Unix In Depth

Control Characters

```
ctl-m           //return
ctl-d           //logout - same as typing exit
ctl-g           //rings bell on the terminal
ctl-h           //backspace
ctl-u           //delete whole line
ctl-s           //pauses output to screen
ctl-q           //un-pauses output to screen
```

Unix Mail

```
mail            //to read your mail
Return          //advances to the next email
d               //deletes the email
p               //reprint the email
s <filename>    //saves it in a file that you name
q               //quit mail
mail <user>     //opens a new email to <user>
ctl-d           //sends the email and closes the mail program
```

LS Options

```
ls -t           //sort by time
ls -u           //gives info on when files were used
ls -r           //reverses order of any other option used (e.g
. -t)
ls -d           //check just the directory that you're in
ls -c           //
ls -i           //reports the i-number of each file (decimal n
otation)
```

CMP

```
cmp             //compares 2 files byte by byte - diff better
```

`cmp` works on any type of file, although `diff` only works on text files.

Directories

```
pwd             //print working directory
echo *          //echoes all the non-hidden files in the direc
tory
cat *           //prints all the files in the dir
rm *            //deletes all files in the current directory
```

Files

```
file            //determines what type a file is
```

A runnable program is maked by a binary 'magic number' at it's beginning. Use `od` with no options to find it. The octal value `410` marks a purely executable program. `410` is not ASCII text, so an editor cannot create it.

In Unix there is only one type of file, and all that is required to access it is its name.

Misc

```
&               //if you end a command with &, it will start t
he                    //program but accept further prompts
od              //octal dump. Shows the bytes of a file. Use w
ith -cx
```

Programs retrieve the data in a file by a system call (a subroutine in the kernel) called `read`. Each time `read` is called, it retrieves the next part of a file. E.g. the next line of text typed on the terminal.

```
rm -f           //forces removal without interactive request
```

Parentheses can be used to group commands. Here the output of `date` and `who` are concatenated into a single stream that can be sent down a pipe.

```
(date; who) | wc
```

You can grab the interum output from a command that is being piped and store it in a file with the `tee` command. E.g.

```
(date; who) | tee save | wc
```

WC receives the data as if `tee` weren't int he pipeline.

Processes

An instance of a running program is called a process. Processes are not the same as programs.

Every time you run the **program** `wc`, it creates a new **process**.

If several instances of the same program are running at the same time, each is a separate process with a difference process ID.

```
kill 0
```

This will kill all your processes besides the login shell

```
nohup <command> &
```

The command will continue to run if you log out and will save any output into the file `nohup.out`

```
nice <resource heavy command> &
```

If you have a command that uses up a lot of processor resources, you can run it with lower priority, so that other users don't suffer.

```
at time
<commands>
ctl-d
```

This will run a command at whatever time you like.

SHELL Variables

```
$PS1            //Terminal Prompt
$PATH           //Search Path
$TERM           //Name of Terminal you're using
```

To tell other programs that you want to use a personal variable you've set in `.bash_profile` or just in the terminal, use `export`. e.g.

```
export d="/dev/"
```

Permissions

When you login, you are assigned a `uid` by the system. 2 different login names

can have the same `uid`, making them indistinguishable to the system, although this is not good for security reasons.

Every new user is assigned to the group of `Other`, although this varies by system.

In `/etc/passwd` you'll find all the passwords for all users of the system. While the file is ordinary text, the field definiteions and separators are agreen upon conventions used by the programs that use the file.

```
login-in:encrypted-password:uid:group-id:mescellany:login-dire
coty:shell
```

So for my Unix configuration, it's

```
root:x:0:0:root:/root:/bin/zsh
```

If the shell field is empty, it implies that you use the default shell. The miscellany filed may comtain anything (phone number/postal address).

When you give you password to `login`, it encrypts it and compares the result against the encrypted password in `/etc/passwd`. If they agree, it logs you in.

The file `/etc/group` encodes group names and group id's, and defines which users are in which groups. `/etc/passwd` only identifies users in your login group.

```
newgrp              //changes your group permissions to anothe
r group                       //and logs you into that group
```

To change your password, use the `passwd` command.

If you use `which passwd` to find it's path, then use `ls -lah /usr/bin | grep passwd` (or whatever it's path is), you'll see `passwd`'s permissions.

Note that instead of `-rwxr-x-r-x`, it has `-rwsr-xr-x`. The `s` in the

execute field means that when the command is run, it is to be given the permissions corresponding to the file owner (i.e. root). This means that any user can run the `passwd` command to edit the password file.

What **executable** means: when you type something like `who` to the shell, it looks in a set of directories name `who`. If it finds the file, and has the execute permission, the shell calls the kernel to run it. The kernel checks the permissions, and if valid, runs the programm.

NOTE: A program is just a file with excecute permissions.

If you have write permission to a directory, you can delete files in that directory, even if you don't have write permissions to those files.

If you `chmod` a file or directory, it won't update its modification date. That only happens when you modify the contents of a file/dir.

Inodes

Administrative information, such as permissions, modification dates, disc location, and file size are not stored in the file itself, but in a system structure called an index node, or **Inode**.

There are 3 times in the Inode - last modified, last used (read or executed), last change of Inode itself.

The system's internal name for a file is its **i-number** - the number of the Inode holding the file's information.

The i-number is stored in the first 2 bytes of a directory, before the name. `od -d` will show you this. These 2 bytes are the only connection between the filename and its contents. Therefore a filename in a dir is actually a *link*, because it links the name in the directory hirarchy to the Inode, and hence the data.

The same i-number can appear in more than 1 directory. The `rm` command removes links, and when the last link to a file disappears, the system removes the Inode itself, and hence the file.

The number printed between permissions and owner with the `ls -lah` command is the number of links to the file. There is no difference between the first link and subsequent ones.

Devices

Instead of system routines to control devices, there are files in `/dev` that contains device information that the kernel references before issueing hardware commands.

If you do `ls -l /dev`, the first char of the permissions will be either `b` or `c`. For a device file, the inode contains the internal name for the device, which consists of its type - character `c` or block `b`, and a pair of numbers calle dthe major and minor device numbers.

The major number encodes the type of device, while the minor number distinguishes different instances of the device.

Disks are block devices, and everything else is a character device. On Mac Unix, nearly everything is a `c`. Only 4 or so `b`'s

`mesg n` Will turn off messages `mesg y` Will turn them back on again

To time a command without your screen getting filled up with junk output, you can use `/dev/null`. E.g.

```
time ls -R / > /dev/null
```

You get

```
real    0m21.931s
user    0m2.174s
sys 0m3.378s
```

In order, these times are elapsed clock time, CPU time spent in the program, and CPU time spent in the kernel while the program was running.

The Shell

Metacharacters

Characters like `*` that have special properties are known as metacharacters. There are a lot of them. To stop a character from being interpreted as a metacharacter, enclose it in single quotes `'`.

Double quotes don't work as well, because the Shell still looks inside for a `$` or a `\`.

Another way is to escape every instance of the metacharacter with a slash `\`.

A `\` at the end of the line tells the shell to ignore the line break.

Creating New Commands

One way to create a new command is to create a file that contains the set of commands you want to execute. E.g.

```
echo 'who | wc -l' > nu
```

Then you can call it from the shell in 2 ways:

```
sh < nu
sh nu
```

If a file is executable and contains text, then the shell assumes that it is a file of shell commands. That said, you still have to place it in one of the directories in $PATH, or add the current directory to $PATH.

```
chmod +x nu
```

Now you can run the command just by typing `nu`.

Command Arguments and Parameters

When the shell executes a file of commands, each occurance of $1 is replaced by the first argument, each $2 is replaced by the second argument, and so on until $9.

Eg. if you make

```
chmod +x $1
```

with the command

```
cx nu
```

it will take `nu` as the first argument.

To make a command take an unlimited number of arguments, use `$*`. E.g.

```
chmod +x $*
```

Although if an argument with more than 1 word is supplied (e.g. "elis island"), it will throw an error, even if the argument is in quotes, because bash will interpret the space as a delimiter between 2 difference arguments, so

```
echo 'grep $* phone.txt' > grop
echo 'elis island' > phone.txt
chmod +x grop
grop "elis island"
```

Will throw an error. What you need to do is encase the `$*` in double quotes, since bash will look inside it for instances of `$`, `\`, and `…`

```
echo 'grep "$*" phone.txt' > grop
```

Now arguments with a space will work.

The argument `$0` is the name of the program being executed. So in the example above, it would be `grop`.

Commands in the Sub-Shell

Commands are carried out in a sub-shell. This means that without modification, they cannot set shell variables, because variables are associated with the shell they are created in, and are not inherited by child shells. Unix provides a dot operator `.` that executes commands in the current shell, rather than a sub-shell. Unfortunately, it can't be used in files full of commands - tested on mac.

You can't pass arguments to a command prefixed by `.` so you can't use `$1` `$2`, etc

If you set shell variables in a sub-shell, they are only available in the sub-shell, until you declare them using the `.` operator.

When you want to make the variable available in sub-shells, you should use the `export` command.

Although, if you export a shell variable in a sub-shell, it will not be avaiable in the parent shell.

Echoing Commands

Using batsticks (`), you can echo out commands:

```
echo hello `date`
```

Redirecting the Standard Error Output

Every program has 3 default files that are created when it starts, numbered by small integers called file descriptors.

The standard input is `0`. The standard output is `1` which is often redirected from and piped into. The standard error output is `2`.

Sometimes programs produce output on the standard error even when they work properly, e.g. `time wc`.

```
time wc desktop 2>tmr.txt
```

This will store the standard error output in `tmr.txt`. It also works for any error message, e.g.

```
fdsaf 2>error.txt
```

No spaces are allowed between `2` and `>` and `filename`

The notation `2>&1` tells the shell to put the standard error on the same stream as the standard output. The notation `1>&2` tells the shell to put the standard output on the same stream as the standard error.

It can help stop output disappearing into a pipe or other file. Very useful.

The shell allows you to put the standard input for a command along with the command, rather than a separate file, so the shell file can be completely self-contained.

There's something called a `here document`, which takes standard input from the first delimiter until the next delimiter, then substitutes for `$`, `…`, and `\`.

```
<<s         //regular here document
<<\s        //no substitution
<<'s'       //no substitution
```

Looping in the Shell

The shell is a programming language with variables, loops, and decision making capabilities.

A `for` statement reads as follows:

```
for var in list of words
do
    commands
done
```

e.g.

```
for i in *
do
    echo $i
done
```

The `i` can be any shell variable, although `i` is traditional. Note that the var's value is accessed by `$i` but the `for` loop refers to the var as `i`. `*` is used to pick up all the files in the `pwd`, but any other list of args can be used.

You can also write a `for` loop as such:

```
for i in <list>; do <commands>; done
```

You should use the for loop for multiple commands, or where the built-in argument processing in individual commands is not suitable.

Usually, the argument list in a for loop comes from file names, but it can come from anything. E.g.

```
for i in `cat list.txt`
```

Or you can just type in the arguments regularly. E.g.

```
for i in 3 4 5 6; do ln 2 $i; done
```

PROBLEMS

- How to use the `&` sign

- How to use `here documents`.