Computer Science 340

Operating Systems

DOING THINGS WITH UNIX

CAVEAT : these notes should be sufficient to get you through the 340 assignments on Unix. They are not intended to be a complete, or even adequate, description of the system for general use.

AND FURTHERMORE : some of the details depend on the Unix implementation you use, and the system in which it is running, so using Unix in the city is not quite the same as using Unix at Tamaki. The examples were designed at the city site, so reflect circumstances there, but there will be differences in detail with different systems – or even with the same system, as things evolve. We think these notes are more or less up to date, with significant variations mentioned where it seems appropriate, but if you come across something you think it would be helpful to include please let us know. You might like to know for later reference that "Bash is ultimately intended to be a conformant implementation of the IEEE POSIX Shell and Tools specification (IEEE Working Group 1003.2)".

(The material here is part introduction and part examples of things you'll come across later in the course. Don't worry too much if some of it seems hard to absorb; unless you've learnt about it before, there is quite a bit which you might not understand until some way through the semester. You *should* understand the instruction parts; if not, ask a laboratory demonstrator.)

How Unix starts.

When a Unix system is first set into action, it starts a process using a programme called **init**, which then starts a copy of itself for each terminal attached. When anything happens at the terminal, **init** looks after the details of the logging-in operation. If the login is completed satisfactorily, **init** starts a *shell* (which is just a special sort of programme, written to accept instructions from the terminal, and to run other programmes as required by the instructions), and then dies, leaving the shell to look after the rest of the terminal session. When the session is over, the shell dies, and the original **init** waits for the next login.

The shell reads all the instructions you enter at the terminal, and executes them, commonly by starting a new process. The shell then waits for the new process to die – that is, for the programme it is running to end – then once more awaits instructions.

There are several common shells, with names like Bourne shell, C shell, and Bash. (Not "Bash shell" – "Bash" stands for "born again shell", so you needn't say "shell" again afterwards.) They are slightly different. It has been customary to use the C shell in the city and Bash at Tamaki; the programme which **init** runs to implement the C shell is call **csh**, and to implement Bash it runs **bash**. All these shells are generally similar, each exhibiting certain features which are supposed to be good. We have no particular preference which one you use, unless it turns out that the assignments are ridiculously easier in one than in the others, when we'll insist that you do it the hard way. We'll let you know if we need to.

As part of the system's logging-in process, it tries to find its knowledge about you. Obviously, the system has to store your login name and your password (or something equivalent) somewhere, and it also remembers a fair amount of other information. An

important part of that is your *home directory*, which is the way in to all the files you own in the system. When you log in, this directory is set up as your current directory; we shall see what that means later.

Logging in for the first time in the city. (*If you've already done some of this, you needn't repeat it*.)

- First find your login name and initial password. Go to a printer (preferably in the Computer Science department, but the Mathematics printers will work too if you take Mathematics papers) and pass your identity card through the reader. The screen will show your login name and show one page to print. You MUST print this page. It has on it the rules of the laboratory and your initial password. If you don't print the page then something will happen which is so nasty that no one has told me any details. You should then go to a booking terminal and change your password. Then, and only then, will you be able to use Unix.

- Book a Macintosh in the usual way.

- The communications between your Macintosh and the processors running the Unix system are handled by a programme called **telnet**. Double-click on the **telnet** icon to start **telnet**. Then select "Open connection ..." from the file menu, and enter the name of the processor you want to use where the "session name" is requested. (You can use any of those available to you, which are at the moment essentially cs26; others might be possible in principle, but cs26 is strongly preferred.) You can enter anything you like as a "window name", including nothing at all. Then click on "OK".

- After a few seconds, your screen should display a window, with some text ending with `login`. The Macintosh has transformed itself into your Unix terminal.

- Enter your login name. Then enter your password, when requested.

Logging in for the first time at Tamaki.

It isn't so easy to give general instructions, as there are several different ways to do it. You don't need to book terminals - you just find one and use it. All being well, you'll find instructions at the terminal.

Once connected –

The sequence (after connecting) looks something like this :

```
OSF/1 (cs20.cs.auckland.ac.nz) (ttyp0)

login: alan                          -- here I entered my login name, "alan".
Password:                            -- here I entered my password, which isn't
                                            displayed.
Last login: Mon Feb 27 16:45:33 from alan.cs.auckland
                                     -- should really end "alan.cs.auckland.ac.nz".
You have mail.                       -- Unix tells me there is electronic mail
                                            waiting.
%
```

(NOTE : I apologise for the squashed characters; I'm using them because I want to reproduce reasonably well what actually appeared on the screen when I ran the examples. If anyone can tell me of a narrower evenly-spaced font it would help.)

The % is the *prompt* character, meaning that Unix is awaiting instructions. (It might be something else on your system.)

When you've finished your session, use `logout` to log out from Unix, then select "Quit" from the File menu to get out of **telnet**. DO NOT OMIT THE `logout` STEP : if you do, your Unix session might continue until you do log out, presumably after your next session, consuming resources which will impair the performance of the system for other people.

Giving instructions.

Enter the instructions at the keyboard. The first "word" you enter is the instruction itself; it may be followed by one or more arguments, usually prefixed with "-". Some examples appear in the descriptions below. Some of the instructions are built into the shell itself; they are called *internal* instructions. All others – which is to say, most – are called *external* instructions, and are in fact the names of programmes. Therefore, in most cases a Unix shell executes a separate programme to obey an instruction.

(This is quite a clever trick, because it separates the operating system's terminal handler from the programmes which do the work. It is therefore (comparatively !) easy to write a new shell if you don't like the existing version. For example, you might want to write a shell which presents a graphical interface instead of the C shell's textual interface.) (*NOTE : You are advised not to start writing your own shell until after completing the 340 course. You are unlikely to have time for both at once.*)

Changing your password.

This is the first thing you should do once you have logged in for the first time, so we'll use it as an example of how to execute a Unix instruction. Most Unix instructions are the names of programmes, so if you enter "`dribble x y z`" Unix will try to find an executable file (code or command file) called `dribble`, and execute it passing `x y z` as the parameter string. (The exceptions are a few common instructions which are built into the shell and are therefor executed directly, but we won't worry about the distinction.) The instruction we want to change the password is `passwd` (NOT `PASSWD`, or any other combination of uppercase and lowercase letters; Unix worries about `case`). For me, `alan`, the dialogue goes like this :

```
% passwd                          -- here I enter the instruction.
Changing password for alan
Old password:                     -- here I enter my existing password.
New password (6 to 8 chars):      -- here I enter the password I want in future –
Again:                            -- and here I enter it again.
Password changed
There may be a delay while the UNIX password database is updated
```

Getting information.

A programme called **man** gives you access to the system manual. To find out about how to use any of the system instructions, enter man <name of instruction>. Here's how to find out about **passwd** :

man passwd -- the response below starts on a new page.

passwd(1) passwd(1)

NAME
passwd, chfn, chsh – Changes password file information

SYNOPSIS
passwd [-f | –s] [username]

chfn [username]

chsh [username]

This security-sensitive command uses the SIA (Security Integration Architecture) routine as an interface to the security mechanisms. See the matrix.conf(4) reference page for more information.

DESCRIPTION
The passwd command changes (or installs) the password associated with your username (by default) or the specified username.

The chfn command changes the finger information in the GECOS field associated with your username or the specified username. GECOS is an obsolete term, but refers to the finger information field of the passwd structure as defined in the pwd.h file and the finger information field of the /etc/passwd file as described in the passwd(4) reference page. The information in the GECOS field has been formalized by POSIX and is a comma separated list containing the user's full name, office phone, office number, and home phone number.

.... and much, much more – see below.

To get out of **man**, enter q.

man can also help if you don't know what's available which might be relevant to a topic. If you enter man –k <keyword>, (or apropos <keyword>, which is equivalent, debateably more comprehensible, and one keystroke longer) it will display a list of all entries in the on-line manual which mention the <keyword> in their short descriptions. It's up to you to find the entry you want !

```
% man -k password
acceptable_password (3) - Determines if a password meets deduction requirements
(Enhanced Security)
conflict (8)            - search for alias/password conflicts
dscrypt (3)             - encrypt a password, dispatching based on the associate
d algorithm (Enhanced Security)
dispcrypt (3)           - encrypt a password, dispatching based on the associate
d algorithm (Enhanced Security)
dxchpwd (1X)            - Create or change password program
getespwent, getespwuid, getespwnam, setprpwent, endprpwent, putespwnam (3)  - Ma
nipulate protected password database entry (Enhanced Security)
getpass (3)             - Reads a password
getprpwent, getprpwuid, getprpwnam, putprpwnam (3)  - Manipulate protected passw
ord database entry (Enhanced Security)
lock (1)                - Requests and verifies a user password
locked_out_es (3)       - determine if password-management disallows user login
(Enhanced Security)
passlen (3)             - Determines minimum password length (Enhanced Security)
passwd (4)              - Password files
passwd, chfn, chsh (1)  - Changes password file information
popwrd (8)              - Sets password for a POP subscriber
printpw  (8)            - Outputs the contents of the password database
```

..... and some more.

The numbers in brackets identify different sections of the manual; you need them to find entries which are not the first listed for the keyword. So

```
% man 4 passwd
```

gives you something which starts like this :

```
passwd(4)                                                    passwd(4)


NAME
       passwd - Password files

DESCRIPTION
       A passwd file is a file consisting of records separated by
       newline characters, one record per user, containing  seven
       colon (:) separated fields.  These fields are as follows:
```

.... and much, much more.

About “much, much more”.

- or, at least about more. You will not be surprised to know that we are not the first people to notice the inconvenience of the torrent of information which washes over the screen when you enter something like “man passwd”. Indeed, not only has someone noticed, but someone has done something about it. To see what this is, enter :

```
% man passwd | more
```

The effect of the | (which denotes what is called a *pipe*) is to take the output from whatever is to the left (in this case `passwd`), and to redirect it from the screen to whatever is on the right (in this case another programme called `more`). The output from `more` naturally goes to the screen – unless, of course, you redirect it somewhere else with another pipe. You can chain a lot of programmes together with pipes like this to form what is (inevitably) called a `pipeline`. The result obviously depends on the programmes concerned, but with the example given what appears on the screen is :

```
passwd(1)                                                    passwd(1)


NAME
       passwd, chfn, chsh – Changes password file information

SYNOPSIS
       passwd [-f | –s] [username]

       chfn [username]

       chsh [username]

       This security-sensitive command uses the SIA (Security
       Integration Architecture) routine as an interface to the
       security mechanisms. See the matrix.conf(4) reference page
       for more information.

DESCRIPTION
       The passwd command changes (or installs) the password
:
```

- and that's all (unless you have a different screen size). When the screen is full, everything stops, giving you time to read what's there. When you're ready for the next page, press the space bar. Alternatively, try these :

Press	`space`	for a new page;
Press	`f`	for a new page;
Press	`b`	for the previous page;
Press	`u`	to go back half a page;
Press	`d`	to go forward half a page;
Press	`q`	to stop.

Files.

Unix files are regarded by the system as ordered sets of bytes; no other higher level structure is defined. The simplest form of file name is a string of alphanumeric, and a few more, characters. More complicated names can be composed as a sequence of simple names separated by `/`; this name structure reflects the organisation of the files in a hierarchic directory.

There are several ways in which Unix can interpret a file name which you enter at your terminal. This isn't just to make life complicated; it's also to make it simpler. The reason for the variety is that you want to use files which live in different places. You

certainly want to use your own files, which live in your own directories, out of sight of anyone else; but you also want to use the system files, which live in the system directories, and can be seen by everybody. Whenever you give a file name, therefore, the system will first look for it in your current directory, and then in appropriate system directories. The sequence of directories inspected is called the *search path*, and is specified in a file (in your directory) called **.cshrc**, which you can inspect and change if you wish. (Just use an ordinary editor.) Only if the system can't find the file anywhere in this search path will it report failure.

Your home directory in the city is **/users/studs/ugrad/stage3/yourname**; at Tamaki, it depends which system you're using. (If you want to know, use `pwd` – see below.) That becomes your current directory when you log in. If it were not for the search path, you would have to type `/users/studs/ugrad/stage3/yourname/xyz` whenever you wished to use a file **xyz** in your directory.

Using files.

The simplest way to make a file is to use a programme called **cat**, which is short for catenate, which means chain together, which is quite inappropriate in this context. This illustrates a Unix habit, a delight to its devotees and a pain to everyone else, of putting general programmes to particular purposes which are not obvious from the programmes' names. (Try `man cat` for more details.)

To understand the instruction we shall use, we must first find out about *file redirection* – but before even that (zeroth ?), we shall indulge in some propaganda. In most of this "Using files" section, we'll be talking about *streams* rather than *files*. We'll call them files, because that's common, and it's what you'll find in Unix literature, but in the 340 course material we shall try to distinguish fairly carefully between the two structures. Here, we'll mark something we think is a stream by quotation marks. The distinction is simple. A file is composed of data standing still; it is a set of data items connected together into some sort of structure, and usually persistent – it sits there on a disc or other medium until you want it. In contrast, a stream is composed of data moving about; it is a set of data connected primarily by sequence in time, and usually not persistent – if you don't catch the items as they come, they are likely to be lost. The things we connected together by a pipe are primarily the input and output streams of the programmes we use. Bearing that in mind, then, we proceed.

Unix programmes generally have two special "files", called the *standard input* and *standard output*, which are normally identified with the keyboard and screen, respectively. If you just give the instruction `cat` by itself, you haven't specified any alternative input or output "files", so the programme copies from keyboard to screen. Furthermore, it will keep on doing so until it finds the end of its input "file" – so you will need to know that Unix sees `<ctrl-D>` as the end-of-file signal from a terminal.

We can use a file as input in place of the usual "file" by appending its name to the `cat` – so `cat xyz` will take its input from the disc file **xyz** and send it to the (assumed) output "file", the screen. That's a way to look at a small file on the screen. If we try `cat xyz uvw rst`, then we will see the three files **xyz**, **uvw**, and **rst** displayed on the terminal in sequence. There at last is the catenation.

We can also redirect the standard output to a disc file. The instruction

```
cat xyz uvw rst > abc
```

will direct the chained output to the disc file **abc**, replacing any existing file of the same name. You could write the new material to the end of an existing file **abc** with

```
cat xyz uvw rst >> abc
```

What we want, though, is to copy from the keyboard to a disc file. What happens during the session fragment below should now be clear.

```
% cat > newfile ?-- Copy from keyboard to a file called newfile.
This is the first line.
This is the second line.
This is the last line.          -- After this line, I entered <ctrl-D>.
% cat newfile                   -- Copy from a file called newfile to the
                                   screen.
This is the first line.
This is the second line.
This is the last line.
%
```

We can redirect input “files”, too. The `xyz` part of `cat xyz` above is an argument to **cat**; **cat** reads it, and itself switches from the keyboard to the named file for input. Generally, as you would expect, what programmes do with their arguments is up to the programmes. But whatever the programme does, you can instruct the system to use a different “file” for input. If you enter `cat < xyz`, the result is just the same as for `cat xyz`. Now, though, you are using this input “file” redirection technique; **cat** has no parameters, and therefore still uses the standard input, but the system has changed the source of the standard input to be the file **xyz** instead of the keyboard. To emphasise that there is a difference, consider what happens if you enter `cat < xyz uvw rst` : **uvw** and **rst** are displayed on the screen. Why ? Because first the “`< xyz`” defines **xyz** as the standard input, but then the presence of two arguments, **uvw** and **rst**, causes **cat** to replace its standard input by these two files, as described earlier.

Doing things with files.

Now we shall introduce a number of operations which apply to files themselves. These don’t do anything to the contents of the file, like those we mentioned above, but they’re useful for managing – particularly – disc files. Most of these really *are* files, not streams.

Files are kept in directories. (Strictly, they’re subdirectories; the directory is the whole list of files for the disc, but the “sub” is usually dropped and everyone seems to understand.) If you’re used to MS-DOS or Windows directories, or Macintosh folders, you will have no surprises (well, not many), so we won’t describe them in any detail.

When you log in, your home directory becomes your current directory. To inspect your list of files in the current directory, use the instruction `ls` :

```
% ls
340
c
disability
neural
tex
%
```

What happened to **.cshrc** ? It's a perfectly legitimate file name, but the `ls` instruction doesn't normally list names which begin with fullstops. You can make it do so by adding the "flag" `–a` – so, starting from my home directory :

```
% ls –a
.                       - - "." always means this directory.
..                      - - ".." means the parent of this directory.
.cshrc
.history
.login
.plan
340
c
disability
neural
tex
%
```

(Those are edited lists, obtained with some trickery to illustrate the point; there are quite a lot of files with names beginning with fullstops, conventially each giving initial parameters for some operation of the system.)

To begin with, your home directory is all you have. You can make new directories within your current directory using the `mkdir` instruction :

```
mkdir test
```

makes a new directory called **test**. To change your current directory, use the instruction `cd` :

```
cd test
```

If you want to find out what your current directory is, use `pwd` (for "print working directory" – absolutely nothing to do with passwords) :

```
% pwd
/users/staff/alan/test
%
```

The new directory appears in your directory listing :

```
% ls
340
c
disability
neural
test
tex
%
```

To distinguish between directories and files, add the "flag" `–l` :

```
% ls –l
total 2173
drwxr-xr-x   6 alan     staff        8192 Dec 22  1995 340
drwxr-xr-x   6 alan     staff        8192 Dec 27  1995 c
drwxr-xr-x   2 alan     staff        8192 Jul 16 11:23 disability
-rw-r--r--   1 alan     staff       29696 Mar 10 10:55 neural
drwxr-xr-x   4 alan     staff        8192 Dec  4  1995 test
-rw-r--r--   1 alan     staff     2118823 Apr 30 15:36 tex
```

The cryptic first column lists various properties of the item; the first letter is `d` if it's a directory, and the rest are to do with protection. They're in three groups, one for the file's owner, one for a group, and one for everyone else. "-" means no access; "r", "w"., and "x" mean access is permitted for reading, writing, and executing the file. The item for **neural** is protected thus :

	owner (u, for user)			group (g)			anyone else (o, for other)		
-	r	w	-	r	-	-	r	-	-
not a directory	owner may read	owner may write	owner may not execute	group may read	group may not write	group may not execute	other may read	other may not write	other may not execute

For a directory, the significance of "x" is different : it means that people in the appropriate category can search the directory. The other items retain the same meanings as for files.

To change the protection parts, use **chmod**. Here are some examples :

`chmod u-w neural`	Remove (–) permission for the owner (u) to write (w) to the file neural.
`chmod a+x tex`	Give (+) permission for everyone (u + g + o) (a) to execute (x) to the file tex.
`chmod o-x 340`	Remove (–) permission for others (o) to search (x) the directory 340.

There are many simple file operations which you can use to organise your files in sensible ways. Here are a few examples. You can delete a file with **rm** (remove) :

```
rm tex
```

but to delete a directory you use **rmdir** :

```
rmdir test
```

Use **mv** (move) to change the name of a file :

```
mv neural anothername
```

It's "move" because it will move the name from one directory to another if you give it that sort of information :

```
mv neural test
```

moves the file neural to the directory test, while

```
mv neural test/anothername
```

both moves the name and changes it.

Shell scripts – which is to say, command files.

A command file is a series of operating system instructions stored in a file, which amounts to a programme to be executed by the operating system. A Unix command file is called a *shell script*, doubtless, though not necessarily, for some reason. As you would expect, it is executed by the shell; as the shell is an ordinary (more or less) programme, driving it from a file instead of from the keyboard is exactly analogous to redirecting a programme's input "file" from keyboard to disc file.

Here's a rather trivial example. Suppose that you had some conscientious objection to using Unix pipes. (Or perhaps your "|" key doesn't work.) Here's how you could make a shell script to do the same job as the example we gave for looking at the **man** page for **passwd** :

```
% cat > nopipes                -- Make a file called nopipes.
man passwd > notapipe
more < notapipe
rm notapipe                    -- followed by <ctrl-D>.
% chmod u+x nopipes            -- Make the file executable.
% nopipes                      -- Execute it.



passwd(1)                                                  passwd(1)


NAME
       passwd, chfn, chsh – Changes password file information

SYNOPSIS
```

..... etc.

Editing.

Using **cat** is a convenient way to make or inspect a little file, but for a much larger one we need an editor. The standard Unix editor is **vi**. It isn't very exciting, but it works. Other editors are available – it doesn't matter which you use, but **vi** is the standard.

Here are some elementary instructions which will probably suffice for the 340 exercises. There is a great deal more to **vi**, some of which you will need if you want to use it on bigger files; refer to an appropriate textbook for more information.

To edit the file **newfile** :

```
vi newfile
```

The contents of the file are displayed on the screen. With an extraordinarily boring **newfile**, you could therefore end up with a screen looking like this :

```
This is the first line.
This is the second line.
This is the last line.
```

with a cursor positioned at the first character. You can move the cursor with the arrow keys on the keyboard. Move the cursor to the first `t`; then enter "`iabsolutely `". (*Don't* enter the quotation marks – they're to identify the ends of the string.) The `i` means "insert characters", and the ` means "stop inserting characters". Between the `i` and the ` the editor is in *insertion mode*; every character you enter (except ` ) is written into the file. ( If you want to enter a character ` from the Macintosh keyboard, press and ` together.) You have to leave insertion mode before you can give any further instructions to the editor. The result is :

```
This is absolutely the first line.
This is the second line.
This is the last line.
```

Move the cursor to the `l` of `last`, and enter `xxxxx` :

```
This is absolutely the first line.
This is the second line.
This is the line.
```

Move the cursor to anywhere on the second line, and enter "`oThe third line.`". The `o` means "Make space for a new line after this one, and start inserting characters into it". (If you use `O` instead of `o`, the new line is inserted before the current line.) The result is :

```
This is absolutely the first line.
This is the second line.
The third line.
This is the line.
```

Move the cursor back to the second line, and enter `dd`. This deletes the current line; the result is :

```
This is absolutely the first line.
The third line.
This is the line.
```

You can save the file at any time – provided that the editor is not in insertion mode – by entering :w (for write). When you enter the colon, the cursor jumps to the bottom of the screen. Do not be dismayed; **vi** remembers where it was, and puts it back for you when the operation is finished. Other operations of this sort include :q to leave **vi** (or :q! if the file has been changed and you don't want to file the changes), and :x to save the file and leave **vi**.

Printing files.

To print a file called **filename**, use :

```
lpr filename
```

(That works at the city campus, where all the printers work as a single system. At Tamaki, you have to identify the printer – so "lpr –p <printername>" instead of just lpr, with similar specifications for the other printer instructions below. The printer names are on the printers – look at the closest one, and use that, unless you want to send your files somewhere inaccessible.)

The file will be queued for printing. When you are ready to collect the printed copy, go to the printer room and pass your student identity card through one of the barcode readers. A list of your queued files should then appear on the associated computer screen, and you can select those you wish to have printed. Any you don't select will be deleted. If there's a long queue, you might prefer to wait; you can inspect the queue with the instruction

```
lpq
```

That will give you a list of jobs waiting to be printed, each identified by a number. If you want to delete one of your jobs from the queue, use

```
lprm <number>
```

You can't delete anyone else's jobs.

FURTHER INFORMATION.

There are some ancient standard books; the classic is

S.R. Bourne : *The Unix system* (Addison-Wesley, 1983).

Another, much more explanatory, and with more examples, is

M.G. Sobell : *A practical guide to the UNIX system* (Benjamin Cumings, 1984).

It's quite likely that there are newer ones, but Unix, almost alone among practical operating systems, doesn't change; it grows, but the basics are rock-steady. (It doesn't necessarily always look the same, because the shell you use can make a difference, but that's a matter of learning the shell you have.) For me, that's its biggest advantage.

There is a fair amount of Unix information on the WWW server; a good place to start is http://www.cs.auckland.ac.nz/tech-support/software/unix/, which includes links to several interesting sources.

Alan Creak,
July, 1998.