Introduction to Unix

DISCLAIMER: This document leaves out *lots* of stuff. If you want to learn Unix properly, read the O'Reilly handbook. There is also a web-based tutorial[1] available (or find your own by doing a `google` search for "unix tutorial").

The Unix Philosophy

Unix consists of a multitude of tiny commands that each generally accomplish only one task, but accomplish it well. This is a deliberate choice to make the operating system as lightweight as possible. The commands can be combined using stream control (see below) in any number of ways to perform more sophisticated compound tasks. An experienced user has tremendous freedom for manipulating data under Unix, something that is sorely lacking in more "user-friendly" operating systems like Windows. With a bit of practice you'll wonder how you could ever live without it.

The Man Pages

Every Unix command has an associated "man page," i.e. a terse description of the command usage and function. For example, `man man` will tell you more about how to read man pages. Unfortunately, if you don't know the name of the command you're looking for, this doesn't help you very much. In that case, try `man -k` ***keyword***, which will search for man pages with ***keyword*** in the description. This often leads to a bewildering screenful of possible commands. Generally you need only concern yourself with man pages from section 1 (Unix commands), 2 (library calls), and 3 (C and FORTRAN functions), which are indicated by the corresponding number in parantheses in a `man -k` listing (sometimes with a small letter appended, e.g. 3c for C functions). WARNING: the `man` command often behaves differently on different operating systems. For example, under `SunOS`, by default only the first matching man page is shown (use `-a` to show them all) and sections are specified using `-s`.

System Information

Some flavours of Unix have commands or files that show useful system information. Under SunOS/Solaris, try `sysinfo`, under SGI IRIX try `hinv`, and under Linux look at the contents of `/proc/cpuinfo`. Information available usually includes total memory, processor speed, cache size, attached devices, etc.

Quick Summary of Important Unix Commands

In the table on the next page a variety of the most important Unix commands are listed with their common usage and a brief description. This is intended only as a quick reference; check the man pages for more details. Entries in *italics* are built-in shell commands (`csh`) and entries underlined are used most frequently in scripts. Square brackets [ ] indicate optional arguments. An ellipsis ... indicates multiple arguments of the given type are supported. Note that the syntax for arguments varies between commands (e.g. whether you can use `-al` in place of `-a -l`); unfortunately only the man page can tell you for sure.

Using I/O Streams

When we say a command "prints" a value we typically mean that it displays the value on your terminal screen. What's really happening is that the result of the command is being sent to "standard output" (stdout), which by default is your screen. Similarly, "standard input" (stdin) is by default your keyboard. Data moving from the standard input to the standard output is an example of an I/O stream. It is possible to divert such streams to a certain extent. The following table summarizes the most common stream controls used in the `csh` environment with some examples:

[1] http://www.ee.surrey.ac.uk/Teaching/Unix

Command with Common Usage	Description
alias [name [string]]	Make name an alias for string
awk script [file]	See description below
basename [path]	Return everything after last / in path
bg [%job-id]	Put job in the background
break	Escape from control loop
cat [file ...]	Concatenate and display files
cd [dir]	Change working directory
chmod mode file	Change permissions mode of file
continue	Continue next iteration of control loop
cp source target	Copy source file to target file
diff file1 file2	Display differences between file1 and file2
dirname [path]	Return everything before last / in path
echo [-n] string	Write string to standard output
exit [value]	Exit from current shell with optional return value
fg [%job-id]	Return job to foreground
foreach word (list) [...] end	Succesively set word to elements in list
grep [-i] pattern [file]	Search for pattern in file
head [-number] [file]	Print first few lines of file
hostname	Print name of current host
if (expr) [then ... else ... endif]	Conditional evaluation (else if also supported)
kill [-signal] pid	Terminate or signal a process
ln -s source [target]	Make symbolic link from target to source
ls [-alrt]	List contents of directory
make	Maintain and regenerate related programs and files
man [-k] command	Display man page for a command or keyword
mkdir dir	Make directory
more [file ...]	Browse or page through a text file
mv source target	Move or rename source to target
nice [-priority] command	Execute command with specific priority
ps	Print information about active processes
pwd	Return working directory name
rehash	Refresh command search path
renice priority pid	Change process priority
rm file	Remove file
rmdir dir	Remove directory
sed script [file]	See description below
set [variable [= value]]	Set shell variable to value (opposite unset)
setenv [VARIABLE [word]]	Set environment variable to word (opposite unsetenv)
sort [file]	Sort lines of text file
source file	Execute commands from text file
switch (expr) case: [...] endsw	Choose from among a list of actions (like in C)
tail [-number] [file]	Print last few lines of file
tee [file]	Replicate standard output
time command	Time a command
top	Display information about top CPU processes
touch file	Create empty file or update access & mod times
uptime	Show system load and how long system has been up
w	Display information about logged-in users
wc [file ...]	Count lines, characters, and bytes in file
whatis command	Display summary about a command
which command (also *where command*)	Locate a command; display its pathname or alias
while (expr) [...] end	Loop while expr true

Operator	Function	Example	Description
`<`	redirect stdin	`mail dcr < myfile`	mail `dcr` the contents of `myfile`
`>`	redirect stdout	`myprog > log`	write output from `myprog` to `log`
`>>`	append stdout	`myprog >> log`	append output from `myprog` to `log`
`<< word`	redirect stdin until `word`	`myprog << STOP`	send next lines to `myprog` until `STOP`
`|`	pipe stdout to stdin	`echo test | lpr`	send phrase "test" to printer

There are also modifiers like `>&` (to redirect both stdout and "standard error" (stderr), the diagnostic stream) and `>!` (to "clobber" an existing output file without confirmation).

awk

`awk` is a sophisticated file parser that is capable of manipulating data in columns and performing some high-level math operations. A reasonably general representation of a typical `awk` invocation is as follows:

```
awk 'BEGIN {commands} /pattern/{commands} END {commands}' file
```

The `BEGIN` and `END` steps are optional. They are used to issue commands before and after the parsing step, usually to first initialize counter variables and then print the result at the end. The `pattern` is compared against every line in the file and when a match occurs the associated commands are executed (see `sed` below for more info on pattern matching). Omitting `pattern` is equivalent to matching all lines. The commands can refer to columns in the data file, e.g. `print $3` simply prints the third column (columns are delineated by "whitespace"—any combination of tabs and spaces, up to the end of the line). You'll notice that the commands have a very C-like look and feel. Even the `printf()` function is supported, which makes `awk` a powerful way of cleaning up poorly formatted data files.

Here are a few examples:

1. To count the number of lines in a file (this is equivalent to the Unix command `wc` but it's illustrative of the power of `awk`):

   ```
   awk 'BEGIN {nl=0} {nl++} END {print nl}' file
   ```

   Variables are actually automatically initialized to zero, so the `nl=0` step is unnecessary, but it's good practice. Also, `awk` has a special variable `NR` which contains the current line (record) number in the file, so `awk 'END {print NR}' file` would accomplish the same thing as this example.

2. To compute the average of the values in the 4th column of `file`:

   ```
   awk 'BEGIN {sum=0} {sum += $4} END {printf("%.2f\n",sum/NR}' file
   ```

   Here we used `printf()` to restrict the output to two places after the decimal. Note the `\n` to force a new line, just like in C.

3. To print fields in reverse order between all "start" and "stop" pairs (why not?):

   ```
   awk '/start/,/stop/{for (i=NF;i>0;i--) printf("%s ",$i); printf("\n")}' file
   ```

One handy argument to awk is `-Fc`, where `c` is any character. This character is used as the delimiter instead of whitespace, which can be useful for, say, extracting the month, day, and year digits from an entry like "MM/DD/YY".

There's lots more you can do with `awk`. In fact, there's an entire O'Reilly book on both `awk` and `sed` if you're interested...

sed

`sed` is a useful editor that can alter text that matches a particular pattern in a file or I/O stream. The basic syntax is `sed script file`, where the script contains a pattern to match and replace using "regular expression" rules (`man regexp`). The usage is best illustrated with examples. For more information, check the man pages or that `awk` and `sed` book by O'Reilly.

1. Replace all occurences of "foo" with "bar" in a file:

   ```
   sed s/foo/bar/g file
   ```

   The `g` instructs `sed` to replace *all* occurences on a given line, rather than the first as is the default. Note that this invocation directs the output to the screen; use redirection to put it in a new file. Also, any character (except `\`) can be used in place of `/` to delimit the target and replacement, so long as the same character is used throughout. This can be useful when dealing with pathname substitutions.

2. Replace all occurences of "foo" with "bar", but only in lines containing the keyword "doh":

   ```
   sed /doh/s/foo/bar/g file
   ```

3. Same again, but this time replace the *4th* occurence of "foo" and only on lines *starting* with "doh":

   ```
   sed /^doh/s/foo/bar/4 file
   ```

4. Same again, but now "doh" must be at the end of the line:

   ```
   sed '/doh$/s/foo/bar/4' file
   ```

   Note the `''` to hide the special meaning of `$` to the shell.

5. See the shell scripting section below for another example, namely using `sed` to rename files.

Shell Scripting

Shell scripting is a lot about trial and error. The most frustrating aspect can be sorting out the correct syntax for double quotes, single quotes, and reverse single quotes! The following examples highlight some of the most common features used in typical `csh` scripts:

```
#!/bin/csh
# rename -- simple script to rename .txt files to .dat
#
if ($#argv < 1) then
    echo "Usage: $0 txt_file [ txt_file ... ]"
    exit 1
endif
#
foreach txt_file ($argv)
    echo $txt_file
    set dat_file = `echo $txt_file | sed s/.txt/.dat/`
    mv -f $txt_file $dat_file
end
```

The first line indicates to the shell that we want to use `csh` (you can actually specify *any* command that accepts scripts here, e.g. `/local/bin/bash`, `/local/bin/perl`, etc.). After the first line anything following a `#` is ignored as a comment. We next check to see that at least one argument was passed (the `$#var` construct returns the number of elements in a multi-valued shell variable `var`). The double quotes `""` are necessary in the echo statement to avoid misinterpretation by the shell of the square brackets [ ] *while at the same time* allowing the shell to expand `$0`, which stands for the name of the script itself (confusing, isn't it?). Crudely, double quotes allow for *some* interpretation by the shell of special characters on an input line, single quotes allow for none, while no quotes allow full interpretation.

A return code of 1 is used to indicate an error and can be tested by immediately checking the value of the special shell variable `status` after running the script.

The main procedure is to loop over all the arguments and use `sed` together with `mv` to change the extensions of the files. The intermediate variable `dat_file` is unnecessary but helps to make the procedure clearer. Note the use of the reverse single quotes which causes the result of the enclosed command, in this case a pipe of the name of the `.txt` file to `sed`, to be returned as a value.

To use this script like a normal command, you need to `chmod` it to executable status, e.g. `chmod a+x rename`. Then type `rename` to run it (or `./rename` if '`.`' is not in your path).

Here's another example:

```
#!/bin/csh
# examine -- simple script to graphically show file size
#
if ($#argv < 1) then
    echo "Usage: $0 file [ file ... ]"
    exit 1
endif
#
foreach file ($argv)
    echo -n $file': '
    @ nl = `wc $file | awk '{print $1}'`
    while ($nl > 0)
        echo -n '*'
        @ nl--
    end
    echo ''
end
```

The main loop here uses `wc` together with `awk` (which we use to extract the first column of the output from `wc`) to get the number of lines of each file (recall we could write `` @ nl = `awk 'END {print NR}' $file` `` instead). This value is then used in a decremental loop to print one asterix for each line in the file. The `-n` argument to `echo` suppresses the linefeed; the `echo ''` command forces one. Note the use of `@` for shorthand integer variable manipulation and arithmetic (`set nl = $nl - 1` would also work but it's not as clean).

Hint: You can get the shell to echo each line of your `csh` script before it's executed by adding a `-x` argument at the end of the shell declaration line (e.g. `#!/bin/csh -x`). This is *very* useful for debugging...

As a final example here's a script to run a code three times with three different inputs that are specified at the beginning of the script (this is done to make the values easy to change later on; alternatively these values could be passed to the scripts via `argv`):

```
#!/bin/csh
set run_values = (10 100 1000) # specify run values here
if ($#argv != 0) then
    echo $0 takes no arguments
    exit 1
endif
#
foreach value ($run_values)
    echo Starting run with value = $value
    ./myprog $value >&! log$value
    if ($status) echo WARNING: Problem during execution
end
```

Notice that a shell variable can be set to an array using the ( ) notation. To access elements of an array, use [ ]: `$run_value[2]` has a value of 100 in this example. WARNING: array indices start at 1, not 0, in `csh`. Why this is different from C (which starts at 0) is one of those unanswered mysteries of life.

The call to `myprog` simply passes the index value `$value` as an argument. It's up to `myprog` to use the `argc` and `argv` formalism in `main()` to read the passed value (assuming `myprog` is written in C/C++ of course). Output (including stderr) is redirected forcefully to a log file that will be different for each run. The `status` variable is checked to see if there was a problem during the run (an example of how religiously returning meaningful values from your codes can be useful).

A Word on HTML

It's a great idea to learn how to write your own web page. In a competitive job market you should be striving to advertize yourself effectively to potential employers. These days a good web page can make all the difference. Most web browsers have built-in editing facilities, but if you want to learn from scratch, check out some online tutorials:

`http://www.htmlgoodies.com/`

(or do a web search). For those new to the Astronomy department, see the document:

`http://www.astro.umd.edu/computers/homepage.html`

for information on how to link your content to the department web pages.

By the way, this tutorial (along with most of the other handouts for this class) was converted automatically from LaTeX to HTML using `latex2html`. See the man page or:

`http://www.latex-project.org/`

for more details. For convenience, the HTML versions contain hyperlinks for URLs like this.