H  More About UNIX

The previous section of this manual served as a brief introduction to UNIX, providing enough to get you started in 433-171/151. This section contains some slightly more advanced topics in UNIX which you need in first year, some in 433-171/151, and others in 433-172/152. Refer to the Schedule of UNIX commands for details of which commands you should know at each stage of your course.

Of course, there is a lot more to UNIX than what is contained in this manual. If you continue studying Computer Science in later years, you will learn more there, but you may also wish to find a good reference book on UNIX to learn on your own. (There are a couple of references given at the start of the previous section, but there are many more books available. Most large book stores will have a number of such books, and it is worth taking time to browse and find one that you prefer.)

1  Piping and Redirection

In previous sections, you have used the piping (|) and redirection (>) operators. (For example: man -k copy | more; cal 3 2003 > Mar2003.)

What exactly do these operators do?

- The redirection > operator takes the output of a command and writes it to a file.
- There is another redirection operator (<) which directs the contents of a file to be the input to a command.
- Another related operator is the concatenation (>>) operator, which takes the output of a command and adds it to the end of a file (whereas the redirection operator will overwrite the old contents of a file).
- The piping operator takes the output of a command, and uses it as the input for another command. For example, the using the command w | more is the same as using the following series of commands:

  w > tmp
  more tmp
  rm tmp

You may find these operators very useful when you are testing some of your C programs.

2  UNIX command options

In a previous section, we looked at the command rm -i *. This is an example of a command used with an option. Another example of a command we have seen used with an option is man -k.

Most commands have options, indeed most have several options. The options available for any command are described in the man page for that command. Let's look at a few of the options for ls. Enter the commands:

```bash
ls -al
ls -F
ls -h
ls -v
ls -1
ls -d
ls -R
ls -a
ls -A
ls -t
ls -T
ls -m
ls -s
ls -Z
ls -M
ls -R
ls -l
ls -d
ls -r
ls -T
ls -m
ls -s
ls -Z
ls -M
```
ls -l
ls -F
ls -s
ls -Fis

(Use the alphabetic character l, not the number 1 in these commands.)

See if you can work out what each of the options does. Check the manual page to confirm your guesses.

Check the man pages of other commands you commonly use (e.g. cp) to see what options they can take.

3 Keeping a backup copy

When you are doing important work (e.g. projects), you should always keep a backup copy of your work. Some of your projects will have several stages, the idea being you complete one stage and then go on to the next. After each stage, you should make a backup copy. This way, if you run into difficulties in the next stage, you can go back to your previous work.

Let’s say you are working on your first project, and you’ve just finished stage one. Change into the directory that contains the work for this project. Say you’ve called the file with the first stage your project projA-1.hs (or projA-1.c). First of all, we need to create a directory where we’ll store your backup copies (if you haven’t already done so). Enter the command:

    mkdir ../saved

This will create a subdirectory (in the directory above this one) called saved. You can use this directory for all your backups, not just the ones for this project. Now enter the commands:

    cp projA-1.hs ../saved
    cp projA-1.hs projA-2.hs

(or

    cp projA-1.c ../saved
    cp projA-1.c projA-2.c

) This means that if something goes wrong, you’ll have a backup of all the work up to stage 1. (You should do a similar backup after each stage.)

4 Functions; .bashrc and .bash_profile files

UNIX allows you to give simple names to complex commands to make typing faster. This is done using the function command. Enter the following:

    function ll() { ls -l; }

(If it gives an error, or when you pressed return you got a line starting with ‘>’, you probably didn’t include the whitespace — this is one of the few times that the whitespace (after the
open brace and before the close brace) is important.) Now enter the command `ll` and see how it behaves like the old command `ls -l`.

Now enter the command:

`ls -a`

This time you should see some further files in your current directory. The names of these files begin with the `.` character, and may include two files named `.bashrc` and `.bash_profile`. These two files are used to customise the behaviour of your account.

If you do not have these files (their names will not appear in the listing if you don’t) then you will need to copy the default system ones into your home directory in order to be able to customise your account to your liking. The system files contain reasonable defaults, so you could get by without your own copies, but as time goes by you will find that customisation makes working with UNIX much more agreeable. Ask your demonstrator or classmates for the locations of the system files if necessary.

The `.bash_profile` file is important because the commands it contains are executed every time you log on. Usually, there is a line at the bottom of your `.bash_profile` which looks like this:

```bash
source ~/.bashrc
```

This means that every time you log on, the commands in your `.bashrc` will also be executed.

Take a look at your `.bash_profile` and `.bashrc` files by using `more` or `less`. Any functions that have already been set up for you will probably be in your `.bashrc`.

Below are some functions that other users have found useful. Examine these, determine their effects (by testing them and by using the `man` command), and decide whether or not you would like to include these, or others, in your `.bashrc` file.

```bash
function cp() { /bin/cp -i; }
function mv() { /bin/mv -i; }
function rm() { /bin/rm -i; }
function l() { less; }
function ls() { ls -aF; }
function ll() { ls -alsF; }
function m() { more; }
function moer() { more; }
```

If you do modify your `.bashrc` file and then want to see the effects of your modifications, first enter the command

```bash
source .bashrc
```

in order to activate your new `.bashrc` file. Note that functions that you create at your UNIX prompt will only exist for your current session, whereas functions that you put in your `.bashrc` file will be there every time you log in.

5 History list and recalling commands

Enter the command:

`history`
to display a listing of the most recent UNIX commands that you have made. You may wish to pipe the output from history into more if it fills more than one screen (i.e. `history | more`).

Whilst this list may be of interest in its own right, its greater value and interest for you will lie in the use you can make of this to re-execute a previously-executed command. You will need to experiment using your own history of commands, to see exactly what occurs in various instances. The following example illustrates some usages.

Suppose that you have the following history list:

60 pwd
61 cd
62 vi .bashrc
63 source .bashrc

The current event is therefore number 64.

Suppose you now want to execute a command that is the same as event number 62 (that is, make another change to your .bashrc). You could enter any one of the following:

```
!62 !-2 !vi
```

The first of these re-executes the numbered command, No. 62; the second re-executes the command that is 2 before the current number; and the third re-executes the most recent command that commences with vi.

To execute a command similar to number 62 but on .bash_profile instead of .bashrc, we can use the `substitute` command:

```
!62:s/bashrc/bash_profile/
```

This would then execute the command: `vi .bash_profile`.

But, you must experiment using your own history of commands, and learn how this command re-call provision can assist you greatly, especially if you use long command calls.

6 Finding out who else is on-line

There are times when you will find it useful to know who else is logged on, and how much the machines are being used. There are a number of UNIX commands which allow you to do just that:

```
who (or who | more)
w (or w | more)
finger (or finger | more)
```

Try them for yourself, you might want to use them later. Also, the last two commands can take a username as an optional argument. For example, if your friend’s username was “fred”, you could find out if he was logged in to your machine by using the command `w fred` or `finger fred`. Try this out. Notice that the `finger` command gives very different output depending on how you use it.
7 Combining commands into one command line

Do you want to create a new subdirectory into which you will place backup copies of your Haskell program script files?

The following single command line uses the semi-colon (;) to separate the line into three separate commands that are executed sequentially.

```bash
mkdir saved; cp *.hs saved; ls saved
```

Make sure you understand the three commands that this command line executes, then try a similar multi-command line of your own choosing.

8 Summary

To recapitulate the major points of interest in this section:

- To modify the behaviour of a command, use an appropriate command option. Command options generally follow the name of the command and consist of a `-` and a character.

- Functions are a way of conveniently associating a number of commands with a name which you supply. Put the functions you wish to keep into your .bashrc file.

- To redirect the output of a command into a file, use the `>` operator.

- To redirect the output of a command into the input of another command, use the `|` operator.

- To view your command history, use the command `history`.

- To run a command in the command history, use the `!` operator.

- To combine commands into one line, separate them by the `;` operator.

- To find out who else is logged on, use the following commands: `who`, `w`, `finger`.