Uppsala University

Introduction to Linux & UPPMAX

Martin Dahlö
martin.dahlo@scilifelab.uu.se

Marcus Holm
marcus.holm@it.uu.se

January 17, 2019

Contents

1 Linux Introduction

Important: If you are not able to connect to the UPPMAX servers there are a few parts of this lab that you will not be able to participate in. You will also not be able to acquire a copy of the laboratory files from the project folder on the UPPMAX servers. These are, however, available for download through this link: labs.tar.gz. A few of the more specific instructions are designed for use on the UPPMAX servers. These will work for you as well, but they might yield different results when performed on your local machine.

Note: in syntax examples, the dollar sign ($) is not to be printed. The dollar sign is usually an indicator that the text following it should be typed in a terminal window.

1.1 Connecting to UPPMAX

The first step of this lab is to open a SSH connection to UPPMAX. You will need a SSH program to do this:

On Linux: it is included by default, named **Terminal**.

On OSX: it is included by default, named **Terminal**.

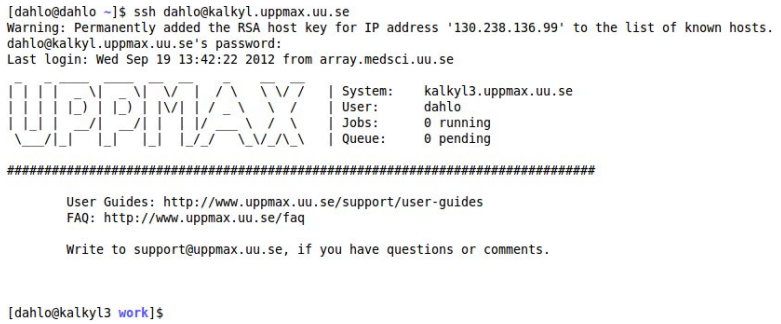
On Windows: Google MobaXTerm and download it.

Fire up the available SSH program and enter the following (replace **username** with your UPPMAX user name). -X means that X-forwarding is activated on the connection, which means graphical data can be transmitted if a program requests it, i.e. programs can use a graphical user interface (GUI) if they want to.

```
$ ssh -X username@rackham.uppmax.uu.se
```

and give your password when prompted. As you type, nothing will show on screen. No stars, no dots. It is supposed to be that way. Just type the password and press enter, it will be fine.

Now your screen should look something like this:



1.2 Moving and Looking Around

It is good to know how to move around in the file system. I'm sure you all have experienced this using a graphical user interface (**GUI**) before, Windows Explorer in Windows and Finder in OSX. Using the command line can be confusing at first, but the more your do it, the easier it gets.

When you connect to UPPMAX, you will start out in your home folder. The absolute path to your home folder is usually /home/<username>.

Start with looking at what you have in your home folder. The command for this is ls, and it stand for **LiSt** (list).

```
$ ls -l
```

You should see something similar to:

```
[marcusl@rackham1 ~]$ ls -l
total 8210
drwxr-xr-x  2 marcusl marcusl      2048 Jan  9  2008 bin
drwxr-x---  3 marcusl marcusl      2048 Jan 18  2017 glob
drwx--S---  2 marcusl marcusl      2048 May  4  2010 private
drwxrwxr-x  3 marcusl marcusl      2048 Apr  1 13:24 tmp
drwxrwxr-x  6 marcusl marcusl      2048 Jan 12 19:57 uppmax-intro
[marcusl@rackham1 ~]$
```

Your home directory is the place where you keep personal files, programs, and other tools that aren't associated with a particular project. Your files

are backed up, and you are limited to 32 GB of data here. Most actual work-related data must be stored in a project directory.
You may notice that you have a "glob" directory in your home. This exists for historical reasons and used to be a faster, larger storage area without backup, but is currently just a normal directory with no special functions or properties.
As seen in the lecture, the command for moving around is **cd**. The command stands for **C**hange **D**irectory and does exactly that. It is the equivalent of double clicking a folder in a GUI. To enter your glob folder (for instance), simply type

```
$ cd glob
```

We can see that this is a relative path, since it does not start with a '/' . That means that this command will only work when you are standing in your home folder. If you are standing somewhere else and say that you want to enter a folder named glob, the computer might tell you that there is no folder named glob where you are located at the moment. The absolute path to your glob folder would be **/home/<username>/glob**.

It is the exact same thing as if you are using a GUI. If you are standing on your desktop, you can double click a folder which is located on your desktop. But if you are standing in another folder, you can't double click on the same folder, because it is just not located in that folder. You have to move to your desktop first, and then double click it.

If you look around in your glob folder, you probably only have a folder called **private**. This is how everyone's glob folder looks before you start putting files there.

Next, let's move to the course's project folder. A project folder is like a home folder, but it is shared between all the members of the project. It is the common file area in the project, and the place where you will store your raw data and important analysis results. This course's project id is **g2018034**, so the path to the project folder is **/proj/g2018034** .

NOTE: Remember to tab-complete to avoid typos and too much writing.

```
$ cd /proj/g2018034
```

Look at what is in the folder:

```
[rackham2:/proj/g2017016]$ ls -lh
total 16K
drwxrwsr-x 4 marcusl g2017016 4.0K Aug  8 09:26 completed
drwxrwsr-x 5 marcusl g2017016 4.0K Aug  8 14:37 labs
drwxrwsr-x 3 root    g2017016 4.0K May 19 10:40 nobackup
drwxrws--- 2 root    g2017016 4.0K May 19 10:40 private
```

Now we have practised moving around and looking at what we have in folders. The next step will show you how to do the same thing, but without the moving around part. If we want to look at what we have in our home folder, while standing in the course's project folder, we type **ls /home/<username>/** and remember to substitute <username>with your own user name.

```
$ cd /home/<username>
```

Since most programmers are lazy (efficient), there is a shortcut to your home folder so that you don't have to write it all the time. If you write ∼/ it means the same as if you would write **/home/<username>/**

Let's move back to our home folder before going to the next step. There are at least 3 different ways of getting to your home folder, and they are all equally good:

```
$ cd /home/<username>
```

or:

```
$ cd ~
```

or simply:

```
$ cd
```

1.3 Copying files needed for laboratory

To be able to do parts of this lab, you will need some files. To avoid all the course participants editing the same file all at once, undoing each other's edits, each participant will get their own copy of the needed files.

The files are located in the folder **/proj/g2018034/labs/linux_tutorial**

or if you are not on UPPMAX at the moment they can be downloaded here: files.tar.gz (instruction on how to download further down).

For structures sake, first create a folder named **uppmax-intro** in your home directory, and a folder called **linux_tutorial** inside that folder, where you can put all your lab files.

This can be done in 2 ways:

```
$ mkdir ~/uppmax-intro/
```

```
$ mkdir ~/uppmax-intro/linux_tutorial
```

or

```
$ mkdir -p ~/uppmax-intro/linux_tutorial
```

The reason for this is that Linux will not like it if you try to create the folder linux_tutorial inside a folder (uppmax-intro) that does not exist yet. Then you have the choice to either first create uppmax-intro (the first way), or to tell Linux to create it for you by giving it the -p option (the second way).

Next, copy the lab files to this folder. -r means recursively, which means all the files including sub-folders of the source folder. Without it, only files directly in the source folder would be copied, NOT sub-folders and files in sub-folders.

NOTE: Remember to tab-complete to avoid typos and too much writing.

```
Ex: cp -r <source folder> <destination folder>
```

```
$ cp -r /proj/g2018034/labs/linux_tutorial/*
~/uppmax-intro/linux_tutorial
```

(Just write this command on one line with a space where the line break is.)

If you are unable to copy the files on UPPMAX, you can download the files instead of copying them. This is done with the command **wget** (web get). It works kind of the same way as the cp command, but you give it an source URL instead of a source file, and you specify the destination by giving it a prefix, a path that will be appended in front on the file name when it's downloaded. I.e. if you want to download the file http://somewhere.com/my.file and you give it the prefix ∼/analysis/, the downloaded file will be saved as ∼/analysis/my.file

```
Ex: wget -P <destination prefix> <source URL>
```

```
$ wget -P ~/uppmax-intro/linux_tutorial
http://user.it.uu.se/~marlu734/files.tar.gz
```

(Just write this command on one line with a space where the line break is.)

1.4 Unpack Files

Go to the folder you just copied and see what is in it.

NOTE: Remember to tab-complete to avoid typos and too much writing.

```
$ cd ~/uppmax-intro/linux_tutorial
```

```
$ ls -l
```

tar.gz is a file ending give to compressed files, something you will encounter quite often. Compression decreases the size of the files which is good when downloading, and it can take thousands of files and compress them all into a single compressed file. This is both convenient for the person downloading and speeds up the transfer more than you would think.

To unpack the **files.tar.gz** file use the following line while standing in the newly copied linux_tutorial folder.

```
$ tar -xzvf files.tar.gz
```

The command will always be the same for all tar.gz files you want to unpack. -xzvf means e**X**tract from a **Z**ipped file, **V**erbose (prints the name of the file being unpacked), from the specified **F**ile (f must always be the last of the letters).
Look in the folder again and see what we just unpacked:

```
[marcusl@rackham1 linux_tutorial]$ ls -l
total 448
drwxrwxr-x 2 marcusl marcusl   2048 Sep 19  2012 a_strange_name
drwxrwxr-x 2 marcusl marcusl   2048 Sep 19  2012 backed_up_proj_folder
drwxrwxr-x 2 marcusl marcusl   2048 Sep 19  2012 external_hdd
-rwxr-xr-x 1 marcusl marcusl  17198 Oct  9  2015 files.tar.gz
drwxrwxr-x 2 marcusl marcusl   2048 Sep 19  2012 important_results
drwxrwxr-x 2 marcusl marcusl 129024 Sep 19  2012 many_files
drwxrwxr-x 2 marcusl marcusl   2048 Sep 19  2012 old_project
-rwxr-xr-x 1 marcusl marcusl      0 Oct  9  2015 other_file.old
drwxrwxr-x 2 marcusl marcusl   2048 Sep 19  2012 part_1
drwxrwxr-x 2 marcusl marcusl   2048 Sep 19  2012 part_2
drwxrwxr-x 2 marcusl marcusl   2048 Jan 28  2012 this_has_a_file
drwxrwxr-x 2 marcusl marcusl   2048 Jan 28  2012 this_is_empty
-rwxrwxr-x 1 marcusl marcusl      0 Sep 19  2012 useless_file
[marcusl@rackham1 linux_tutorial]$
```

1.5 Copying and Moving Files

Let's move some files. Moving files might be one of the more common things you do, after cd and ls. You might want to organise your files in a better

way, or move important result files to the project folder, who knows?

We will start with moving our important result to a backed up folder. When months of analysis is done, that last thing you want is to lose your files. Typically this would mean that you move the final results to your project folder.

In this example we want to move the result file only, located in the folder important_results, to our fake project folder, called backed_up_proj_folder. The syntax for the move command is:

$ mv <source><destination>

First, take a look inside the important_results folder:

```
[marcusl@rackham1 linux_tutorial]$ ls -l important_results/
total 0
-rwxrwxr-x 1 marcusl marcusl 0 Sep 19  2012 dna_data_analysis_result_file_that_is_imp
-rwxrwxr-x 1 marcusl marcusl 0 Sep 19  2012 temp_file-1
-rwxrwxr-x 1 marcusl marcusl 0 Sep 19  2012 temp_file-2
[marcusl@rackham1 linux_tutorial]$
```

You see that there are some unimportant temporary files that you have no interest in. Just to demonstrate the move command, I will show how you would move one of these temporary files to your backed up project folder:

```
$ mv important_results/temp_file-1 backed_up_proj_folder/
```

Now do the same, but move the important DNA data file!

Look in the backed up project folder to make sure you moved the file correctly.

```
[marcusl@rackham1 linux_tutorial]$ ls -l backed_up_proj_folder/
total 0
-rwxrwxr-x 1 marcusl marcusl 0 Sep 19  2012 dna_data_analysis_result_file_that_is_imp
-rwxrwxr-x 1 marcusl marcusl 0 Sep 19  2012 last_years_data
-rwxrwxr-x 1 marcusl marcusl 0 Sep 19  2012 temp_file-1
```

Another use for the move command is to rename things. When you think of it, renaming is just a special case of moving. You move the file to a location and give the file a new name in the process. The location you move the file to can of very well be the same folder the file already is in. To give this a try, we will rename the folder **a_strange_name** to a better name.

```
$ mv a_strange_name a_better_name
```

Look around to see that the name change worked.

```
[marcusl@rackham1 linux_tutorial]$ ls -l
total 448
drwxrwxr-x 2 marcusl marcusl   2048 Sep 19  2012 a_better_name
drwxrwxr-x 2 marcusl marcusl   2048 Jun 22 13:27 backed_up_proj_folder
drwxrwxr-x 2 marcusl marcusl   2048 Sep 19  2012 external_hdd
-rwxr-xr-x 1 marcusl marcusl  17198 Oct  9  2015 files.tar.gz
drwxrwxr-x 2 marcusl marcusl   2048 Jun 22 13:27 important_results
drwxrwxr-x 2 marcusl marcusl 129024 Sep 19  2012 many_files
drwxrwxr-x 2 marcusl marcusl   2048 Sep 19  2012 old_project
-rwxr-xr-x 1 marcusl marcusl      0 Oct  9  2015 other_file.old
drwxrwxr-x 2 marcusl marcusl   2048 Sep 19  2012 part_1
drwxrwxr-x 2 marcusl marcusl   2048 Sep 19  2012 part_2
drwxrwxr-x 2 marcusl marcusl   2048 Jan 28  2012 this_has_a_file
drwxrwxr-x 2 marcusl marcusl   2048 Jan 28  2012 this_is_empty
-rwxrwxr-x 1 marcusl marcusl      0 Sep 19  2012 useless_file
```

Sometimes you don't want to move things, you want to copy them. Moving a file will remove the original file, whereas copying the file will leave the original untouched. An example when you want to do this could be that you want to give a copy of a file to a friend. Imagine that you have a external hard drive that you want to place the file on. The file you want to give to your friend is data from last years project, which is located in your backed_up project folder, **backed_up_proj_folder/last_years_data**

As with the move command, the syntax is

$ cp <source><destination>

```
$ cp backed_up_proj_folder/last_years_data external_hdd/
```

Take a look in the external_hdd to make sure the file got copied.

```
[marcusl@rackham1 linux_tutorial]$ ls -l external_hdd/
total 0
-rwxrwxr-x 1 marcusl marcusl 0 Jun 22 13:32 last_years_data
[marcusl@rackham1 linux_tutorial]$
```

1.6 Deleting Files

Sometimes you will delete files. Usually this is when you know that the file or files are useless to you, and they only take up space on your hard drive or UPPMAX account. To delete a file, we use the **R**e**M**ove command, **rm**. Syntax:

```
$ rm <file to remove>
```

If you want, you can also specify multiple files a once, as many as you want!

```
$ rm <file to remove> <file to remove> <file to remove>
```

IMPORTANT: There is no trash bin in Linux. If you delete a file, it is gone. So be careful when deleting stuff..

Try it out by deleting the useless file in the folder you are standing in. First, look around in the folder to see the file.

```
[marcusl@rackham1 linux_tutorial]$ ls -l
total 448
drwxrwxr-x 2 marcusl marcusl   2048 Sep 19  2012 a_better_name
drwxrwxr-x 2 marcusl marcusl   2048 Jun 22 13:27 backed_up_proj_folder
drwxrwxr-x 2 marcusl marcusl   2048 Jun 22 13:32 external_hdd
-rwxr-xr-x 1 marcusl marcusl  17198 Oct  9  2015 files.tar.gz
drwxrwxr-x 2 marcusl marcusl   2048 Jun 22 13:27 important_results
drwxrwxr-x 2 marcusl marcusl 129024 Sep 19  2012 many_files
drwxrwxr-x 2 marcusl marcusl   2048 Sep 19  2012 old_project
```

```
-rwxr-xr-x 1 marcusl marcusl      0 Oct  9  2015 other_file.old
drwxrwxr-x 2 marcusl marcusl   2048 Sep 19  2012 part_1
drwxrwxr-x 2 marcusl marcusl   2048 Sep 19  2012 part_2
drwxrwxr-x 2 marcusl marcusl   2048 Jan 28  2012 this_has_a_file
drwxrwxr-x 2 marcusl marcusl   2048 Jan 28  2012 this_is_empty
-rwxrwxr-x 1 marcusl marcusl      0 Sep 19  2012 useless_file
[marcusl@rackham1 linux_tutorial]$
```

Now remove it:

```
$ rm useless_file
```

Its not only files you can remove. Folders can be removed too. There is even a special command for removing folders, **rmdir**. They work similar to **rm**, except that they can't remove files. There are two folders, **this_is_empty** and **this_has_a_file**, that we now will delete.

```
$ rmdir this_is_empty
```

```
$ rmdir this_has_a_file
```

If you look inside **this_has_a_file**

```
[marcusl@rackham1 linux_tutorial]$ ls -l this_has_a_file/
total 0
-rwxrwxr-x 1 marcusl marcusl 0 Jan 28  2012 file
[marcusl@rackham1 linux_tutorial]$
```

you will see there is a file in there! Only directories that are completely empty can be deleted using rmdir. To be able to delete **this_has_a_file**, either delete the file manually and then remove the folder

```
$ rm this_has_a_file/file
$ rmdir this_has_a_file
```

or delete the directory recursively, which will remove this_has_a_file **and** everything inside:

```
$ rm -r this_has_a_file
```

1.7 Reading files

So what happens if you give your files bad names like 'file1' or 'results'? You take a break in a project and return to it 4 months later, and all those short names you gave your files doesn't tell you at all what the files actually contain. Of course, this should never happen, because you should **ALWAYS** name your files so that you definitely know what they contain. But lets say it did happen. Then the only way out is to look at the contents of the files and try to figure out if it is the file you are looking for.

Now, we are looking for that really good script we wrote a couple of months ago in that other project. Look in the project's folder, **old_project**, and find the script.

```
[marcusl@rackham1 linux_tutorial]$ ls -l old_project/
total 96
-rwxrwxr-x 1 marcusl marcusl 39904 Sep 19  2012 a
-rwxrwxr-x 1 marcusl marcusl     0 Sep 19  2012 stuff_1
-rwxrwxr-x 1 marcusl marcusl  1008 Sep 19  2012 the_best
[marcusl@rackham1 linux_tutorial]$
```

Not so easy with those names.. We will have to use less to look at the files and figure out which is which. Syntax for less:

$ less <filename>

Have a look at **the_best**, that must be our script, right?

```
$ less old_project/the_best
```

(press q to close it down, use arrows to scroll up/down)

I guess not. Carrot cakes might be the bomb, but they won't solve bioinformatic problems. Have a look at the file **a** instead.

That's more like it!

Now imagine that you had 100s of files with weird names, and you really needed to find it.. Lesson learned: name your files so that you know what

they are! And don't be afraid to create folders to organise files.

Another thing to think about when opening files in Linux is which program should you open the file in? The programs we covered during the lectures are **nano** and **less**. The main difference between these programs in that less **can't edit files**, only view them. Another difference is that less **doesn't load the whole file** into the RAM memory when opening it. So, why care about how the program works? I'll show you why. This time we will be opening a larger file, located in the course's project folder. It's 65 megabytes, so it is a tiny file compared with bio-data. Normal sequencing files can easily be 100-1000 times larger than this.

First, open the file with **nano**. Syntax:

$ nano <filename>

```
$ nano /proj/g2018034/labs/linux_additional-files/large_file
```

(press ctrl+x to close it down, user arrows to scroll up/down)

Is the file loaded yet? Now take that waiting time and multiply it with 100-1000. Now open the file with **less**. Notice the difference?

Head and **tail** works the same was as **less** in this regard. They don't load the whole file into RAM, they just take what they need.

To view the first rows of the large file, use **head**. Syntax:

$ head <filename>

```
$ head /proj/g2018034/labs/linux_additional-files/large_file
```

The same syntax for viewing the last rows of a file with **tail**:

$ tail <filename>

```
$ tail /proj/g2018034/labs/linux_additional-files/large_file
```

$ tail -n <number of rows to view><filename>

```
$ tail -n 23 /proj/g2018034/labs/linux_additional-files/large_file
```

1.8 Wildcards

A lot of the time, you have many files. So many that it would take you a day just to type all their names. This is where wildcards saves the day. The **wildcard** symbol in Linux is the star sign, * , and it means literally **anything**. Say that you want to move all the files which has names starting with sample_1_ and the rest of the name doesn't matter. You want all the files belonging to sample_1. Then you could use the wildcard to represent the rest of the name:

(don't run this command, it's just an example)

```
$ mv  sample_1_* my_other_folder
```

We can try it out on the example files I have prepared. There are two folder called **part_1** and **part_2**. We want to collect all the .txt files from both these folders in one of the folders. Look around in both the folders to see what they contain.

```
[marcusl@rackham1 linux_tutorial]$ ls part_1
file_1.txt  file_2.txt
[marcusl@rackham1 linux_tutorial]$ ls part_2
file_3.txt  file_4.txt  garbage.tmp  incomplete_datasets.dat
[marcusl@rackham1 linux_tutorial]$
```

We see that **part_1** only contains .txt files, and that **part_2** contains some other files as well. The best option seem to be to move all .txt files from **part_2** info **part_1**.

```
$ mv part_2/*.txt part_1/
```

The wildcard works with most, if not all, Linux commands. We can try using wildcards with ls. Look in the folder **many_files**. Yes, there are ∼1000

.docx files in there. But not only .docx files.. There are a couple of **.txt** files in there as well. Find out which numbers they have. Try to figure out the solution on your own. I have written the answer below, with white text. Mark the text with the mouse, or press cmd+a to mark everything on the page to see the answer.

1.9 Utility Commands

Ok, the last 2 commands now. **Top** and **man**.

Top can be useful when you want to look at which programs are being run on the computer, and how hard the computer is working. Type **top** and have a look.

```
$ top
```

```
top - 19:54:16 up 77 days,  6:52, 31 users,  load average: 1.72, 1.44, 1.46
Tasks: 674 total,   2 running, 667 sleeping,   1 stopped,   4 zombie
Cpu(s):  6.8%us,  1.6%sy,  0.0%ni, 90.8%id,  0.0%wa,  0.0%hi,  0.7%si,  0.0%st
Mem:  24597168k total, 24408312k used,   188856k free,   170436k buffers
Swap: 67108848k total,  6649208k used, 60459640k free, 11330804k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
26006 perk      20   0 63348 8372 2772 R 78.4  0.0  46:31.42 ssh
 6824 perk      20   0 63316 8348 2772 S 26.1  0.0  49:17.99 ssh
 8793 perk      20   0 13396 2460  412 S 18.5  0.0  12:10.10 rsync
23142 perk      20   0 13388 2484  428 S  6.3  0.0  13:09.78 rsync
20411 perk      20   0  535m  80m 1864 S  1.7  0.3   7:35.99 celeryd
20410 perk      20   0  526m  69m 1504 S  1.3  0.3   6:42.48 celeryd
 2404 root      20   0     0    0    0 S  1.0  0.0 557:00.12 kpanfs_thpool
 2398 root      20   0     0    0    0 S  0.7  0.0 556:29.31 kpanfs_thpool
 2399 root      20   0     0    0    0 S  0.7  0.0 556:48.43 kpanfs_thpool
 2402 root      20   0     0    0    0 S  0.7  0.0 556:55.02 kpanfs_thpool
 2403 root      20   0     0    0    0 S  0.7  0.0 556:19.43 kpanfs_thpool
 2405 root      20   0     0    0    0 S  0.7  0.0 556:44.15 kpanfs_thpool
 2406 root      20   0     0    0    0 S  0.7  0.0 556:20.34 kpanfs_thpool
 2408 root      20   0     0    0    0 S  0.7  0.0 556:58.39 kpanfs_thpool
 2410 root      20   0     0    0    0 S  0.7  0.0 556:40.44 kpanfs_thpool
 2412 root      20   0     0    0    0 S  0.7  0.0 556:31.42 kpanfs_thpool
 2413 root      20   0     0    0    0 S  0.7  0.0 556:31.38 kpanfs_thpool
10391 perun     20   0 13044 5024 1356 S  0.7  0.0   0:00.49 bash
19651 dahlo     20   0 13740 1644  884 R  0.7  0.0   0:00.33 top
```

(press q to close it down)

Each row in top corresponds to one program running on the computer, and the column describe various information about the program. **The right-most column** shows you which program the row is about.

There are mainly 2 things that are interesting when looking in top. The first is how much **cpu** each program is using. I have marked it with **blue** in the picture. If you are doing calculations, which is what bioinformatics is mostly about, the cpu usage should be high. The numbers in the column is how many percent of a core the program is running. If you have a computer with 8 cores, like the UPPMAX computers, you can have 8 programs using 100% of a core each running at the same time without anything slowing down. As soon as you start a 9th program, it will have to share a core with another program and those 2 programs will run at half-speed since a core can only work that fast. In the image, one instance of the program ssh is using 78.4% of a core.

The areas marked with **red** is describing how much **memory** is being used. The area in the top describes the overall memory usage. Total tells you how much memory the computer has, used tells you how much of the memory is being used at the moment, and free tells you how much memory is free at the moment. Total = Used + Free

The **red column** tells you how much memory each program uses. The numbers mean how many percent of the total memory a program uses. In the image, the program celeryd is using 0.3% of the total memory.

A warning sign you can look for in top is when you are running an analysis which seems to take forever to complete, and you see that there is almost no cpu usage on the computer. That means that the computer is not doing any calculation, which could be bad. If you look at the memory usage at the same time, and see that it's maxed out, you should abort the analysis because it has stalled from lack of memory.

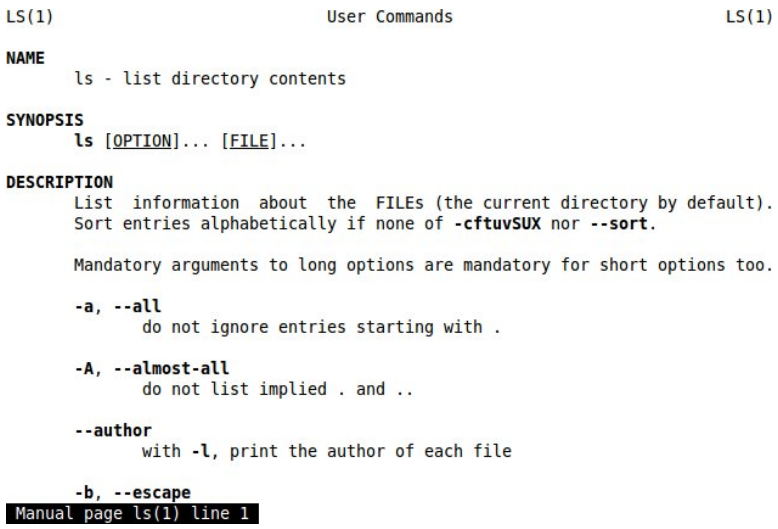
When the memory runs out, the computer more or less stops. Since it can't fit everything into the RAM memory, it will start using the hard drive to store the things it can't fit in the RAM. Since the hard drive is $\sim$1000 times slower than the RAM, things will be going in slow-motion. The solution to that could be to either change the settings of the program you are running to decrease the memory usage (if the program has that functionality), or just get a computer with more memory.

Ok, the last command in this part of the course, and arguably one of the most important: the command that shows you how to use other commands. You won't ever remember all of the flags and options to all the commands.

You might remember ls, but was it -l or -a you should use to see hidden files? You might wish that there was a manual for these things. Good news everyone, there is a manual! To get all the nitty-gritty details about ls, you use the **man** command. Syntax:

$ man <command you want to look at>

```
$ man ls
```



This will open a "less" window (remember, q to close it down, arrows to scroll) with the manual page about ls. Here you will be able to read everything about ls. You'll see which flag does what (-a is to show the hidden files, which in linux are files with a name starting with a dot .), which syntax the program has, etc. If you are unsure about how to use a command, look it up in man.

The man pages can be a bit tricky to understand at first, but you get used to it with time. If it is still unclear after looking in man, try searching for it on the internet. You are bound to find someone with the exact same question as you, that has already asked on a forum, and gotten a good answer. 5 years ago.

2 Advanced Linux

For the purposes of this course, this chapter is optional. Go for it if you have time left in the morning session, otherwise skip it.

NOTE: in syntax examples, the dollar sign ($) is not to be typed. The dollar sign is usually an indicator that the text following it should be typed in a terminal window.

2.1 Ownership & Permissions

As Linux can be a multi-user environment it is important that files and directories can have different owners and permissions to keep people from editing or executing your files.

2.1.1 Owners

The permissions are defined seperately for **users**, **groups** and **others**.

The **user** is the username of the person who owns the file. By default the user who creates a file will become its owner.

The **group** is a group of users that co-own the file. They will all have the same permissions to the file. This is useful in any project where a group of people are working together.

The **others** is quite simply everyone else's permissions.

2.1.2 Permissions

There are four permissions that a file or directory can have. Note the one character designations/flags, **r**,**w**,**x** and **-**.

In all cases, if the file or directory has the flag it means that it is enabled.

Read: r

File: Whether the file can be opened and read.

Directory: Whether the contents of the directory can be listed.

Write: w

File: Whether the file can be modified. Note that for renaming or deleting a file you need additional directory permissions.

Directory: Whether the files in the directory can be renamed or deleted.

Execute: x

File: Whether the file can be executed as a program or shell script.

Directory: Whether the directory can be entered through "cd".

SetGID: s (with execute) or S (no execute)

File: N/A.

Directory: When another user creates a file under a SetGID directory, the new file will have its group set as the group of the parent directory's owner instead of the group of the user creating the file. *Don't worry too much about this one, I only put this here for completeness.*

No permissions: -

2.1.3 Interpreting the permissions of files and directories

Start your terminal, log onto UPPMAX.
Make an empty directory we can work in and make a file.

```
$ cd ~/glob/uppmax-intro
$ mkdir advlinux
$ cd advlinux
$ touch  filename
$ ls -lh
total 0
```

```
-rw-rw-r-- 1 marcusl marcusl 0 Jun 27 14:32 filename
```

(-lh means long and human readable, displaying more information about the files or directories in an humanly understandable format)

The first segment, -rw-r–r–, describes the ownerships and permissions of our newly created file. The very first character, in this case "-", shows the files type. It can be any of these:

d = directory
- = regular file
l = symbolic link
s = Unix domain socket
p = named pipe
c = character device file
b = block device file

As expected the file we have just created is a regular file. Ignore the types other than directory, regular and symbolic link as they are outside the scope of this course.

The next nine characters, in our case rw-r–r–, can be divided into three groups consisting of three characters in order from left to right. In our case rw- r– and r–. The first group designates the **users** permissions, the second the **groups** permissions and the third the **others** permissions. As you may have guessed the within group permissions are ordered, the first always designates read permissions, the second write and the third executability.

This translates our files permissions to say this:

```
-rw-r--r--
```

"It is a regular file. The user has read & write permission, but not execute. The group has read permission but not write and execute. Everyone else (other), have read permission but not write and execute."

As another example, let's create a directory.

```
$ mkdir directoryname
```

```
$ ls -lh
total 4.0K
drwxrwxr-x 2 marcusl marcusl 4.0K Jun 27 14:33 directoryname
-rw-rw-r-- 1 marcusl marcusl    0 Jun 27 14:32 filename
```

As you can see the first character correctly identifies it as **d**, a directory, and all user groups have **x**, execute permissions, to enter the directory by default.

2.1.4 Editing Ownership & Permissions

The command to set file permission is "chmod" which means "CHange MODe". Only the owner can set file permissions.

1. First you decide which group you want to set permissions for. User, **u**, group, **g**, other, **o**, or all three, **a**.
2. Next you either add, +, remove, **-**, or wipe out previous and add new, =, permissions.
3. Then you specify the kind of permission: **r**, **w**, **x**, or **-**.

Let's revisit our example file and directory to test this.

```
$ ls -lh
total 4.0K
drwxrwxr-x 2 marcusl marcusl 4.0K Jun 27 14:33 directoryname
-rw-rw-r-- 1 marcusl marcusl    0 Jun 27 14:32 filename
$ chmod a=x filename
$ ls -lh
total 4.0K
drwxrwxr-x 2 marcusl marcusl 4.0K Jun 27 14:33 directoryname
---x--x--x 1 marcusl marcusl    0 Jun 27 14:32 filename
```

As you can see this affected all three, **a**, it wiped the previous permissions, =, and added an executable permission, **x**, to all three groups.

Try some others both on the file and directory to get the hang of it.

```
$ chmod g+r filename
$ chmod u-x filename
$ chmod ug=rx filename
$ chmod a=- filename
$ chmod a+w directoryname
```

2.1.5 Exercise

In no more than two commands, get the file permissions from

```
----------
```

to

```
-rw-rw--wx
```

Notice also that we here gave everyone writing permission to the file, that means that ANYONE can write to the file. Not very safe.

2.2 Symbolic links - Files

Much like a windows user has a shortcut on his desktop to WorldOfWarcraft.exe, being able to create links to files or directories is good to know in Linux.

An important thing to remember about symbolic links is that they are not updated, so if you or someone else moves or removes the original file/directory the link will stop working.

Let's remove our old file and directory. (Notice: `rm -r *` removes *all* folder and subfolders from where you are standing. Make sure you are standing in our advlinux/ directory.)

```
$ rm -r *
```

Now that the directory is empty let's make a new folder and a new file in that folder.

```
$ mkdir stuff
$ touch stuff/linkfile
```

Let's put some information into the file, just put some text, anything, like "slartibartfast" or something.

```
$ nano stuff/linkfile
```

or, depending on preference,

```
$ gedit stuff/linkfile &
```

Now let's create a link to this file in our original folder. `ln` stands for link and `-s` makes it symbolic. The other options are not in the scope of this course.

The usage format is `ln <target> <link_name>`. If you don't supply a link name then it name the link after the target's filename.

```
$ ln -s stuff/linkfile linky
$ ls -l
total 4
lrwxrwxrwx 1 marcusl marcusl   14 Jun 27 14:41 linky -> stuff/linkfile
drwxrwxr-x 2 marcusl marcusl 4096 Jun 27 14:41 stuff
```

Notice that we see the type of the file is **l**, for symbolic link, and that we have a pointer after the links name for where the link goes, -> stuff/linkfile.

2.2.1 Assignment

I want you to change the information in the file using the link file, then check the information in the original file. Did editing the information in the link change the information in the original?

Change the information using the original file, then checking the link will result in the same result. The files are connected.

Now move or delete the original file.

What happens to the link?

What information is there now if you open the link?

Now create a new file in stuff/ with exactly the same name that your link file is pointing to with new information in it.

Is the link re-activated to the new file? (You can investigate this by seeing if the new information is seen when opening the link file).

2.3 Symbolic links - Directories

Now let's create a link to a directory.

Let's clean our workspace.

```
$ rm -r *
```

Now let's create a directory three, arbitrarily, directories away.

```
$ mkdir -p one/two/three
```

Now let's enter the directory and create some files there.

```
$ cd one/two/three
$ touch a b c d e
$ ls -lh
total 0
-rw-rw-r-- 1 marcusl marcusl 0 Jun 27 14:43 a
-rw-rw-r-- 1 marcusl marcusl 0 Jun 27 14:43 b
-rw-rw-r-- 1 marcusl marcusl 0 Jun 27 14:43 c
-rw-rw-r-- 1 marcusl marcusl 0 Jun 27 14:43 d
-rw-rw-r-- 1 marcusl marcusl 0 Jun 27 14:43 e
```

Return to our starting folder and create a symbolic link to folder three

```
$ cd ../../..
$ ln -s one/two/three
$ ls -lh
total 4.0K
drwxrwxr-x 3 marcusl marcusl 4.0K Jun 27 14:43 one
lrwxrwxrwx 1 marcusl marcusl   13 Jun 27 14:44 three -> one/two/three
```

Once again we see that it is correctly identified as a symbolic link, l, that its default name is the same as the directory it is pointing too, same as the files link had the same name as the file by default previously, and that we have the additional pointer after the links name showing us where it's going.

Perform "ls" and "cd" on the link. Does it act just as if you were standing in directory two/ performing the very same actions on three/?

While having entered the link directory with "cd", go back one step using "cd ..", where do you end up?

Moving, deleting or renaming the directory would, just like in the case with the file, break the link.

2.4 Grep - Searching for text

If you have a very large file, perhaps so large that opening it in program would be very hard on your computer. It could be a file containing biological data, it could be a loggfile of a transfer where we want check for any errors. No matter the reason a handy tool to know the existence of is the "grep" command.

What grep does is search for a specific string in one or many files. Case sensitive/insensitive or regular expressions work as well.

Let's start, as always, by cleaning our directory.

```
$ rm -r *
```

Then let's create a file with some text in it that we can grep around with. I have supplied some great text below (courtesy of Eleanor Farjeon).

```
$ nano textfile
```

```
Cats sleep anywhere, any table, any chair.
Top of piano, window-ledge, in the middle, on the edge.
Open draw, empty shoe, anybody's lap will do.
Fitted in a cardboard box, in the cupboard with your frocks.
Anywhere! They don't care! Cats sleep anywhere.
```

Now let's see how the grep command works. The syntax is:

```
grep "string" filename/filepattern
```

Some examples for you to try and think about:

```
$ grep "Cat" textfile
```

```
$ grep "cat" textfile
```

As you can see the last one did not return any results. Add a -i for case insensitive search.

```
$ grep -i "cat" textfile
```

Now let's copy the file and check both of them together by matching a pattern for the filenames.

```
$ cp textfile textcopy
$ grep "Cat" text*
```

The * will ensure that any file starting with "text" and then anything following will be searched. This example would perhaps be more real if we had several text files with different texts and we were looking for a specific string from any of them.

2.4.1 Exercise

Copy the file sample_1.sam to your folder using the command below

```
$ cp /proj/g2018034/labs/linux_additional-files/sample_1.sam .
```

(Note the '.' that indicates you're copying a file to the current directory!)

Use grep to search in the file for a specific string of nucleotides, for example:

```
$ grep "TACCACCGAAATCTGTGCAGAGGAGAACGCAGCTC
CGCCCTCGCGGTGCTCTCCGGGTCTGTGCTGAGGA" sample_1.sam
```

Try with shorter sequences. When do you start getting lots of hits?

This file is only a fraction of a genome, you would have gotten many times more hits doing this to a complete many GB large sam file.

Use grep to find all lines with chr1 in them. This output is too much to be meaningful. Send it to a file:

```
$ grep chr1 sample_1.sam > chr1.seq
```

The (>) redirected the output of grep to the file chr1.seq where you have now effectively stored all the chromosome 1 information. This technique is known as *output redirection*, and you'll see more of that later in the course.

(The astute student will notice that chr1.seq contains more than just chromosome 1 information. We will address this in an exercise below.)

2.5 Piping

A useful tool in linux environment is the use of pipes. What they essentially do is connect the first command to the second command and the second command to the third command etc for as many commands as you want or need.

This is often used in UPPMAX jobs and other analysis as there are three major benefits. The first is that you do not have to stand in line to get a core or a node twice. The second is that you do not generate intermediary data which will clog your storage, you go from start file to result. The third

is that it may actually be faster.

The pipe command has this syntax

```
command 1 | command 2
```

The "|" is the pipe symbol (on Swedish mac keyboards alt+7), signifying that whatever output usually comes out of command 1 should instead be directly sent to command 2 as input.

In a hypothetical situation you have a folder with hundreds of files and you know the file you are looking for is very large but you can't remember its name. Let's do a ls -lh and pipe the output to be sorted by filesize. -n means we are sorting numerically and not alphabetically, -k 5 says "look at the fifth column of output", which happens to be the filesize of ls command.

```
$ ls -lh | sort -k 5 -n
```

An example use would be to align a file and directly send the now aligned file to be converted into a different format that may be required for another part of the analysis.

2.5.1 Exercise

In this exercise, you will use `|` and `grep` to easily make a more refined search. It is possible to do this sort of thing in other ways, but piping is often convenient.

The file "chr1.seq" which you created above is supposed to contain only chromosome 1 information, but grepping for "chr1" also matches results like "chr15", which we don't want.

`grep -v` returns as a result the lines which do *not* match the search string.

Use grep to search `sample_1.sam` for chr1, pipe it to a grep command searching for strings lacking the wrong chromosomes (or a series of such commands), and redirect the results to chr1.seq.

You should end up with a file named chr1.seq with only chromosome 1 information.

We will have some more examples of pipes in the next section.

2.6 Word Count

Word count, or wc, is a useful command for counting the number of occurrences of a word in a file. This is easiest explained with an example. Let's return to our sample_1.sam.

```
$ wc sample_1.sam
233288 3666760 58105794 sample_1.sam
```

The output can be interpreted like this:

Number of lines = 233288

Number of words = 3666760

Number of characters = 58105794

To make this more meaningful let's use the pipes and grep command seen previously to see how many lines and how many times the string of nucleotides "CATCATCAT" exist in the file.

```
$ grep "CATCATCAT" sample_1.sam | wc
60  957 15074
```

To see only the line count you can add **-l** after wc and to count only characters **-m**.

2.6.1 Exercise

Output only the amount of lines that have "chr1" in them from sample_1.sam.

2.7 Extra material 1

This is some harder assignments, so don't worry about it if you didn't have time to do it.

Let's look at grep and use some regular expressions
http://www.cheatography.com/davechild/cheat-sheets/regular-expressions/

Task1: Using sample_1.sam find all lines that start with "@" and put them in a file called "at.txt".

Task2: Find all the lines that end with at least 3 numbers from at.txt. (hint: remember to escape {} with \{\})

2.8 Extra material 2

Sed is a handy tool to replace strings in files http://www.grymoire.com/Unix/Sed.html

Task: You have realized that all the chromosomes have been missnamed as "chr3" when they should be "chr4". Use sed to replace "chr3" with "chr4" in sample_1.sam and output it to sample_2.sam.

Some food for though: The solution to this replaces the first instance on each line of chr3, what if we have multiple instances? What if we had wanted to replace "chr1", this would effect chr10-19 as well! There are many things to consider :).

2.9 Extra material 3

Bash loops are great for moving or renaming multiple files as well as many many other uses. This is the subject of the 3rd day of the course, but here's a quick intro!

Create some files as seen below

```
$ touch one.bam two.sam three.bam four.sam five.bam six.sam
```

Task: All the files are actually in bam format. What a crazy mistake! Create a bash loop that changes all files ending in .sam to end with .bam instead.

This is of course not a real file conversion, that would have to be done with samtools, but we'll stick to just renaming the files in this loop-exercise.

Three hints:

1. The bash loop syntax is this:

```
$ for _variable_ in _pattern_; do _command with $variable_; done
```

2. To rename file1 to file2 you write this:

```
$ mv file1 file2
```

which effectively is the same thing as

```
$ cp file1 file 2
$ rm file1
```

3. Ponder how this can be used to your advantage:

```
$ i=filename
$ echo ${i/name}stuff
filestuff
```

Good luck and have fun! :)

3 UPPMAX Tutorial

Note: in syntax examples, the dollar sign ($) is not to be printed. The dollar sign is usually an indicator that the text following it should be typed in a terminal window.

3.1 Copying files needed for laboratory

To be able to do parts of this lab, you will need some files. To avoid all the course participants editing the same file all at once, undoing each other's edits, each participant will get their own copy of the needed files.
The files are located in the folder **/proj/g2018034/labs/uppmax_tutorial**

Next, copy the lab files from this folder. -r means recursively, which means all the files including sub-folders of the source folder. Without it, only files directly in the source folder would be copied, NOT sub-folders and files in sub-folders.

NOTE: Remember to tab-complete to avoid typos and too much writing.

Ex.

$ cp -r <source> <destination>

```
$ cp -r /proj/g2018034/labs/uppmax_tutorial ~/glob/uppmax-intro/
```

Have a look in ∼/**glob/uppmax-intro/uppmax_tutorial**:

```
$ cd ~/glob/uppmax-intro/uppmax_tutorial
```

```
$ ls -l
```

```
[marcusl@rackham1 uppmax_tutorial]$ ls -l
total 64
-rwxr-xr-x 1 marcusl marcusl 1188 Oct 15  2015 data.bam
-rwxr-xr-x 1 marcusl marcusl  246 Oct 15  2015 job_template.txt
[marcusl@rackham1 uppmax_tutorial]$
```

3.2 Running a program

Among the files that were copied is **data.bam**. Bam is a popular format to store aligned sequence data, but since it is a so-called binary format it isn't readable if you are human. Try it using **less**:

```
$ less data.bam
```

Not so pretty.. Luckily for us, there is a program called **samtools** (http://sourceforge.net/apps/mediawiki/samtools/index.php?title=SAM_FAQ) that is made for reading bam files.

To use it on uppmax we must first load the module for **samtools**. Try starting samtools before loading the module:

```
$ samtools
```

```
[marcusl@rackham1 uppmax_tutorial]$ samtools
-bash: samtools: command not found
```

That did not work, we have to load the module:

```
$ module load bioinfo-tools samtools
```

```
$ samtools
```

NOTE: All modules are unloaded when you disconnect from UPPMAX, so you will have to load the modules again every time you log in. If you load a module in a terminal window, it will not affect the modules you have loaded in another terminal window, even if both terminals are connected to UPPMAX. Each terminal is independent of the others.

```
[marcusl@rackham1 uppmax_tutorial]$ samtools

Program: samtools (Tools for alignments in the SAM format)
Version: 0.1.19-44428cd

Usage:   samtools <command> [options]

Command: view        SAM<->BAM conversion
         sort        sort alignment file
         mpileup     multi-way pileup
         depth       compute the depth
         faidx       index/extract FASTA
         tview       text alignment viewer
         index       index alignment
         idxstats    BAM index stats (r595 or later)
         fixmate     fix mate information
         flagstat    simple stats
         calmd       recalculate MD/NM tags and '=' bases
         merge       merge sorted alignments
         rmdup       remove PCR duplicates
         reheader    replace BAM header
         cat         concatenate BAMs
         bedcov      read depth per BED region
         targetcut   cut fosmid regions (for fosmid pool only)
         phase       phase heterozygotes
         bamshuf     shuffle and group alignments by name
```

To use samtools to view a bam file, use the following line:

```
$ samtools view -h data.bam
```

```
@HD     VN:1.0  SO:coordinate
@SQ     SN:chr1 LN:249250621
@SQ     SN:chr10        LN:135534747
@SQ     SN:chr11        LN:135006516
@SQ     SN:chr12        LN:133851895
@SQ     SN:chr13        LN:115169878
@SQ     SN:chr14        LN:107349540
@SQ     SN:chr15        LN:102531392
@SQ     SN:chr16        LN:90354753
```

-h also print the bam file's **header**, which is the rows starting with **@ signs** in the beginning of the file. These lines contain so called **metadata**; information about the data stored in the file. It contain things like which program was used to generate the bam file and which chromosomes are present in the file. Try running the command without the **-h** to see the difference.

The not-binary version (ASCII, or text version) of a bam file is called a **sam file**, which was just printed directly into the terminal window. The sam file is not to much use for us printed in the terminal window, aesthetics aside. It is probably much better to have the sam file saved as an actual file, something that is very easy to do. Any text that is printed to the terminal can be saved to a file instead of the terminal window using *output redirection*, >

Ex.

$ programname arguments > outfile

which will launch a program named **programname**, supply it with the argument **arguments**, and write any output that would have been printed to the screen to the file **outfile** instead.

To use this on samtools:

```
$ samtools view -h data.bam > data.sam
```

Look at the created file:

```
$ ls -l
```

The sam file is now human-readable. Try viewing it with **less**:

```
$ less data.sam
```

You can also edit the file with a text editor, e.g. **nano** or **gedit**:

```
$ nano data.sam
$ gedit data.sam &
```

Try deleting the whole last line in the file, save it, and exit.

3.3 Modules

To view which module you have loaded at the moment, type **module list**

```
[marcusl@rackham1 uppmax_tutorial]$ module list
Currently Loaded Modulefiles:
  1) uppmax            2) bioinfo-tools     3) samtools/0.1.19
```

Let's say that you want to make sure you are using the latest version samtools. Look at which version you have loaded at the moment.

Now type **module avail** to see which programs are available at UPPMAX. (On Milou, you can use **module spider**, which lets you search.) Can you find samtools in the list? Which is the latest version of samtools available at UPPMAX?

To change which samtools module you have loaded, you have to unload the the module you have loaded and then load the other module. To unload a module, use **module unload <module name>**. Look in the list from module list to see the name of the module you want to unload.

When the current samtools module is unloaded, load the newest samtools module (samtools/1.3). Type **module list** to check that it is loaded.

3.4 Submitting a job

Not all jobs are as small as converting this tiny bam file to a sam file. Usually the bam files are several gigabytes, and can take hours to convert to sam files. Longer or heavier programs are submitted as jobs to the queue

system. What you submit to the queue system is a script file that will be executed as soon as it reaches the front of the queue. The scripting language used in these scripts is **bash**, which is the same language as you usually use in a terminal i.e. everything so far in the lecture and lab has been in the bash language (cd, ls, cp, mv, etc.).

Have a look at **job_template.txt** in your **uppmax_tutorial** folder.

```
$ less job_template.txt
```

```
#! /bin/bash -l
#SBATCH -A g2016030
#SBATCH -p core
#SBATCH -J Template_script
#SBATCH -t 01:00:00

# load some modules
module load bioinfo-tools

# go to some directory
cd ~/uppmax-intro/uppmax_tutorial

# do something
echo Hello world!
```

Edit this file to make the job convert **data.bam** to a sam file named **jobData.sam**. Remember how the queue works? Try to approximate the runtime of the job (almost instant in this case) and increase it by $\sim 50\%$, and use that time approximation when writing your script file. Longer jobs will wait longer in the queue because it is harder to fit them into gaps in the queue! Also remember to change the project ID to match this course project (g2018034).

Remember, just write the command you would run if you were sitting by the computer, i.e. load the correct modules, go to the correct folder, and run samtools the right way.

Submit your job using sbatch:

```
$ sbatch job_template.txt
```

3.5 Viewing the queue

If you want to know how your jobs are doing in the queue, you can check their status with **squeue -u username** or **jobinfo -u username**.

Rewrite the previous sbatch file so that you book 3 days of time, and to use a node instead of a core. This will cause your job to stand in the queue for a bit longer, so that we can have a look at it while it is queuing. Submit it to the queue and run **jobinfo**.

Sometimes, the cluster has idle nodes and the above instructions don't seem to work. If so, you can specify the sbatch option `--nice=100000` or even `--hold` which will give your job the lowest possible priority or start it on hold, respectively.

```
$ jobinfo -u <username>
```

```
[dahlo@kalkyl3 glob]$ jobinfo -u dahlo

CLUSTER: kalkyl
Running jobs:
  JOBID PARTITION                     NAME     USER      ACCOUNT ST          START_TIME  TIME_LEFT  NODES CPUS NODELIST(REASON)
2226984     devel           InteractiveNode    dahlo     g2012157  R 2012-09-22T21:48:51      40:54      1    8 q36

Nodes in use:                           245
Nodes in devel, free to use:              9
Nodes in other partitions, free to use:  92
Nodes available, in total:              346

Nodes in test and repair:                 1
Nodes, otherwise out of service:          1
Nodes, all in total:                    348

Waiting jobs:
  JOBID    POS PARTITION                      NAME     USER      ACCOUNT ST          START_TIME  TIME_LEFT PRIORITY CPUS NODELIST(REASON)     FEATURES DEPENDENCY
2226987     12      node                   My_job    dahlo     g2012157 PD                 N/A 10-00:00:00   190000   96 (AssociationJobL        thin
[dahlo@kalkyl3 glob]$
```

If you look under the heading "**Waiting jobs**:" you'll see a list of all the jobs you have in the queue that have not started yet. The most interesting column here is the **POS** column, which tells you which position in the queue you have (12 in my example). When you reach the first place, your job will start as soon as there are the resources you have asked for.

In our case, we are not really interested in running this job at all. Let's cancel it instead. This can be done with the command **scancel**. Syntax:

$ scancel <job id>

You see the job id number in the output from jobinfo or squeue.

```
$ scancel 2226987
```

3.6 Interactive

Sometimes it is more convenient to work interactively on a node instead of submitting your work as a job. Since you will not have the reservations we have during the course, you will have to book a node using the **interactive** command. Syntax:

$ interactive -A <project id> -t <time> -p <node or core>

This will create a booking for you which has a higher priority than the jobs submitted with sbatch. That means that they will start faster, but only if the booking is shorter than 12 hours. If the booking is longer than 12 hours, it will get the standard priority. When the job starts you will be transferred to the node you got automatically.

Try closing down your current session on the reserved node you connected to in the beginning of the lab by typing exit. Then make a new booking using interactive:

```
$ interactive -A g2018034 -t 02:00:00 -p core
```

Congratulations, you are now ready to be let loose on the cluster!

4 Extra Exercises

This section contains more tips and build on what you have learned. For the purposes of the course, this material is optional. Don't forget to use `man` if you forget the usage of a command.
NOTE: in syntax examples, the dollar sign ($) is not to be printed. The dollar sign is usually an indicator that the text following it should be typed in a terminal window.

4.1 The devel queue

If you submit a really big job, it might be in the queue for a day or two before it starts, so it is important to make sure it does not immediately crash because you made a typo on line 7. One way to test this is to open a new connection to uppmax, and line by line try your code. Copy-paste (ctrl+shift+c and ctrl+shift+v in the terminal window) to make sure it's really the code in the script you are trying.

If your script is longer than a couple of lines, this approach can be tiring. There are 12 nodes at uppmax that are dedicated to do quick test runs, which have a separate queue called devel. They are available for use more or less all the time since not very many are using them. To avoid people abusing the free nodes for their analysis, there is a **1 hour time limit** for jobs on them. To submit jobs to this short testing queue, **change -p to devel** instead of node or core, and make sure **-t** is set to maximum **01:00:00**. Try submitting the samtools sbatch file we used earlier to the devel queue and run it again.

4.2 Information about finished jobs

If you want information about jobs you have run in the past, you can use the tool **finishedjobinfo**. It will print information about the jobs you have run lately.

Fun things to look for in this information is **jobstate** which will tell you if the program reported any error while running. If so, jobstate will be **FAILED** and you could suspect that something didn't go according to the plan, and you should check the output from that job run (the slurm-.out

file) and see if you can solve the error.

Other good things to look for could be:

- **maxmemory_in_GiB**: tells you how much memory the program used at most.
- **runtime**: tells you how long time the program ran before it finished/-failed

4.3 Time and space

Remember the commands uquota (show how much of your storage space you are using) and projinfo (shows you how much of your allocated time you have used) from the lecture? Try running them and see how you are doing.

4.4 Working graphically

Sometimes, you just need to look at something. It could the Matlab's user interface, a plotting function, or even a generated pdf file.

The old way is with *X11-forwarding*. If you connect to UPPMAX with `ssh -X`, the sshd server on UPPMAX starts an X-server that listens to a certain display number and sets the `$DISPLAY`nvironment variable to that display number. The X clients on UPPMAX will then read `$DISPLAY` and know to connect to this X server, which in turn transmits the X graphics commands to the SSH process on your system. Your local SSH process then uses the local X server on your system to draw the graphics.

Usually, all this works without you needing to know the details. If something stops working, though, an understanding of the steps involved can help you figure out a solution.

Try this: Make sure you log in to UPPMAX with the -X flag. In the terminal, use `echo $DISPLAY` to check `$DISPLAY`. Run `xclock &` to see it work. The `&` lets it run in the background of the terminal process, so you can continue working while you look at the time. Try changing `$DISPLAY` or logging in without -X to see what happens.

Other useful programs that use X are `evince` for reading document formats like PDF, `gedit` for Wordpad-like text editing, `gnuplot` for drawing figures, and endless more.

X11 forwarding relies on the transmission of drawing commands that were never really designed to be transmitted, and is therefore quite slow. UPPMAX offers (in limited edition) a faster graphical alternative, ThinLinc.

You can log in to a desktop session on Tintin or Milou with your browser by pointing to e.g. https://tintin-gui.uppmax.uu.se. You can also download the ThinLinc client, which replaces ssh.

For licensing reasons, there are some downsides to ThinLinc on UPPMAX. You will always end up on the same login node. If many people are using ThinLinc at once, we may run out of licenses. Let support know if there are problems.

4.5 Command History

You have probably already noticed that you can scroll through previously executed commands using the up and down arrow keys. This command history is stored in `~/.bash_history`. You can display your whole history with the command `history`.
Use `wc` to see how many entries are stored in your command history. Use `head` to display the first 12 commands stored in your command history. Make a note of one of these commands.
Using the arrow keys is handy if you want to repeat a command that you used just a minute ago, but after a whole day's work it can be very tedious to find an old command that you entered in the morning. Fortunately, there is a better way.
Press `Ctrl-r`. Your Bash prompt should change to display `(reverse-i-search)‘’:`.
Now start typing an old command. Bash tries to match what you are writing to a command in your history! You can type anything, e.g. a filename or the name of a program, and Bash will give you the most recent command that contains that word.
Curious about what you've done today? Use `grep`, `wc`, and `|` to count how many times you've used commands like `ls` and `cd`.

4.6 More Useful UPPMAX Tools

You have already looked at the command `finishedjobinfo`, and other commands were mentioned in the lecture. The directory `/sw/uppmax/bin` contains scripts that you can use to view and manage your jobs and projects. Take a look at the directory and see what's there. This is where you go if you forget what `finishedjobinfo` is called, for instance.
Try using `jobstats`, `jobinfo`, `finishedjobinfo`, `projinfo`, `projplot`, and `projsummary`. Some of these appear to be pretty similar, but each provides different information.