Introduction to Linux

All of the current ARC systems run an operating system called Linux. Whereas Microsoft Windows and Mac OS X place almost total emphasis on graphical interaction with the operating system, Linux encourages users to do lots of things from the "command line" or "prompt".

Working from a command line has many benefits, especially when dealing with multiple files or performing complex operations on data, and once used to it, many users complain how slow and painful using graphical systems can be. However, it is different and many users find the learning curve off putting. This is where simple tutorials with Linux can help.

Note that this introduction to Linux can be run on any computer running that operating system – all sections will work though in places the output produced by the commands will be slightly different. To keep things simple, however, we suggest you logon onto arcus-b and run the commands on there. To do this log into your workstation, open a terminal, and issue the command ssh command which is used to make a connection to a remote machine:

```
ssh –CX <username>@arcus-b.arc.ox.ac.uk
```

You should replace <username> with the username you have been provided. You should then enter your password when asked, after which a welcome message for arcus-b will be displayed, and you will get a prompt at which you can issue commands.

The Linux File System

On a computer the operating system is the program that relays instructions to the various parts of the computer and makes sure that they are carried out. These instructions can come from you typing commands at the command line, or from an application such as the program you run to perform your research, or a graphical user interface; it doesn't matter from where, the operating system turns those instructions into something the computer can understand.

The Linux operating system is built around the concept of a filesystem. This stores not only the operating system itself, it in fact stores everything for the whole system including your data and program files, files for commands , system and software configuration information, temporary workfiles, and various special files that are used to give controlled access to system hardware and operating system functions.

It is worth making clear at this early stage that Linux is case sensitive – In almost all circumstances a and A are considered to be different. Thus ls is a standard Linux command, while LS is not, and abc.dat is a different file from Abc.Dat.

There are **four types of files** in a Linux filesystem. At this stage the first two are the most important, but we list all 4 for completeness.
*Ordinary files* can contain text, data, or program information. This is the type of file you use to contain your data and programs. Linux filenames can contain any character except for '/' and be up to 256 characters long. However we strongly suggest you avoid characters such as *,?,#,- and & as we shall see these have special meaning in Linux - if you use them in file names you will make your life difficult. Similarly putting spaces in filenames also makes them difficult to manipulate, rather use the underscore '_'. In fact we suggest you stick to alphanumeric characters (i.e. a-z, A-Z [remember they are different!] and 0-9), the underscore and the point (.) when choosing file names

Directories are containers that hold files, and other directories. These are an almost direct equivalent for Windows folders – and indeed in early Microsoft operating systems they were called directories!

Devices: To provide applications with easy access to hardware devices, Linux allows them to be used like ordinary files. There are two types of devices in Linux - block-oriented devices which transfer data in blocks (e.g. hard disks) and character-oriented devices that transfer data on a byte-by-byte basis (e.g. modems and dumb terminals).

Link is a pointer to another file. There are two types of links - a hard link to a file is indistinguishable from the file itself. A soft link (or symbolic link) provides an indirect pointer or shortcut to a file.

The directory structure in Linux is a tree like structure with the so called "root" directory at the bottom, with other directories branching off it. A typical set up for the root directory is as below. Here root contains a directory called home, and home itself will usually contain more directories, and in particular home usually contains your "home" directory, the place you store files when you log in. We will see soon that you can create your own directories.
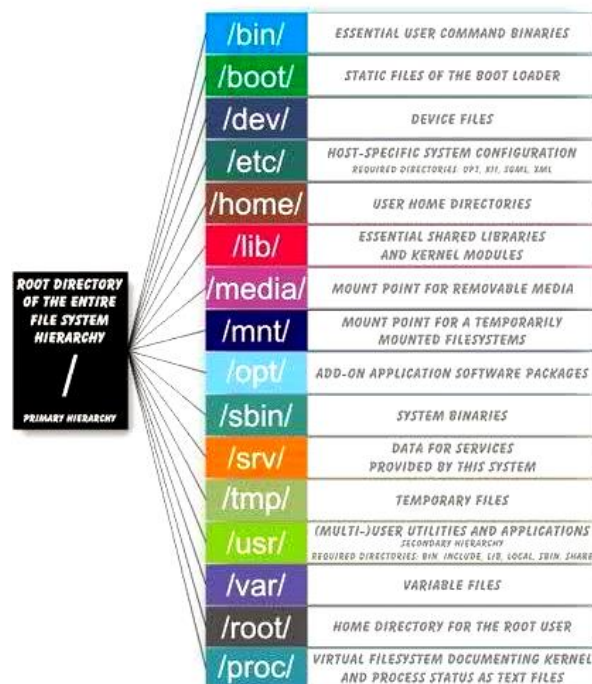
Basic Directory and File handling commands:

At the prompt type `pwd` (print working directory) `pwd` displays the full absolute path to your current location in the filesystem. So immediately after login if you type (note here $ is the prompt)

  `$ pwd`↵
  `/home/projectname/username`   (It will show the current working directory)
e.g. /home/engs-tvg/ouit111 . This means that you are currently working in the directory ouit111, which itself is held in the directory engs-tyg, which in turn is in the home directory, which is held in the root directory. Note this will vary with your login details, but should be similar.
`ls` (list directory)
`ls` lists the contents of a directory. If you don't specify a directory, then the contents of the current working directory are displayed. So, if the current working directory is /,
`$ ls`↵
`bin dev home mnt share usr var boot etc lib proc sbin tmp vol`
Type ls to see what files are currently in the directory you are in.

By default `ls` doesn't show *all* the entries in a directory - files and directories that begin with a dot (.) are hidden. These files usually contain configuration information and normally should not be changed. If you want to see all files, `ls` supports the `-a` (for all) option or "flag":

```
$ ls -a ↵
```

Many Linux commands have many flags that modify their behaviour. Another example with `ls` is the `-l` flag (long listing) to see more detailed information. This can be combined with the `-a` flag as follows:

```
$ ls -a -l ↵
(or,)   $ ls -al ↵
```

Each line of the output looks like this:

permissions	*owner*	*group*		*date*	
drwr-xr-x	ouit01	engs-tvg	4096	July 29 08:30	myprogs
d is type , here a directory			*size*		*name*

There is a lot of information here! At this stage the most important are the name of the file, the date it was last modified, and its size in bytes.

`ls` has many more flags , type *'man ls'* or *'info ls'* to see the options – this also tells use the standard Linux help commands. When using info pressing space means display the next page, and q means quit to the command prompt.

At the moment, if this is the first time you have logged on, you won't have many files in your directory! A simple way to create a file is the `touch` command, which will create a new file if one with the given name does not exist. Try

```
$ touch my.txt ↵
```

and use `ls` to find out how big the file is. Can you work out what happens if you `touch` the file for a second (or third, fourth…) time? Hint use `ls -l`. You will notice that touch is not actually that useful for creating files, we'll see better ways soon!

To create a directory use `mkdir`. Try

```
$ mkdir my_data ↵
```

And use `ls` to find out what has happened. Once you have made a directory you may want to change your place in the file system so that you can work in it. For this use the `cd` (change directory) command. So try

```
$ cd my_data ↵
```

And see what `pwd` outputs now, do you understand it? To check you understand what is going on create a new directory within `my_data` called `results` and change into it, and use `pwd` again. Can you predict the output? Now try

```
$ cd ↵
```

i.e. don't specify a directory. Use `pwd` to find out where you are in the file system now
Finally change into `my_data` and then into `results` and then issue the command

```
$ cd .. ↵
```

(Note the 2 dots after the cd command) Where in the file system are you now? Repeat this command, where are you now? What happens if you do this a third time? Can you predict what will happen if you keep repeating it?

Other useful command are `cp` (copy a file) and `mv` (move, i.e. rename, a file). Change into your home directory and use `ls` to work out what the following command does:

```
$ cp my.txt my_old.txt ↵
```

Now try the command
```
$ mv my_old.txt my_very_old.txt ↵
```
Using `ls` can you see the difference between what `cp` and `mv` do?

To delete a file use rm. BEWARE! By default Linux won't ask you if you want to do this, it will just get on and do it. Thus to get rid of the file you don't want use
```
$ rm my_very_old.txt ↵
```
Try this and yet again use `ls` to see what has happened. Experienced command line users prefer not being asked, in fact one of their many irritations with GUIs is often all those "stupid" questions you get asked. However if you want to be careful you can use the `-i` flag to `rm` which will ask you if you want to remove the file:
```
$ rm -i myfile ↵
rm : remove 'myfile' ?
```
`rm` by default won't remove a directory, instead use the `rmdir` command which will, but only as long as the directory is empty.

Editors
Now you should understand how the filesystem is structured, how to move about in it, and some of the most common file manipulation commands. However we don't yet know how to give the files any content! The most common way is to use an editor – a special program that allows you to type into a file, and change a file's contents. There are many good linux editors (e.g. emacs, gedit, nano) some of which inspire a fairly fanatical following, and there is also vi. Here we'll illustrate use of a fairly simple one, `nano`. Others are broadly similar but the details will vary; if you know one already go ahead and use that.
Start editing by typing
```
$ nano my.txt
```

If the file doesn't exist it will be created. You will then see screen like the image below which displays the file's contents (currently nothing), a prompt at which you can type, and a set of key shortcuts:
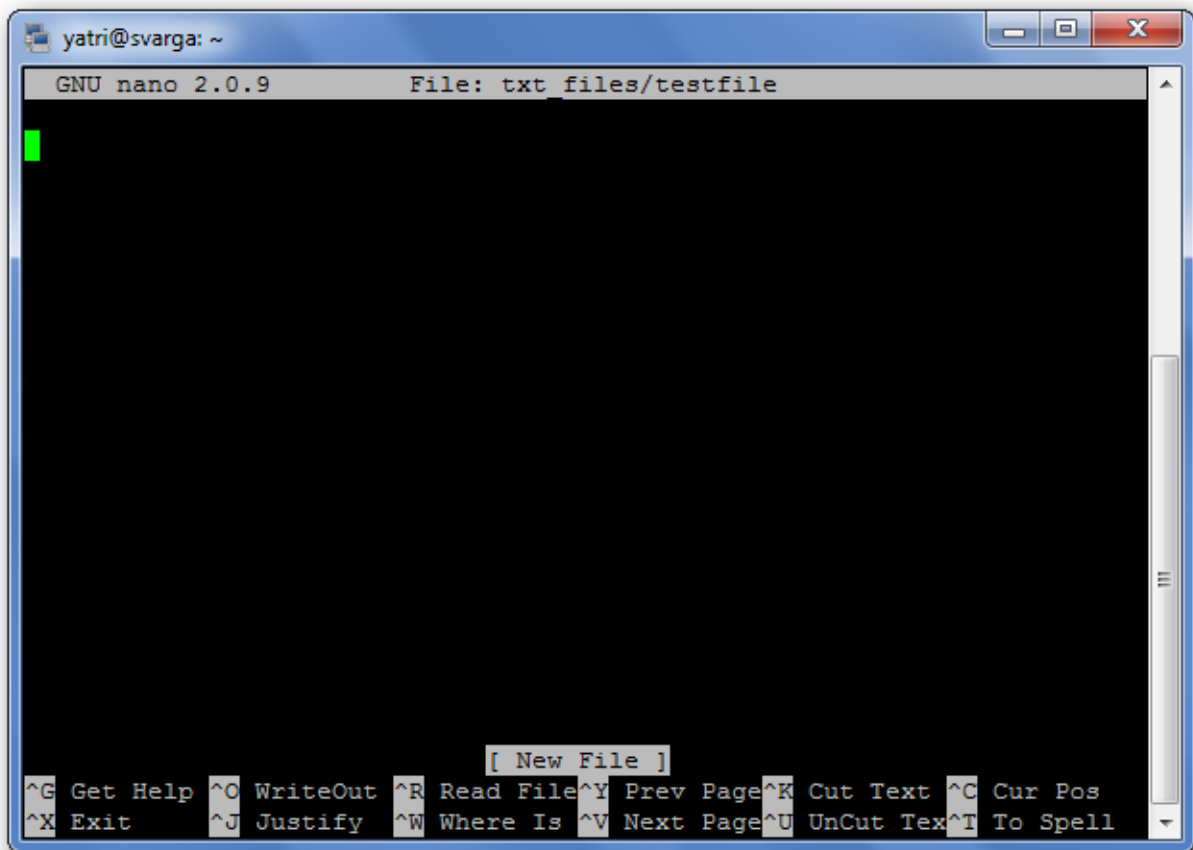
Editing functions such as saving, quitting, justifying, etc. are referred to as "shortcuts" in nano. The syntax means that if you type, for instance, control and X at the same time you will quit nano (i.e. return to the command line) and will be given the option to save the changes you have made to the file. The most common ones are listed at the bottom of the screen, but there are many more that aren't. Note that nano does not use the Shift key in shortcuts. All shortcuts use lowercase letters and unmodified number keys, so Ctrl+G is NOT Ctrl+Shift+G.

Hit Ctrl+G to bring up the Help documentation and scroll down to see a list of valid shortcuts.

```
  yatri@svarga: ~

  GNU nano 2.0.9           File: txt_files/testfile           Modified

  value.  The following keystrokes are available in the main editor
  window.  Alternative keys are shown in parentheses:

 ^G       (F1)              Display this help text
 ^X       (F2)              Close the current file buffer / Exit from nano
 ^O       (F3)              Write the current file to disk
 ^J       (F4)              Justify the current paragraph

 ^R       (F5)              Insert another file into the current one
 ^W       (F6)              Search for a string or a regular expression
 ^Y       (F7)              Move to the previous screen
 ^V       (F8)              Move to the next screen

 ^K       (F9)              Cut the current line and store it in the
  cutbuffer
 ^U       (F10)             Uncut from the cutbuffer into the current line
 ^C       (F11)             Display the position of the cursor
 ^T       (F12)             Invoke the spell checker, if available

 ^_       (F13)    (M-G)    Go to line and column number

^L Refresh         ^Y Prev Page      ^P Prev Line      M-\ First Line
^X Exit            ^V Next Page      ^N Next Line      M-/ Last Line
```

Use nano to provide some contents for the file my.txt in your home directory and save your changes. How big is this file? Use ls -l to find out.

More Commands

Now we can make files with contents it would be nice to display them at the command line without having to open nano every time. Two useful commands to do this are cat and less. This last command is a more modern version of the older more, which you may also come across. Cat just displays the files "as is". Try
$ cat my.txt
And see what happens. This is good for short files but if the file takes more than one screen to display with cat the top will disappear probably before you can read it! This is where less comes in, it asks you before it proceeds onto the next page. Use nano to make sure that my.txt is longer than one screen, and then use less to display with
$ less my.txt
When using less pressing space means display the next page, and q means quit to the command prompt, just as for info. Also useful are head and tail, which show the top and the bottom of the given file respectively. Try these on your file.

Command Line Short Cuts

You will have noticed now that often we repeat very similar commands and it would be useful to be able to repeat, or slightly change, previous commands without having to retype them. At the command prompt hit the up arrow. See how the command you just typed is now displayed? Also see how that using the left and right keys you can move around in the command and change it as required. Hit the up arrow again, you should now have gone back one further command. In fact you

have a complete history of your commands which is searchable an editable! Below is a list of useful short cuts at the command line:

Key or key combination	Function
Ctrl+**A**	Move cursor to the beginning of the command line.
Ctrl+**C**	End a running program and return the prompt
Ctrl+**D**	Log out of the current shell session, equal to typing **exit** or **logout**.
Ctrl+**E**	Move cursor to the end of the command line.
Ctrl+**H**	Generate backspace character.
Ctrl+**L**	Clear this terminal.
Ctrl+**R**	Search command history
Ctrl+**Z**	Suspend a program
ArrowLeft and **ArrowRight**	Move the cursor one place to the left or right on the command line, so that you can insert characters at other places than just at the beginning and the end.
ArrowUp and **ArrowDown**	Browse history. Go to the line that you want to repeat, edit details if necessary, and press **Enter** to save time.
Shift+**PageUp** and **Shift**+**PageDown**	Browse terminal buffer (to see text that has "scrolled off" the screen).
Tab	Command or filename completion; when multiple choices are possible, the system will either signal with an audio or visual bell, or, if too many choices are possible, ask you if you want to see them all.
Tab Tab	Shows file or command completion possibilities.

The Tab requires further explanation: if you want to change into the directory `directory_with_a_very_long_name`, you don't need to type that very long name, on the command line `cd dir` (first 3 letters of your long name), then you press Tab and the shell completes the name provided no other files starts with the same three characters. If there are no other items starting with "d", then you can just type `cd d` and then Tab. If more than one file starts with the same characters, the shell will signal this to you, upon which you can hit Tab twice with short interval, and the shell presents the choices you have.
For example if you have 3 directories starting with the same letters:
`gnome-run1  gnomics   gno-test`
if you use `$ cd gno` after the first three characters and hit Tab again, the shell completes the directory name, showing all 3 options. If you now add an m and hit tab again you will get
`gnome-run1   gnomics`
as these are the only 2 of the 3 that now match what you have at the command line. Finally if you were to add an e and hit tab again what do you expect to happen? Create these 3 directories and try this out.
At this point we have reached a stage where you have enough knowledge to do most of your work on ARC systems! However there a few other topics which, while not absolutely necessary, can be useful to aware of, especially redirection which we consider below. These begin to illustrate the power of the command line for complex operations on multiple files.

Specifying Multiple Files and Wildcards
So far we have worked with one file at a time. However Linux allows us to work with many files at once, and many commands can be applied to multiple files at once. As an example use nano to create 3 files, 1.txt, 2.txt and 3.txt and then issue the command

```
$ cat 1.txt 2.txt 3.txt
```
What do you see? This is how `cat` gets its name, from con*cat*enate.

However it's boring to have to type out all these file names. Instead you can look for multiple filenames using special pattern-matching characters, often called wildcards. The rules are:

'?' matches any single character in that position in the filename.

'*' matches zero or more characters in the filename. A '*' on its own will match all files. '*.*' matches all files containing a '.'.

Characters enclosed in square brackets ('[' and ']') will match any filename that has one of those characters in that position.

For example:

??? matches all three-character filenames, i.e. `ls ???` lists all files in the current directory that have exactly 3 characters in their name

?ell? matches any five-character filenames with 'ell' in the middle.

he* matches any filename beginning with 'he', e.g. hello, help, heterogeneous but NOT Henrietta – Remember Linux is case sensitive!

[m-z]*[a-l] matches any filename that begins with a letter from 'm' to 'z' and ends in a letter from 'a' to 'l'.

Thus [Hh]e* matches all of hello, help, heterogeneous and Henrietta

Note that the Linux shell performs these expansions (including any filename matching) on a command's arguments before the command is executed.

Try
```
$ ls -l *.txt
```
In the directory where you created the 3 files above. Do you understand the output? What do you think
```
$ cat *.txt
```
will display? Try it!

Quoting

As we have seen a number of special characters (e.g. '*', '-','{' etc.) are interpreted in a special way by the shell. In order to pass arguments that use these characters to commands directly (i.e. without the filename expansion we have just seen etc.), Linux uses special quoting characters which forces them to be interpreted as-is rather than in any special way. There are three forms of quoting:

1) For a single character insert a '\' (backslash) in front of the special character.

2) For a string of characters use double quotes (") around the string to prevent *most* expansions.

3) For a string of characters use single forward quotes (') around the string to prevent *all* expansions.

In the directory with 1.txt in try the command
```
$ ls \*.txt
```
What does this display? Do you understand now why we recommend avoiding *, ? and similar in file names? Also try
```
$ ls "*.txt"
```
There is a fourth type of quoting in Linux. Single backward quotes (`) are used to pass the output of a command as an input argument to another. For example try the following, which also illustrates two new commands. Use the `man` command to find out what they do.
```
$ hostname
login11
$ echo This machine is called `hostname`
This machine is called login11
```

Finding Files

If you don't know the exact location of a file there are ways of finding it using the *find* command.

```
$ find directory -name filename -print
```
`find` will look for a `file` called filename in any part of the directory tree with `directory` at the top. You can also use wildcard characters :
```
$ find /home -name "*.txt" -print
```
will search all user directories for any file ending in ".txt" and output any matching files (with a full absolute or relative path). Here the quotes (") are necessary to avoid filename expansion.
`find` can in fact do a lot more than just find files by name, as usual `man find` will provide information.
A particularly useful form of this is
```
$ find . -name "*.txt" -print
```
Note the period (.) after the find command – this is Linux shorthand for the current directory. Thus this will find all files ending in .txt in the current directory and all directories underneath it.
Create files whose names end in .txt in the directory structure you have created and use them to practice using find.

Finding Text in Files

The command `grep` (general regular expression print*)* searches files for lines that match a given pattern. For example
```
$ grep Twinkle star.txt
```
will print out all lines that contain the text Twinkle (but not twinkle - remember Linux is case sensitive) in the file star.txt. Can you work out a way so that grep does find all lines with both Twinkle and twinkle? Hint: `man grep` may be useful, or another way is via the wildcards we saw earlier, many of these work with the text for grep, but think carefully about quoting! Use nano to create a file to test your ideas. We can now begin to see the power of the command line. Let's say we want to find every shell script (for us a file whose name ends in `.sh`) below the current directory that contains the text SBATCH. This will do it:
```
$ grep SBATCH `find . -name "*.sh" -print`
```
searches all shell script files in the directory tree below the current directory for lines containing "SBATCH". Think carefully about this command, make sure you understand it.

Sorting Files

`sort` sorts lines contained in a group of files alphabetically (or numerically if the -n flag is specified) numerically.
```
  $ sort file1.txt file2.txt
```
outputs the sorted concatenation of files input1.txt and input2.txt.

Redirecting Input and Output

Our commands are getting increasingly complex. Sometimes it would be nice if we could save the output to a file. This is very easily done in Linux.
Technically every command writes its output to something called Standard Output. By default standard output in our terminal is going to the terminal. It is possible to change this so that standard output goes to a file instead by use of the > character. Thus
```
$ echo "Hello"
```
Will write Hello to the screen (try it!). But
```
$ echo "Hello" > hello.txt
```
will write the Hello to a file called hello.txt (again, try it!). Be a little careful as if hello.txt already existed it will overwrite it, and as we are using Linux we will get no warning. To see this try
```
$ echo "Bye" > hello.txt
```
What is the contents of the file now? If you want to avoid this use >> to append to a file – what does hello.txt contain of you now do?
```
$ echo "Hello again" >> hello.txt
```

Similar to the above commands take their input from Standard In. So far this has been connected to the keyboard, but it is also possible to redirect this using < . This will cause a command to read its input from the given file.

Redirection is especially important when we run our jobs in Batch mode, which is the main way of using the ARC systems. Here you will find you your program is running in a way that does NOT allow you to type commands into it, and is NOT connected to a screen. In this case I/O redirection may be the only way to allow your program to read its input!

Pipes:

In the previous section we saw how to redirect the output of one command into a file. It is also often very useful to redirect the output so that it acts as the input for another command. To do this you use and operator pipe denoted by the "|" symbol.

Consider the following command, an example of a pipeline:

```
$ cat listfile | sort | uniq
```

listfile has a list of 10 names, some duplicated. Can you see how this command will generate a sorted list of unique names in the file? `cat` we know writes the file to standard output. However now, instead of going to the screen because of the pipe it acts as the input for `sort` command. Note the difference between > and |, the former redirects standard output of a command to a file, the pipe redirects the standard output of one command to the standard input of another command. `sort` we know sorts its standard input into alphabetic order and writes that to standard output. This output, because of the second pipe, acts as the input for the `uniq` command. This is a new one – it removes all neighbouring duplicate lines in a file, and thus gives us our desired result. Create a file with suitable contents to test this and make sure you understand how this works.

File Compression and copying files

`tar` backs up entire directory structures and files into a single archive file.

```
$ tar -cvf archivename list_of_files to_be archived
```

where archivename will usually have a .tar extension. So for example

```
$ tar -cvf my_files.tar .
```

will create a file called `my_files.tar` that contains all files in the current directory and all directories below it (note the . at the end, do you remember what this means in Linux?) In the above c=create, v=verbose, and f=file. To list the contents of a tar archive, use the tabulate option:

```
$ tar -tvf archivename
```

To eXtract files from a tar archive, use

```
$ tar -xvf archivename
```

This can be very useful for transferring many files between machine – use tar to create a single file containing everything you want, transfer that single file, and then extract it on the remote machine. However tar files can get very big and so can take a long time to transfer, in which case a compression utility like gzip can be very useful.

To compress files, use:

```
$ gzip filename
```

To reverse the compression process, i.e. once on your home machine, use:

```
 $ gzip -d filename
```

But how to transfer the file between machines? The simplest method is to use the `sftp` command (secure file transfer protocol). From a terminal on your workstation issue the command

```
$ sftp <username>@arcus-b.arc.ox.ac.uk
```

Again replacing <username> with the username you have been given for arcus-b. You will be asked for you password and then given a prompt. Here some familiar commands will work, including `cd`. Use this to change into the directory where the compressed tar file is, and then issue the

```
get <filename>
```

command, replacing <filename> as appropriate. This will get the file from arcus-b and put it on your workstation. `put` is also available to take file from your workstation and put them on arcus-b.

Shells and Shell Scripts

So far we have typed everything into a terminal. However we can store commands in a file and then run that file, effectively generating a new Linux command from the operations given in the file. This is called a shell script, a shell being the base interpreter that is sorting out what all the commands we use actually mean. Technically we have been using the bash shell, but there others. However at this level it makes little difference which one you use.

There are three extra things you need to know when storing commands in a file

1. How to give execute permission to the file
2. How to run the file
3. How to make sure the commands in the file are interpreted as you want them to be

For the first each file has a set of permissions – we actually saw these in the output of `ls -l`. One of these permissions is whether the file can be executed or run. By default files cannot be run under Linux, this protects you against accidentally running something which shouldn't be, or worse running something maliciously installed on your system. However as we want to run our file we need to add the execute permission. You do this via

```
chmod +x filename
```

To run the file the simplest way is to be in the same directory and then type

```
./filename
```

This means try to run a file called `filename` in the current directory – remember what . means in Linux. To know more about this we have to look at the PATH environment variable, but to get started this simple recipe will do.

Finally how do we make sure Linux understands our file and runs it correctly? To make sure that the script is interpreted by the bash shell you should put

```
#!/bin/bash
```

As the very first line in your shell script. This sets the so called magic number for the script, and ensures that bash is used to interpret it, see

https://stackoverflow.com/questions/8967902/why-do-you-need-to-put-bin-bash-at-the-beginning-of-a-script-file

for more details.

Thus a simple example of a shell script and its use might be as follows

```
$ cat count_text_files
#!/bin/bash
echo There are `ls -l *.txt | wc -l` text files in this directory
$ chmod +x count_text_files
$ ./count_text_files
There are 25 text files in this directory
```

Look at this example carefully, if you understand what is happening you know plenty enough to use ARC machines well!

We need to know about shell scripts as when we move to batch systems where, as mentioned previously, there is no keyboard to type commands at! Thus in this case you must put your commands in a file, and these commands will then be executed. However save for being in a file they behave just as we have seen already.

Extra Exercises 1

1. Nearly all commands in linux are held in a file similar to the shell script you looked at above . The `type` command shows where that file is. What output does
   ```
   $ type less
   ```

give? Change into that directory, can you find the file corresponding to the less command? What about the other commands we have seen, are they in the same directory? If not where are they?

2. How many files do you have ending in .txt in your home directory? The wc command may well be useful for this, especially with the –l flag, so first take a look at `man wc`
3. How many files do you have ending in *.txt in *all* your directories?
4. Using an editor create a file containing a list of names with some duplicates as described above. Experiment with the `sort` and `uniq` commands to see how this works. How in one command might one count how many unique names you have in the file?