, you'll approach Unix on it's "best side". This introduction intends to help a new Unix user get started on the right foot quickly. For more information, readers are referred to the Unix manuals and other listed references. As little detail as possible has been duplicated from the manual.

Copyright © 1989, University of Washington

Commercial use of this document requires permission from NDC

1. Why Use Unix?

In some ways, Unix is "old technology" — it was invented in the late 1960's for a small computer with a 64K-byte address space, it is largely character oriented (not graphic). Why is it still here? Why is it spreading to more and more systems from PC's to Cray Supercomputers? One answer is that Unix is written in a mostly machine independent way (in the high level language "C") and is therefore more easily moved to new machines. Once Unix has moved, a large base of applications also moves easily and your investment in learning Unix continues to pay off. Another answer is that many problems are still character oriented (or at least can be approached that way) and for these problems, like a sharp tool in the hands of a skilled user, Unix really helps you get your work done. Also, you can use Unix from any kind of terminal and over dial-up phone lines or computer network connections.

In the space below, I hope to convey, with a minimum of specific information, the essence of "The Unix Philosophy" so that you can use and enjoy Unix at its best. To try to summarize in just two sentences (for those who really believe in such brevity): Unix comes with a rich set of connectable tools which, even if they don't directly address the problem at hand, can be conveniently composed (using the programmability of the command interpreter) into a solution. Unix also imposes relatively few arbitrary limits and assumptions on the user or the problem domain and has thereby proven to be a suitable platform on which to build many useful and highly portable research and commercial applications.

2. Essential Commands and Concepts

Before I can realistically hope to say more about Unix in general, or give meaningful examples, I must briefly explain some Unix commands and concepts. These descriptions are intentionally minimal. You will soon see how to find more detail in the manuals.

2.1. Login

Unix is a multi-user operating system. This means that several users can share the computer simultaneously. To protect each user's data from damage by other users, Unix requires each user "*login*" to the system to identify him/herself (with a login name) and authenticate him/herself (with a password). During the login process, a user's defaults and "*terminal type*" are usually established. The mechanism Unix uses to allow concurrent users also allows each user to have more than one program (also called "process" or "commands") running concurrently. You will see shortly how convenient this is.

2.2. The Shell, Commands and Arguments

Once you have logged in, you will be running a program called your "*login shell*". The shell is a program which executes the commands you type in and prompts you when it is ready for input. One of the nice features of the Unix shell is that it is a powerful programming language unto itself, however one need not program it to use Unix. There are several different "shell" programs in common use: *csh* (c-shell), *sh* (bourne-shell), *ksh* (korn-shell), *vsh* (visual-shell) to name a few. Most people use "csh".

Unix commands consist of a program name followed by options (or arguments) to that program (if any). One or more spaces follow the program name and separate arguments. Each program examines its argument list and modifies its behavior accordingly. By convention, arguments which begin with a dash are called "switches" or "flags" and they are used to request various non-default program behavior or to introduce other arguments. It is occasionally important to remember that it is the ***shell*** which does filename expansion (such as turning "*.old" into "a.old list.old program.old"). Programs normally don't ever see un-expanded argument lists. Many Unix programs can also take implicit arguments. These are available (to every program you run) via the "*environment*". Your "terminal type", stored in an environment variable called TERM, is an example of this. The manual for each program you use should list the environment variables it examines and the manual for your shell explains environment variables in detail.

2.3. On-line Manuals

Before getting into any specific commands and examples, note that most Unix systems have both on-line and printed manuals. Many commands will be mentioned below in passing without explanation. It is assumed that the interested reader will look them up in the manual.

The on-line manuals generally contain only the numbered sections of the printed manuals. The tutorials and in-depth articles are usually only in printed form. This introduction intends to reproduce as little of the information contained in the Unix manuals as possible. For more information on any Unix *command*, type "man *command*" ("man man", for example gets you "the man-page" for the on-line *man*ual command: *man*). (Note: if you are prompted with the word "more", you are interacting with the "*more*" program. Three quick things to know: you may type a space to get the next screenful, the letter "q" to quit, or "?" for a help screen.)

Among other things, the man-page for the "man" command points out that "man -k *word*" will list the summary line of all on-line man-pages in which the keyword: *word* is present. For example, "man -k sort", will produce something like this:

comm (1)	- select or reject lines common to two sorted files
look (1)	- find lines in a sorted list
qsort (3)	- quicker sort
qsort (3F)	- quick sort
scandir, alphasort (3)	- scan a directory
sort (1)	- sort or merge files
sortbib (1)	- sort bibliographic database
tsort (1)	- topological sort

This tells you that section 1 (user commands) of the manual has man-pages for *comm, look, sort, sortbib, tsort*. Use the man command on any of these to learn more. The other numbered sections of the Unix manual are for system calls, subroutines, file formats, etc. You can find out about each section of the manual by saying, for example, "man 2 intro". Enough about manuals.

2.4. I/O re-direction: stdin, stdout, stderr, pipes

By convention, whenever possible, Unix programs don't explicitly specify from-where to read input or to-where to write output. Instead, programs usually read from "*standard input*" (*stdin* for short) and write to "*standard output*" (*stdout*). By default, standard input is the keyboard you logged in on and standard output is the associated display, however, the shell allows you to re-direct the standard output of one program either to a "*file*" or to the standard input of another. Standard input can be similarly redirected. Perhaps Unix's greatest success comes from the ability to combine programs easily (by joining their standard inputs and outputs together forming a pipeline) to solve potentially complex problems.

"*Standard error*" (*stderr*) is not usually re-directed, hence programs which write warnings, prompts, errors, etc. to stderr will write them to the display even when normal input and output is usefully re-directed. (Note that since I/O devices are implemented as files on Unix, I/O re-direction also works to and from physical devices.) The syntax for I/O re-direction is fully described in the manual for the shell you are using (probably csh).

The following are some simple examples of I/O re-direction. For clarity, the shell's ready-for-input-prompt has been shown as "Ready%" and explanations have been inserted in *italics*. Everything the user would type is shown in slightly bold type after the Ready% prompt.

Running the "date" command prints today's date and time on standard output

Ready% **date**
Wed Mar 22 13:06:30 PST 1989
Ready%

Put the standard output from the date command in a file called "myfile"

Ready% **date > myfile**
Ready%

Use the word-count program to count the number of lines, words, characters in "myfile"

Ready% **wc < myfile**
      1   6   29
Ready%

Pipe the output of the date command directly into the word count command. Note that commands in a pipeline such as this can run simultaneously.

Ready% **date | wc**
      1   6   29
Ready%

Use output from one program as command line arguments to another

Ready% **echo My computer, `hostname`, thinks today is `date`**
My computer, samburu, thinks today is Wed Mar 22 13:06:30 PST 1989
Ready%

Look in the on-line dictionary for words beginning with "pe" and count how many are found

Ready% **look pe | wc**
294 294 2548
Ready%

Pipe those 294 lines through cat -n to insert line numbers and then through sed to select only lines 5-8

```
Ready% look pe | cat –n | sed –n 5,8p
     5	peaceful
     6	peacemake
     7	peacetime
     8	peach
Ready%
```

Now, from those 294 words, select only those containing "va" somewhere and re-direct them into the argument list of the echo command

Ready% **echo I found these: `look pe | grep va`.**
I found these: Pennsylvania Percival pervade pervasion pervasive.
Ready%

Grep (search) through all files with names ending in ".c" for lines beginning with "#define". (Grep -l lists the file names containing the lines which match instead of the lines themselves). These file names are redirected to form the command line of the vi editor — hence, edit all ".c" files which contain "define" statements.

Ready% **vi `grep ^#define *.c`**

The depiction of an interactive session with the "vi" editor is omitted.

Ready%

2.5. Special characters: Interrupt, End-Of-File, Quoting, 'Job Control'

When a program reads from a file or from a pipe it can tell when there is no more to read. This condition is called reading the "*end-of-file*" or EOF. When standard input is a terminal, the EOF must be explicitly typed because the program must otherwise assume you are still typing. Normally EOF is typed as a CONTROL-D (indicated in print as ˆD). Think of the control key as another SHIFT key — it must be pressed and held when the D is typed. If the EOF is not the first thing on a line, two must be typed.

If you are running a program and you wish to interrupt it completely, you can often do so by typing ˆC. You can try this with the "wc" program:

run wc then interrupt it

Ready% **wc**
sample input
ˆC
Ready%

run wc then type EOF

```
Ready% wc
sample input
^D
      1       2      13
Ready%
```

Note that both ˆD and ˆC ended the program however, ˆD allowed the program to finish normally but ˆC killed it (and produced no output). If, for some reason, you want to type a special character such as ˆC and actually have it sent to your program and not generate an interrupt, you can "*quote it*" by typing a backslash (or sometimes a ˆV) before it. The backslash also "quotes" shell "*meta-characters*" such as asterisk, question mark, double-quote, backslash, etc.

"*Job control*" is the name given to an extremely convenient feature of many modern versions of Unix. Job control allows one to suspend a program and resume it later. If you are in the middle of running some program when the phone rings, you can type ˆZ to suspend the program (and get back to your shell prompt) without interrupting or exiting that program. After you handle the phone call, you can type "*fg*" to resume the original program right where you left off. Unix permits one to have a fairly large number of suspended jobs and to resume them in any order. Csh's "*jobs*" command displays which jobs are stopped. (In some ways, job control is "a poor man's window system"; however, even on Unix systems with windows, many people find job control indispensable.) For more information on job control, see the "csh" man-page.

2.6. Files, permissions, Search PATH

Unix files exist in directories. Every user has a "*home directory*", which is the "*current directory*" after logging in. A user can make "*sub directories*" with the "*mkdir*", command and make them the current directory with the "*cd*" command. You can print your current directory with the "*pwd*" command and you can refer to the parent directory as ".." (two dots). You can get back to your home directory by typing "cd" with no arguments.

Files and directories have permissions called "*modes*" which determine whether you, "*your group*", or everyone can: read, write, or execute the file. Permissions are changed with the "*chmod*" command. The main reason for bringing this up now is to point out that a collection of commands which can be typed to the shell can also be put in a file, given a name, made executable and subsequently invoked as a new command by that name. This type of file is called a "*shell script*" and is one of the main ways Unix is customized to the work habits and chores of its users.

When a user types a command, s/he usually doesn't type the full (and unambiguous) path name of the program: (*/bin/date* for example) but instead types only the last component of the path name, *date*, thus requesting the system to search for it. To achieve predictability and efficiency, the system searches only those directories listed in your PATH environment variable and it searches them in that order. By placing your own version of a program in a directory you search before the system directories, you can override a system command with your own version of it. Your version can be anything from an entirely different program to a simple shell script which supplies some arguments you always use and then calls the standard version. The command "echo $PATH" will print the value of the PATH environment variable to stdout. The procedure for setting environment variables such as PATH differs from shell to shell. See the man-page for the shell you use.

3. The Unix Philosophy

Well, so much for the nitty-gritty. I will now try to explain "The Unix Philosophy" in a bit more detail. Basically, the idea is that rather than have a custom program for each little thing you want to do, Unix has a collection of useful tools each of which does a specific job (and does it well). To get a job done, one combines the pieces either on the command line or in a shell script. For example, on Unix, a user would not expect an application to provide an input text editor. Instead, one would expect to be able to use one's favorite (and standard) "*text editor*" (probably "*vi*", perhaps "*emacs*") for all instances of editing text. Electronic mail, C programs, shell scripts, documents-to-typeset can all be edited with the same text editor. By convention, applications invoke the text editor you have specified in your EDITOR environment variable.

Even though Unix editors are generally very powerful and capable programs, they too recognize that they are just tools and they allow you to pipe all or part of your "*editor buffer*" through any pipeline of Unix commands in order to do something special for which there isn't a built-in editor command. (The editor buffer is that private copy of your file to which the editor makes changes before you save them.)

Unlike most other operating systems, Unix has only one "*file type*". Any program which can read or write standard I/O can read/write any "file" (even if it is a device such as a terminal, printer or disk). Granted, not every program can make sense out of the data in every file, however, that is strictly between the program and the data — nothing imposed by Unix. The single file-type contributes greatly to the modular/re-usable pipes-and-filters approach to problem solving.

So, what is to be learned from all this? Just that it is good to construct solutions to your problems in as general and modular a fashion as possible. You will undoubtedly find that a somewhat general program (or shell script) you wrote as part of the solution to one problem will be just what you need as part of the solution to some future problem and it will be simple to hook up.

4. A 'Typically Unix' Solution

Let's assume the following problem, inspired by a real-world situation. You are a professor of English and someone walks into your office with an old manuscript claiming it is an undiscovered work by Shakespeare. You postulate (correctly) that you can use statistics about frequency of word usage to help determine its authenticity. The problem, therefore, is to come up with a histogram (count) of the number of times each word is used.

You could, of course, write a program from scratch in C or FORTRAN to do it, however a partial solution comes to mind using "*awk*", a programmable text processing tool which has 2 particularly useful features: 1) lines are read and processed automatically; 2) arrays can have text-string subscripts. So, if you hadn't already written a "histogram" shell script, you write one now. (Keep it around, you will find a use for it again.) The file "histogram" has the following contents (de-mystified somewhat below):

```
awk '
NF > 0      { counts[$0] = counts[$0] + 1; }
END         { for (word in counts) print counts[word], word; }
'
```

For each line with NF > 0 (NF is awk-talk for number-of-fields-on-this-line, hence for each non-empty line), add 1 to that particular counter hereby associated with the-text-on-this-line ($0 is awk-talk for the-text-on-this-line). Then, at the END of input, for each unique input line; print that line preceded by the count of how many times it was seen. (*Note that even though the preceding solution* "*smacks of programming*"*, it is simple. Thus, even if you don't attempt it yourself, the fact that the solution is simple means that you will have a much easier time finding someone else to do it for you.*)

So, now the task is simply getting the input into a format where all punctuation marks are removed and each word appears on a line by itself. Again, you could write a program to do it; you could manually reformat the text with an editor; or you could notice that Unix has a translate command "*tr*" which will do just what you want when used in two steps as shown:

```
tr –dc "a–zA–Z' \012" | tr " "  "\012"
```

The first "tr" command has options -dc (delete the complement of the indicated characters) so it will delete from standard input all characters except those which are listed (letters, apostrophe, space, and octal 012 (newline)). The resulting output has no punctuation. The second "tr" translates all spaces into newlines, thus causing at most one word to be on each line.

Piping the output of these two commands into "histogram" will give us word counts. Piping the output of histogram into "sort -n" will sort the histogram in numerical order. Putting the whole thing in a file and making it executable makes it available as conveniently as if it had been built into Unix.

Here then is some sample input and the output our script produces:

```
One black bug bled blue black blood
while another blue bug bled black.
```

And the output of tr ... | tr ... | histogram | sort:

```
1 One
1 another
1 blood
1 while
2 bled
2 blue
2 bug
3 black
```

Note that other simple solutions to the problem exist. Our awk-based histogram program can be replaced by "sort | uniq -c" (but that is less intuitive than the awk solution and not necessarily any better). Also, "*sed*" could have been used in place of either or both of the "tr" commands. (Sed is much more powerful than tr however the sed command line would have been less intuitive.)

5. More about Pipelines and Concurrent Execution

Probably the two biggest advantages of concurrent execution of commands in a pipeline are: 1) No disk space is required for intermediate data which flows through pipelines. 2) output can start coming out the end of the pipeline before the entire input is processed through the first program in the pipeline.

For example, imagine you want to compute a histogram for a very large file which is compressed and your disk is too full to hold the uncompressed version. You can uncompress it to standard output and pipe that directly into your histogram pipeline.

Now imagine you have a pipeline which takes 30 minutes to compute and produces data which takes 30 minutes to print. If you first computed and then printed, it would take 60 minutes. If you re-direct the output of the pipeline to the printer, the whole process only takes 30 minutes. (Note: you *can* output directly to a device such as a printer but in a multi-user environment the normal printing mechanism is to spool the output in a file (with "*lpr*") and print it after the computation finishes.)

On Unix you can run any number of programs "*in the background*", which means that the shell doesn't wait for them to finish before giving you a new prompt. Read more about this in the manual for your shell.

You can also have programs started for you automatically at certain times of the day, week, month, etc. (read about "*at*" and "*cron*") or when certain events happen, such as when electronic mail arrives.

6. Other Especially Useful Unix Programs to Read About

Since it is not the intent to duplicate information from the Unix manual in this introduction I won't give many details about the following programs, however, I would like to point them out so you can look them up in the manual if you are interested. Most manual pages have a "SEE ALSO" section at the end. Consider yourself invited to read up on those programs as well. (If you really want to know everything, look up every program in every directory in your $PATH!)

learn	An interactive tutorial on a few subjects. (Not available on all systems). Probably most useful for learning the "vi" editor. Type "learn vi" to try it.
vi, emacs, ex, ed	"vi" is the most common Unix screen-oriented text editor. Emacs can be another good choice. ("ed", the original Unix text editor is essentially subsumed by vi and is much harder to use.) "ex" is really just vi in a non-screen-oriented (ed-like) mode. There are substantial printed manuals on vi, ex and emacs. Whichever editor you choose, you will eventually want to read everything there is to know about it. Unix editors are very powerful and knowing how to use that power really helps a lot.
rm, mv, cp, rmdir	Remove; move (rename); copy a file. Remove a directory.
ls	List directories. More options than just about any other program. Filenames which begin with dot are not listed unless the -a option is used.
stty, tset	Set such aspects of terminal I/O as: number of lines on display device, input character- or line-at-a-time; whether keyboard typing is visible.
cat	Concatenate files to standard output.
more, less, page, pr	Display data a screen or page at a time. Search and skip forward to a page of interest.
cmp, comm, diff, diffmk	Show differences between 2 files.
grep	Find lines which match specified pattern. Incredibly useful.
rlogin, rsh, rcp	Login to remote Unix system, run a command on remote Unix system, copy a file to remote Unix system. Similar to below.
telnet, ftp	Connect to remote system of arbitrary type, copy a file.
talk, rn, mail, mh, mm	Connect your terminal to another user for interactive communication. Read (and reply to) messages posted to a world-wide bulletin board. Send or read electronic mail.
crypt	Encrypt or decrypt data.
compress	Compress data or files, typical compressions are 2-3 to 1.
tar, cpio	Archive and restore files and directories into/from a single file on disk or removable media.
sed	Probably the single most useful command for rearranging or extracting pieces of data quickly. (A bit cryptic for many users, though.)

awk	More powerful than sed but somewhat slower; almost a general purpose programming language but definitely tailored to filtering text from stdin to stdout.
head, tail	First, last part of a file or stdin.
find	Locate files which meet specified criteria.
nroff, troff	Batch oriented (embedded command) text formatters. This document was edited with vi and formatted with troff.
look, spell	Look up words in an on-line dictionary. Find possible spelling errors.
sum	Compute a CRC (checksum) for comparison with supposedly identical data on a remote system.
dd	Real handy for doing low-level I/O to mag-tapes if you get one from who-knows-where in some strange format.
od	Display an octal (or hex) dump of input data. This lets you see every byte of your data as a bunch of numbers.
du, df	Display disk usage and free disk space.
script	Keep a transcript of your session in a file.
who, whoami, su	Who is on the system. What is my username? Become another user temporarily.
tip, cu	Unix's "modem program" or "terminal emulator" — it's how you login to another system via your serial port - a primitive ancestor of kermit which uses essentially no protocol.
ps, kill	Process-status lists attributes and resources associated with each process. Kill sends to a process a "*signal*" which (depending upon the signal sent) will cause the process to terminate in various ways. See also the man-page for "*signal*".

7. Other Sources of Information

1. *4.3BSD Unix Manuals*, U.C. Berkeley, published by USENIX Association, El Cerrito, CA.

2. *The Unix Programming Environment,* Kernighan, B.W. and R. Pike, Prentice Hall, Engelwood Cliffs, N.J.

3. *Welcome to Unix,* Rick Ells, Academic Computing Services, University of Washington, Seattle, Washington.

4. *Introducing the Unix System,* Henry McGilton, McGraw-Hill Software Series.

5. *The C Programming Language,* Kernighan, B.W. and Ritchie, D.M., Prentice Hall, Englewood Cliffs, N.J.

6. *Introducing Unix System V,* Morgan, R., McGilton, H., McGraw-Hill Software Series.

7. *Unix for People,* Birns, P., Brown, P., Muster, J.C.C, Prentice Hall, Englewood Cliffs, N.J.

Appendix 1: An Advanced Example

Let’s assume we are editing a file and submitting it for periodic review. Our reviewers appreciate only having to study those parts which have changed. Unix has a program, "*diff*", to find the differences between 2 text files and report them in several different formats (see the "diff" man-page), however none of those formats is what our reviewers want — the entire text of the new version with indications in the left margin where changes have been made.

As you might expect, Unix tools can be combined into a short shell-script which takes two arguments (an old and new file) and produces (on standard output) the new file with change indicators in the left margin. Don’t be discouraged if you find the solution presented here syntactically intimidating. "Sed scripts" are extremely terse and full of a powerful Unix string pattern matching notation called "*regular expressions*" (usually described in the man-page for "*ed*"). Understanding this example, does not require understanding the sed syntax.

Here is the shell-script (the line numbers and comments in *italics* are not part of the script):

```
 1  : Usage diffbar oldfile newfile
 2  TMP=/tmp/db$$                        # Set TMP to a unique tempfile name
 3  SIGS="0 1 2 13 15"                   # Termination causes to clean up after
 4  trap "rm –f $TMP" $SIGS              # Remove TMP when program terminates
 5  sed ’s/^/  /’ < $1 > $TMP            # Insert blank changebar columns in both
 6  sed ’s/^/  /’ < $2 |                 #   new and old versions
 7    diff –e $TMP – |                   # Diff the old, new versions, but alter
 8      sed ’                            #   ed commands to add change marks
 9      $a\
10       1,$p                            Append a final "ed" print command
11      /^[0–9,]*d$/ {                   Handle delete commands specially:
12          p                                keep the delete command but also
13          s/,*[0–9]*d/s;^.;–;/             modify it into an "s" command.
14          b                            Bypass remaining sed commands for this
15      }                                    line
16      s/^ /+/                          Flag new/changed text with "+"
17      ’ | ed – $TMP                    # Finally, pipe commands into ed
```

The first thing to notice about the solution is that it uses the standard Unix "diff" with the -e option. This form of the diff output is a series of edit commands which if typed to the "*ed*" editor would change the old version into the new version. These commands are of the form:

```
23,25c
New text to replace former lines 23-25.
.
7,9d
```

The solution follows from the observation that all the text which should be marked is contained in the "diff" output. The solution is to temporarily insert 2 spaces in front of every line in the old file; insert 2 character change indicators in the replacement text generated by diff and let "ed" do the replacement (re-creating the [now marked-up] new version from the old). The rest of the script is just "glue" to stick the pieces together and clean up afterwards.

The key to the simplicity of the solution is inserting spaces at the beginning of each line in the TMP versions of both the old and new files before diff’ing them (lines 5-6). This handy trick causes the lines of replacement text in the diff output to be easily distinguished from the diff-generated editor commands because they each begin with 2 spaces of changemark columns. Line 16 replaces any space in column 1 of diff output (which must therefore be replacement text) with a plus sign (+). Lines 11-15 handle deleted text by generating an additional "ed" command to put a minus sign (-) in the change mark column of the first line after the deleted text. Lines 9-10 append a final print command so that after making the changes, "ed" prints them to standard output and that’s all there is to it.

Note that a similar script has been written to diff typesetting input and insert typesetting commands which +
create changemarks in the margin after typesetting. This command is usually known as "*diffmk*" and produces +
changemarks as shown for this paragraph. +