PHYSICS 2T C Programming Under Linux

Linux Lab 1 An introduction to the command line

INTRODUCTION

This lab will serve as a basic introduction to the command line using BASH – the Bourne Again SHell.

Once you have finished this lab you should be relatively comfortable using the command line for basic file management.

HOW TO ANSWER THE QUESTIONS

At the end of each section there are a set of questions, some of which will ask you about the tasks you have already completed in the lab and others will require you to do other research. Do not be put off if the answer is not obvious, read the Getting Help section if you are stuck.

Questions for this lab should be answered in `answers.txt` which can be found in the `Lab01` folder.

GETTING STARTED

Start by opening the Terminal application. If you are using Gnome this can be found under Applications → Accessories → Terminal. This will gave you access to a BASH shell which will allow you to enter commands.

Verify that you are in your home directory, you can do this by typing **`pwd`** (Print Working Directory) into the terminal. It should be similar to `/home/0800890`.

Next, enter `ls`. You will be present with a list of the visible files and folders in the current directory.

GETTING HELP

You can find more information about the **`pwd`** and `ls` commands using their manual pages. All of the labs can be completed using the information provided in the manual pages. As such, it is essential that you get used to using them.

Manual pages can be accessed via `man command`, where `command` is replaced by the name of command you wish to know more about. Take a quick look at the manual pages for BASH (`man bash`) and ls (`man ls`). You can exit the manual pages by pressing `q`.

It is possible to search for man pages using the `man -k` and `apropos` commands. `man -k delete` and `apropos delete` will both search for manual pages which have a description containing the word “delete”.

Manual pages exist for more than just user commands, `man intro`, which serves as a basic introduction to the command line and bash, is one example.

As more commands are introduced, you should read their particular man pages and get an idea of what the command can do. Reading the man pages and using the commands is by the far the most effective way of learning bash.

In addition, this lab comes with `Commands.pdf` which is a list of the most common commands and their most useful options. Try some of them out before you continue.

If you are stuck please remember to ask the demonstrators who will be available during labs.

You are, of course, allowed to use lecture notes, books and the internet while completing the labs.

FILES AND DIRECTORIES

You are going to create a basic directory structure which could be used for storying data and documents for an experiment. Some large experiments result in hundreds or even thousands of files being created – sorting and storing them in an organised manner is essential.

1. Assure that you are in your home directory using the `pwd` command.
   If you find that you are elsewhere then the command `cd` without any arguments will take you to your home directory.
2. Create a directory called `experiment`. If you are unsure how to do this then consult `Commands.pdf` or `man mkdir`. The new directory should be listed when you run `ls`.
3. Use the `cd` command to change your working directory to the `experiment` directory.
4. Create 3 new directories named `docs`, `data`, and `results` in the experiments directory. You should finish with the following directory structure which can be verified using `ls -R`.
   ```
   ~/experiment/
   ~/experiment/data
   ~/experiment/docs
   ~/experiment/results
   ```
5. Use the `cd` command to navigate back to your home folder then the `Lab01` directory.
6. You are going to run a BASH script which will generate some sample data files. Use `ls` to check for a file called `generate.sh` and run the script using `./generate.sh`. It may take a few seconds to finish running, when it does you can use the `less` and `more` commands to take a look at the source of the script.
7. Without changing directory, check the contents of the `experiment` directory using `ls ~/experiment`. The "~" will be automatically substituted by the absolute path of your home directory (You can see this in action using `echo ~`). The `experiment` directory should now contain 50 `.data` files.
8. Navigate back to the `experiment` directory. You need to move the data files into the `data` directory. Start by typing `mv 2` **but do not run the command yet.** Press the tab key twice. This will give you a list of all the files in the directly begging with 2. Press 4 and then tab again. This will complete the rest of the file name. Do the same again to add the name of the destination directory so that the entire command is `mv 24.data data/`.
   Check that the file has been moved using `ls data/`.
9. Moving each file individually would be slow. There is a much quicker way of selecting all `.data` files using the * character. Run the command `mv *.data data/`. The * will match any number of any characters (including nothing or null) so that any file which ends in `.data`

will be selected.
Change into the `data` directory and use `ls` to verify all the files have been moves.

10. You want to filter some specific data files to look at later. Create a new directory called `5s` and copy, using the `cp` command, any file which has a name containing 5 into it. *Remember you can use the * character to move multiple files at once.*
11. The plan is to perform several experiments over the new few weeks. As such, you want to store them all in an organised manner. Create a new directory called `experiments` in your home directory which will store all of the experiments and their data.
12. Copy the `experiment` directory into the `experiments` directory, renaming it to `exp1` (You will need to use the `-R` argument to copy a directory). You should only need a single command to do this.
13. Try to remove the old `experiment` directory using the `rmdir` command.
14. Use the `rm` command to remove the directory. Check its man page or `Commands.pdf` for how to delete a directory and its contents.

QUESTIONS

1. From your home directory, what is the output of `ls -R experiment*`?
2. What command did you use in step 12 to copy and rename the directory? You can use the `history` command to get a list of your most recently used commands. You can also use the up and down arrow keys to scroll through your previous commands.
3. Why did the `rmdir` command give an error in step 13 of the **Files and Directories** section?
4. What command did you eventually use to delete the directory?
5. Read the man pages for `ls`.
   a) What is the difference between `ls -R` and `ls -r`?
   b) What does `ls -a` do?
   c) What is special about the name of a hidden file?
   d) What is the name of the hidden file in the `~/experiments/exp1/data` directory?
6. Every directory contains hidden directories named `.` and `..`
   a) Where do they refer to?
   b) How can you list all of the files and directories, both hidden and visible, except . and ..?
7. How would you sort files
   a) by size?

b) by modification time?

8. How would you use `cd` to change from `~/experiments/exp1/data` to `~/experiments`
   a) using relative paths?
   b) using absolute paths?
9. Open `generate.sh` in `less` and `more`.
   a) How can you force `less` to display line numbers?
   b) What is the contents of line 21 of `generate.sh`
10. Read the man pages for `head`, `tail` and `wc`
    a) How many lines do `head` and `tail` show by default?
    b) How can you make `head` show 5 lines?
    c) Which argument makes `wc` show only the number of lines?

FILE PERMISSIONS

Linux is, in general, very good at looking after itself. There are times, however, when you'll need to perform some basic system administration.

One example of this is changing file permissions. Most Linux file systems have a simple but powerful set of file permissions that dictate who can do what with particular files and folders. Being able to mange them is extremely important – especially when writing software.

1. Use the `touch` command to make a new file called `important` in the `Lab01` folder.
2. Use the *long listing format* for `ls` to display the file permissions and other information, check the manual pages if you're unsure how to do this.

   Look for the line similar to the one below:

   ```
   -rw-r--r--  1 james james   0  2010-06-15 15:50 important
   ```

   We're most interested in the first part, `-rw-r—r--`. The file is readable by everyone but writeable only by its owner (lecture 2 contains information on reading file permissions).
3. Since the file is important you're going to protect it by making it read-only. This is done using the `chmod` command. Run `chmod a-w important`. This will modify the file permissions for all users (`a`) and remove (`-`) their write permissions (`w`).
4. Open the file in a text editor, add some text and try to save the file, it will not let you. Now try to delete the file using `rm important`. You will be asked to confirm the deletion, offering you a second chance before the file is deleted forever.

File permissions also determine whether or not a file is allowed to be executed (run). In the interest of security, files are set to be non-executable by default. You will need to learn how to make files executable before writing bash scripts in lab03.

1. In the `Lab01` directory there is a C program called `secret` containing the answer to one of the questions for this lab. If you to try run it you will get a “permission denied” error since it's not executable. Use `chmod` to make it executable using the `+x` flag.
2. Run it using `./secret` and note down the answer for later.

The `chmod` command and file permissions can be quite confusing. Read the manual pages if you have not already and get used to modifying file permissions.

QUESTIONS

11. There is a file with the following permissions `-rw-rw---x`

    a) Describe who can do what with the file.

b) Which command would change the permissions so that only the owner can write to and execute the file?

12. How can you tell, using only the file permissions, if a file is a directory?

13. Read the man pages for `rm`.

    a) How can you force `rm` to ask before deleting every file?

    b) How can you stop `rm` from asking before deleting a write protected file?

PHYSICS 2T C Programming Under Linux

Linux Lab 2 Advanced command line usage

INTRODUCTION

In this lab you will learn how to use the command line in an effective manner and make use of some of the more advanced features available in BASH.

MANAGING PROCESSES

When developing scripts and programs is it essential you know how to properly manage running processes. It is not uncommon to accidentally create a script or program that contains an infinite loop and have to stop it manually.

Fortunately, managing processes is a relatively simple thing to do. top is similar to the Task Manager found in Microsoft Windows. Run it, take a look at some of the information then use q to quit. Top is very useful for a quick overview of the system and for continued monitoring. In general, however, it is overkill for what you need.

In the Lab02 directory there is a script called **`loop.sh`** which you will need to make it executable before you will be able to run it.

When you start the script, it will repeatedly print the message “Hello World”. The quickest, and often most desirable way of stopping a misbehaving script is to use the **`ctrl + c`** combination. This will terminate the program and return you to the command prompt.

Sometimes you may not want to terminate a program, but pause it while you do something else. The **`ctrl + z`** combination allows you to do this.

1. Open **`xclock`** using **`xclock -update 1`** and pause it using **`ctrl + z`**
   It should no longer be ticking. (Make sure you have the terminal window selected for this to work)
2. Use the **`jobs`** command to get a list of the currently running jobs and their status, xclock should be listed.
3. Start copies of **`top`** and **`gnome-calculator`**, pause them both using the **`ctrl + z`** combination.
4. Run the **`jobs`** command again to see a list of all the jobs currently running.

5. james@x100e:~/Physics-2T/labs/Lab02$ jobs
   [1] Stopped xclock -update 1
   [2]- Stopped top
   [3]+ Stopped gnome-calculator
6. The number inside the square brackets is used to identify the job. **fg 2**, for example, would bring **top** into the foreground and restart it.

   Use the **fg** command to restart **gnome-calculator**.
7. Perform a quick calculation using the calculator then use **ctrl + z** to pause it again.
8. You can send a process to the background using the **bg** command. Sending a process to the background allows it to run without interfering with the bash prompt.
9. Use the **bg** command to send gnome-calculator to the background. jobs will now show the process as Running.

It's also possible to start a process in the background by adding an ampersand (&) after the command like so: **gedit &**. This is very useful for quickly launching a GUI application from the command line and leaves the terminal free to be used. Just like using the **bg** command, anything started this way will be listed by jobs.

Another command which you should be aware of is **ps**, which displays a list of the current active processes. The **ps** command provides you with a PID (process id) which you can then use in with the kill command to kill that process.

Note: The PID of a process is NOT the same as the job number provided by the jobs command.

1. Start a copy of **xclock** in the background (remember to add &).
2. Use the **ps** command to check that it is running and note its PID.
3. Use the **kill** command to terminate the **xclock** process.

For more information about the **ps** and **kill** commands, read their man pages. The **ps** command in particular has many options for filtering which processes it lists.

There are also a few other commands working looking at. Read the man pages for **pkill**, **killall** and **pidof** before continuing.

QUESTIONS

1. Start top on your computer.
   a) What is your computer's current load average?
   b) How many processes are currently running?

c) How would you show only processes started by your user?

2. What effect does an ampersand (&) have when put after a command?
3. Using only **`ps`** and **`wc`** how could you count the number of running processes.

VARIABLES AND THE PATH VARIABLE

In bash, variables are created when they are set. They do not have types, every bash variable is considered and stored as a string.

Note: Variables are case sensitive, so **Foo** and **foo** are two different variables.

Setting a variable is as simple as **`LAB="Lab02"`** (be careful not to leave a space at either side of the equals sign).

To read an variable you prefix its name with a dollar **`($)`** sign like so: **`$LAB`**. This is most often used in conjunction with the echo command: **`echo $LAB`**.

Variables tend not to be used very often from the command line however when they are it often involves editing the **`PATH`** variable.

The **`PATH`** variable is an example environmental variable. Environmental variables are used by bash to configure and store information about the current environment. You can see a list of your path variables, along with their current value, using the **`env`** command.

The **`PATH`** variable itself contains a list of colon separated directory paths which are search while looking for a command to execute. Any command in a directory listed here can be run by you from any directory without needing to enter its path.

1. Use the **`echo`** command to print the contents of your **`PATH`** variable.
2. Create a new directory in your home folder named **`bin`**. This is where you will store any scripts or binary files you want to be able to use anywhere on the system.
3. Add the new directory to your **`PATH`** variable using **`PATH="$PATH:~/bin"`**.
4. Check that the new directory has been added to your **`PATH`**.
5. Copy the script called hello from the Lab02 folder into **`~/bin`** and make sure that it is executable.
6. Navigate to several different directories and run the **`hello`** command, it should work anywhere you run it.
7. Close the terminal window and re-open it.
8. Try running the **`hello`** command again.

Variables are not persistent. That is, they only exist in the session they were created. You can, however, add them to your **`.bashrc`** file so that they are always set.

QUESTIONS

4. What security issues could arise from adding a directory to your PATH variable?
5. Consider the following variables:

   A=Apple

   B=Ball

   C=Cat

   What would be the output of following:

   (Use echo if you're not sure)

   a) $A b) AB c) $AB d) $Cat e) ${C}at f) “$A $B $C”

   g) '$A $B $C' h) \$A

REDIRECTION AND PIPES

REDIRECTION

Bash allows the redirection of a command's output using the the greater-than symbol: **`>`**

For example, **`ls -l > ls.txt`** will redirect the output of **`ls -l`** and place it into **`ls.txt`**. This can be quite useful if you want to save the output of a command for processing later.

It's can also be very useful in conjunction with the cat command. You can quickly concatenate 2 files together using **`cat data1.txt data2.txt > data3.txt`**. The contents of both files will be added together and saves as **`data3.txt`**. Try this using the 2 data files in the Lab02 directory.

PIPES

Pipes are a similar concept to redirections, however the output is sent (piped) to another program instead of a file.

To pipe a command's output you use the pipe **`|`** symbol.

Pipes can be extremely useful for chaining commands together. For example, you may want to find the 5 most commonly occurring values spread across several files. To do this you would need to concatenate all the files together, sort them, count how many times each line occurs, sort the file again to find the highest and then use tail command to get the last 5 lines.

It would look something like this:

```
$ cat data1.txt data2.txt > all_data.txt
$ sort all_data.txt > sorted_data.txt
$ uniq -c sorted_data.txt > uniq_data.txt
$ sort uniq_data.txt > sorted_uniq_data.txt
$ tail -5 sorted_uniq_data.txt
```

Alternatively, you could pipe the output of each command, allowing you to do all of the above in a single line and without creating several files to store the intermittent results.

```
$ cat data1.txt data2.txt | sort | uniq -c | sort | tail -5 > sorted.txt
```

QUESTIONS

6. the example using pipes, why was the data sorted **before** being sent to the **uniq** Incommand?

   Hint: Read the description of the uniq command in its man pages

7. Read the man pages for the **grep** command.

   a) In a single sentence, what does the grep command do?

   b) Which command would search for lines containing the word “Chapter” in the ./Lab02/extras/Bash-Begginers-Guide.txt file?

8. Explain what the following command does and it works:

   ```
   sort ~/.bash_history | uniq -c | sort | tail > commands.txt
   ```

9. What is the difference between a single > and double >> when used for redirection? (Hint: Try directing output to a file several times using them)

PHYSICS 2T C Programming Under Linux

Linux Lab 3 An introduction to bash scripting

INTRODUCTION

This lab will introduce you to writing basic bash scripts and using control structures.

You will find this lab much easier if you read through some of **`man bash`** before starting. It is incredibly detailed and although intimidating, a very useful resource. For each section an example shell script is included in the Lab03 directory, it's important to look at these and try running them.

HELLO WORLD

It's customary for the first example of a new language to print “Hello World”. This lab will be no exception.

1. Create a new file called **`helloworld.sh`** in your **`Lab03`** directory.
2. Add the following and save it:
   ```
   #!/bin/bash
   echo “Hello World”
   ```
3. Use **`chmod`** to make **`helloworld.sh`** executable.
4. Run it using **`./helloworld.sh`**

Congratulations, you've just written and run your first bash script. The process is the same for every bash script you write, the only thing that will changes is be the contents of the file.

Note: To make your life easier, you may want to add the `Lab03` folder to your PATH variable, allowing you to run scripts using only their name (e.g. `helloworld.sh` instead of `./path/to/helloworld.sh`).

QUESTIONS

1. Write a script named **`Task1.sh`** which will create the following directory structure in the current directory:
   ```
   ./data/
   ./data/processed/
   ./docs/
   ```

CONDITIONALS

Conditionals are a core part of a bash scripting, they control the flow of a script and almost all scripts you write will use them.

Conditionals, as the name would imply, allow you to execute certain parts of the script if a condition is met. This is probably best demonstrated with an example.

```
1)  #!/bin/bash
2)
3)  if [ -f “processed” ]; then
4)      echo “There is a file called processed.”
5)  fi
6)
7)  exit 0
```

The code between lines 3 and 5 (**`if[ ]; then`** and **`fi`**) is only executed if the condition between the square brackets on line 3 are met. In this case, line 4 is only executed if a file called **`processed`** exists in the current directory.

`[` is synonymous with the **`test`** command. Line 3 could have been written:

`if test -f “processed”; then`

Of course, there are many more conditionals, the table below details the ones you're most likely to use, however you may want to read **`man bash`** for a full listing.

String Comparisons	Result	File Conditionals	Result
`string1 = string 2`	True if the strings are equal	`-d file`	True if file is a directory
`string1 != string2`	True if the strings are different	`-e file`	True if the file exists
`-n string`	True if the string is not null	`-f file`	True if the the file is regular
`-z string`	True if the string is null	`-r file`	Tue if the file is readable
Arithmetic Comparisons	**Result**	`-s file`	True if the file has a non-zero size
`exp1 -eq exp2`	True if both are equal	`-w files`	True if the file is writeable
`exp1 -ne exp2`	True if both are different	`-x file`	True if the file is executable
`exp1 -gt exp2`	True if exp1 is greater than exp2		
`exp1 -ge exp2`	True if exp1 is greater than or equal to exp2		
`exp1 -lt exp2`	True if exp1 is lesser than exp2		
`exp1 -le exp2`	True if exp1 is lesser than or equal to exp2		
`! exp`	Inverts exp. True if exp is false, false if exp is true		

Much like C, bash also has “else if” and “else” commands named **elif** and **else** respectively.

```
if [ condition ]; then
    …
elif [ condition2 ]; then
    …
else
    …
fi
```

Take a look at **conditionals.sh** in the Lab03 directory for more examples.

QUESTIONS

2. Copy your script from Question 1, **Task1.sh** to **Task2.sh** and modify **Task2.sh** so that it checks if any of the directories already exist. If they do, it should `exit` with an error code and message.

LOOPS

The ability to use loops is what makes bash scripting the perfect tool for repetitive tasks.

FOR LOOPS

For loops do something for every value in a range of values. A basic **`for`** loop looks something like:

```
for value in lots of values
do
      echo $value
done
```

Take a look at **`for.sh`** in the **`Lab03`** directory for examples.

WHILE LOOPS

For loops are useful when looping over a series of strings, however, they are not as useful if you do not know in advance how many times the loop needs to be executed. While loops are much better in this respect.

While loops continue while a condition is true. It is perhaps easiest to think of them as an if statement which will repeat until the condition becomes false.

```
While [ condition ]
do
      …
done
```

Take a loop at **`while.sh`** for some **`while`** loop examples.

QUESTIONS

3. Write a script (**`Task3.sh)`**which checks and reports if the current directory is the end of a branch (i.e. there are no directories inside the current directory).
4. Write a script (**`Task4.sh)`**which generates 100 random numbers and uses redirection to store them in a file called **`random`**.
   Hint: You can generate random numbers using the **`$RANDOM`** variable and can specify a numeric range by using **`{1..10}`**. Use **`>>`** to append to a file.

VARIABLES & COMMAND SUBSTITUTION

Setting and using variables in a bash script is exact the same as using them from the command line.

```
var=”Hello”
echo $var
```

There are, however, a few extra things you should be aware of.

When doing a comparison between two variables you should always surround them in double quotes. This is particularly important when the value of the variable could contain spaces. For example:

```
$test=“this has spaces”
if [ $test = “this has spaces” ]; then
      …
fi
```

will actually result in an error because $test will be replaced by **`this has spaces`** and bash will attempt to compare **`spaces = “this has spaces”`**. This is also helps stop an error if the variable is empty as **`[ = “something ]`** gives an error where as **`[ “” = something ]`** is valid.

There are several important environmental variables which are only used inside scripts. Below is a table of the ones you're most likely to need. **`man bash`** has a full list.

Name	Value
`$0`	The name of the script
`$#`	The number of parameters passed to the script
`$$`	The process ID of the script
`$1, $2, $3...`	The parameters given
`$@`	All of the parameters passed to the script
`$?`	Exit status of last command

Bash also has a feature called Command Substitution. That is, you can set the value of a variable to be the output of a command. There are 2 ways of doing this. The first is to surround the command in backticks (The quote like characters usually inserted by the key above the tab key) like so: **`` `ls- l` ``** or you can used the following structure: **`$(ls -l)`**. The second method is preferred although both are valid.

Take a look at **`variables.sh`** for some examples.

QUESTIONS

5. Write a script (**`Task5.sh`**)which loops through its arguments and prints if they are a file, directory or do not exist.
6. Write a script (**`Task6.sh`**)which checks for a **`.bak`** (backup) file for every **`.data`** file in the current directory (i.e. for every **`xxx.data`** file there should also be a **`xxx.data.bak`** file). Your script should report if there is a missing **`.bak`** and create one. Run your script in the **`data`** directory to check it works.

PHYSICS 2T C Programming Under Linux

Linux Lab 4 Advanced Bash Scripting

INTRODUCTION

Lab 3 introduced the fundamentals of bash scripting: conditionals, loops and variables. These 3 things alone make up the overwhelming majority of what you need to know to write any bash script. This lab will show you how to get input from the user, read files and write functions in bash as well as give you more experience writing bash scripts.

GETTING USER INPUT

Reading user input from the command line is done using the **`read`** command. The read command reads in a line of text and stores it in a variable. **`read name`**, for example, reads in a line of text and stores it in a variable called **`name`**.

```
#!/bin/bash
echo "What's your name?"
read name
echo "Hello, ${name}!"
```

The read command does not have a man page, however you can find out more using **`help read`**. **`input.sh`** also contains some examples.

QUESTIONS

1. Write a script **`(Task1.sh)`** which finds individual files in the current directory with a size of 0, ask for confirmation, and deletes them. Hint: The **`-s`** flag will test if a file is zero size in your conditional statement.

READING FILES

There are several different ways to read a file line-by-line, however easiest and shortest methods use the **`cat`** command to pipe the contents of the file to a loop containing the read command.

Here is an example:

```
#!/bin/bash
cat file.txt | while read line;
```

```
do
	echo ${line}
done
```

Take a look at **readfile.sh** in the **Lab04** directory for more examples.

QUESTIONS

2. Write a script **(Task2.sh)**which prints the word count, using the **wc** command, of each line in **help-read.txt**.

FUNCTIONS

Bash has the ability to use functions, just like C. Creating a function is as simple as:

```
# Create the function
function_name() {
	function_code
}

# Run the function, it do not require parenthesis ()
function_name
```

Just like scripts, functions can have parameters too. They work in exactly the same way as passing parameters to a script however **they are completely separate excluding the $0 variable.** That means that the variable **$1** is different from **$1** in the rest of the script. **$0**, however, will always be the name of the script itself.

```
#!/bin/bash

func() {
	echo “${0}, ${1}”
}

func james
echo “${0}, ${1}”
```

```
james@x110e$ ./param.sh different
./param.sh, james
./param.sh, different
```

Take a look at **functions.sh** in the **Lab04** directory for more examples.

QUESTIONS

3. Write a function **(Task3.sh)** which takes a directory name as a parameter and:
    a) Checks if the directory already exists
    b) Create it if it does not exist
    c) Moves into the directory
4. Write a function **(Task4.sh)** which, for every file name passed as a parameter, prints if it is a file, directory or does not exist. Hint: You should be able to reuse the script you wrote in Lab3, task 5 for much of this script.

PHYSICS 2T C Programming Under Linux

Linux Lab 5 Regular Expressions and GNU Make

INTRODUCTION

This lab is all about Git and GNU Make. Although previous labs have guided you through individual steps and offered advice, this lab comes with no tutorial and you are expected to do your own research to complete the questions.

All of the topics covered in this lab are extremely well documented in lectures, online, in books and on the internet. It is your responsibility to find these yourself.

As with previous labs, if you get stuck don't hesitate to seek help from lab demonstrators, lectures and other students. As before please supply your answers in and answer.txt file.

QUESTIONS

1. You's a web browser to navigate to: https://bitbucket.org/glaphysp2t/lab5-example

   a) What command would you use to clone the repository to your account on brutha? Clone the repository so you can work on it. (**NOTE:** you will need to use the https://bitbucket.org/glaphysp2t/lab5-example.git URL to clone the repository, DO NOT use the ssh URL).

   b) What command would give you a list of commit messages all on oneline? What is the message associated with the hash 2a65f62?

   c) Build the code using the provided makefile, run it with the make test command. After running the program edit the README.md file to include a description of what the code does.

   d) What is the output of git status?

   e) What commands would you use to commit your modified file to you local repository (Hint: you'll need to add the file to the commit first!), with an appropriate commit message. Do this and copy in the results of git log.

   f) Create a new branch called myfeature (note the command you used).

   g) Change into that branch (what command did you use?). What is the output of:

      git branch --list

   h) Make any modification to the code, commit your changes and note the output of git log.

2. Give short descriptions of the following automatic make variables:

   a) **`$@`**

   b) **`$^`**

   c) **`$<`**

   d) **`$?`**

3. Examine the code in the **`simulation`** directory, spend a few minutes getting to know it before continuing. **Make sure you are able to compile the program for the command line before continuing (HINT: compile main.c and simulation.c and link together).**

   a) Write targets for **`main.o`** and **`simulation.o`** in a **`makefile`**. You must make use of at least some of the automatic variables provided by **`make`**.

   b) Write a “Phony” target called **`clean`** which removes any object files generated by **`gcc`**.

   c) Write the body of a function (**`kinetic_energy)`** in **`simulation.c`** which returns the kinetic energy of an object given its mass and relativistic velocity (a skeleton is provided for you). Use this function and add a third column of data to the lookup table.

   d) Use **`make`** to compile your edited source code. Run the program to make sure it works then redirect the output to **`sim.data`**.

   e) Display the results you have produced using **`gnuplot`** using the provided **`graph.plot`** file. You can do this by running the command:

      gnuplot graph.plot

   f) In the simulation **`makefile`**, what library needed to be linked and why? Which C function used in the lookup table requires it?

   g) How would you enable debugging symbols and compiler optimisations in **`gcc`**? Edit the **`makefile`** and enable them.

PHYSICS 2T C Programming Under Linux

Linux Lab 6 GDB

INTRODUCTION

This is Lab session is about familiarising yourself with GDB, the GNU debugger and learning how to use it in developing C code. All the information you need to work with GDB can be found in the lectures which should be used as a reference.

As with previous labs, if you get stuck don't hesitate to seek help from lab demonstrators, lectures and other students. As before please supply your answers in and answer.txt file.

QUESTIONS

1. Download the contents of the code directory found in the Linux Lab 06 directory on Moodle, or obtain it from the supplied zip file. In this directory is a modified version of the code you have seen in previous Lectures along with a makefile.

   a) Examine the Makefile, what is missing from it that is required to begin debugging our program? **Hint:** something needs to be passed to the compiler to generate debug information. Modify the Makefile and run **make debug**.

   b) Try running the program from the command line, what happens? How would we run the debugger? Do so now.

   c) How would you set a breakpoint at the beginning of the code? Set that breakpoint and begin running the code with the run command.

   d) Which line of code and which function is now highlighted/printed after execution has halted?

   e) How would you advance the execution of the code by one instruction so that you entered into the function you identified above? Do that now.

   f) Continue to run the code using **c** or **continue**. The program will now break, what is the error message and on which line did it occur?

   g) How would you print out the contents of the **dataptr** variable, and what is its contents?

   h) How would you print out the contents of the memory location it points to, what happens when you do so? From reading the error message what do you think the problem with this section of the code is?

   i) Quit out of gdb using the **quit** command. Change line 6 in the **util.c** file to read:

int *dataptr = data;

Re-compile the code and try running from the command line. What happens now? **Hint:** you may need to use **ctrl-C** to halt execution of you code.

j) Run your program in the debugger again. Where would you set a breakpoint to continue debugging the code?

k) In this case set a break point at **line 12**, run the program and print the contents of the data array. What are the contents of the data array?

l) How would you set a watch point on the index variable **i?** Do that now.

m) Continue running the code, using either the **next**, **step** or **continue** commands. What happens?

n) Looking at the output observed in the above step and the **line 13** of the **main.c** file, what is the problem with the execution of the code?

o) How would you correct the code on **line 13**? Make the correction, recompile, and run the code. **Congratulations, your code should now successfully run!**