The UNIX operating system is made up of three parts; the **kernel**, the **shell** and the **programs**. The main control program in a UNIX OS is the kernel. The kernel does not allow the user to give it commands directly; instead when the user types commands on the keyboard they are read by another program in the OS called a **shell** which parses, checks and translates them in various ways and then passes them to the kernel for execution.

**Everything in UNIX is either a file or a process.** The UNIX **filesystem** is laid out as a hierarchical tree structure which is anchored at a special top-level directory known as the **root** (designated by a slash '**/**'). Because of the tree structure, a directory can have many child directories, but only one parent directory.

**Exercise 1: basic file and directory manipulation**

Open the “Terminal” application (there should be a black icon for it on the Dock at the bottom of the screen) and follow the commands below:

|  |  |
| --- | --- |
| $ **cd** | change directory to your home directory |
| $ **pwd** | prints the current directory |
| $ **ls –l** | ls means list, -l means long; this command lists all files and directories in the current directory |
| $ **mkdir** tmp | make directory; the command creates the directory “tmp” in your home directory |
| $ **cd** tmp | brings you to your newly created directory tmp |
| $ **pwd** | you can see that your current working directory changed |
| $ **cd ..** | move one level up |
| $ **ls -l** | you should see the newly created tmp directory |
| $ **vi tmp/poem.txt** | start editing file tmp/poem.txt |
| **press i** | press i to switch to “insert” mode in the vi editor |
| <http://www.poetryfoundation.org/poem/171621> | open web page and copy/paste the poem to the poem.txt file in your vi editor |
| **press ESC** | press ESC to switch to “command” mode in the vi editor |
| **:w** | press : and then w ENTER to write contents to file |
| **:q** | press : and then q ENTER to quit the vi editor and return to the shell |
| $ **cat tmp/poem.txt** | displays the poem |
| $ **head tmp/poem.txt** | displays first 10 lines of the poem |
| $ **tail tmp/poem.txt** | displays last 10 lines of the poem |
| $ **man** head | use **man** to get more information about the head command; use the correct parameter to display not only the first 10 lines of the poem but the entire poem |
| $ **less** tmp/poem.txt | less is a convenient command for displaying file contents (“less is more”; **more** is another command people used a lot before less was introduced) |
| $ **grep** house tmp/poem.txt | if you are searching for something, you can use **grep** *text* to display only lines matching text |
| $ **wc** tmp/poem.txt | displays the number of lines in the file |
| $ **wc** -w tmp/poem.txt | counts the words in the file |
| $ **cp** tmp/p + TAB | if you start typing “cp tmp/d” and then press TAB, what happens? |
| $ **cd** ~/tmp | change current directory to directory tmp |
| $ **cp** poem.txt poem2.txt | cp is used to copy the file data.tab to file data2.tab |
| $ **ls** -l | you should see 2 files now |
| $ **cd** .. | go back one level |
| $ **cp** -ap tmp poems | you just made a copy of the entire directory tmp to poems |
| $ **ls** -l | you should see both tmp directories |
| $ **ls** tmp/\*.txt | you can use the wildcard (\*) to list files and directories that match your expression |
| $ **rm** tmp/poem\*.txt | rm removes files and folders |
| $ **rmdir** tmp | finally we remove the bio directory |
| $ **clear** | clears the terminal display |

**Exercise 2: pipes and redirecting input / output**

The pipe ('**|**') operator is used to create concurrently executing processes that pass data between them:

$ command1 | command2 | command3 …

The standard output of command1 is redirected (piped) to the standard input of command2, etc.

|  |  |
| --- | --- |
|  | What is the standard **input** of the command “$ vi data.tab”? |

Pipes are useful for combining system utilities to perform more complex functions. For example:

$ cd poems

$ cat poem.txt | grep "House" | wc

creates three processes (corresponding to **cat**, **grep** and **wc**) which execute concurrently. As they execute, the output of the cat process is passed on to the grep process which is in turn passed on to the wc process. wc displays its output on the screen.

Redirecting input and output: The output from programs is usually written to the screen, while their input usually comes from the keyboard (if no file arguments are given). To redirect standard output to a file instead of the screen, we use the **>** operator:

|  |  |
| --- | --- |
| $ echo hello | displays “hello” on the screen |
| $ echo hello > hello.txt | writes “hello” to the file |
| $ cat hello.txt | displays contents of file; you should see “hello” |

In this case, the contents of the file data.tab will be overwritten if the file already exists. If instead we want to append the output of the echo command to the file, we can use the >> operator:

|  |  |
| --- | --- |
| $ echo hello >> hello.txt | appends “hello” to the end of the file |
| $ cat hello.txt | displays contents of file |

Standard input can also be redirected using the < operator, so that input is read from a file instead of the keyboard:

$ wc < hello.txt

You can combine input redirection with output redirection, but be careful not to use the same filename in both places. For example:

$ wc < hello.txt > hello\_counts.txt

**Exercise 3: writing and executing a shell script**

You now know the basics of the vi editor from the first exercise. Try to copy/paste the below simple bash program into a file and execute it. The hello world bash script:

#!/bin/bash

echo "Hello World"

Copy the above 2 lines and save them to the file “hello.sh”. The first line tells the Unix shell to interpret the program. In order to run the program you can either start it with:

Make your file executable by typing:

$ chmod +x hello.sh

And now you can simply type:

$ ./hello.sh

Note the “./” at the start of the command. This is because the directory where we stored hello.sh is not in the system variable $PATH.

**Exercise 4: download FASTA file and count the number of proteins**

Use the browser (Safari, Chrome) to download the FASTA file of proteins from:

<ftp://ftp.ensembl.org/pub/release-71/fasta/homo_sapiens/pep/Homo_sapiens.GRCh37.71.pep.all.fa.gz>

When downloaded (it will be stored in ~/Downloads), **gunzip** the file. Then count how many proteins are in the file (use **grep** and **wc**).

**Exercise 5: sort file on column**

Download the protein abundance data:

<http://pax-db.org/data/abundances/9606-human3_non-alkylatedPlasma_PRIDE.txt>

If using Safari, click "File / Save As and Format : Page source). Use the command “**sort**” to sort the protein abundance. Sort on column 3 (“abundance”). Explore the sort options by using “$ man sort” (especially look at the k and n options). Also, make the sorting reverse (put higher values on top).

 Can you **pipe** the output of **sort** to **tail** and reverse the output line order?

Write (redirect) the result to a new file (e.g. data\_sorted.tab).

 **Exercise 6: write a simple bash script**

Often, you would like to run the same command with different parameters. As an exercise, write a simple bash script that will output numbers from 1 to 100. Use a for loop.

#!/bin/bash

for i in {1..100}

do

echo $i

done

Save the above code to a file (e.g. script.sh), make the file executable (+x flag) and run it.

What is the output?

 **Exercise 7: iterating over files**

By using the same concept (**for** loop) from the previous exercise, can you try to iterate over all fasta files in a directory, print their name and the number of sequences in each file?

for filename in \*.fasta

Also use **grep** and **wc** to count the number of sequences in each file.

**\* Exercise 8: download and install bowtie2 software**

Bowtie2 is a short-sequence read aligner (e.g. 150nt long). The reads are aligned to a reference sequence (e.g. human genome).

Make a new ~/software directory and download bowtie2 from this link:

<https://sourceforge.net/projects/bowtie-bio/files/bowtie2/2.3.4.1/bowtie2-2.3.4.1-macos-x86_64.zip/download>

Unzip the file and try to run the command:

$ ./bowtie2

Since bowtie2 directory is not in the $PATH environment variable (a list of directory locations which Unix searches for commands when you try to run them), you can only run it from the bowtie2-2.1.0 folder or by providing the full path (~/Downloads/bowtie2-2.1.0). You can add the bowtie2 folder to the $PATH:

$ export PATH=$PATH:~/Downloads/bowtie2-…

Now you can simply type “bowtie2” anywhere (in any directory) and the **shell** will find the **bowtie2** software.

**\* Exercise 9: exploring a FASTA format file**

*Dictyostelium discoideum* ([www.dictybase.org](http://www.dictybase.org)) is an interesting social amoeba and a well studied model organism. The whole genome sequence is already available in the dd/dd.fasta file. The FASTA format is widely used in sequence distribution, see the description at <http://en.wikipedia.org/wiki/FASTA_format>.

Use **grep** to find out how many chromosomes are present in the file. Use **grep** (-v) to only print out the genomic sequence. How large is the genome?

Now look at the RNA-seq data sample stored in the dd/rnaseq.fasta.gz file. This file includes qualities for each nucleotide (see FASTQ description at <http://en.wikipedia.org/wiki/FASTQ_format>).

How many reads are in the file?

**\*Exercise 10: searching for short sequences in the *Dictyostelium discoideum* genome**

Is the sequence “AAAAAGAGATACAT” present in the DD genome (dd/dd.fasta)?

You can simply use **grep** to find out.

\* You can also use **bowtie2**. First build the index of the DD reference genome. The format is: “bowtie2-build <fasta\_file> <custom\_index\_name>”. In the dd folder, you could use:

$ bowtie2-build dd.fasta dd

Once the index is created (you do this only once for each reference genome, i.e. each FASTA file), you align one read (sequence) to the genome by typing:

$ bowtie2 dd -ac AAAAAGAGATACAT > dd.sam

Explore the SAM results file with “less -S”. The parameter “-S” prevents line wraps, so you can see one alignment per line.

Does **bowtie2** find more alignments compared to grep? Why could that be?

**\* Exercise 11: alignment of RNA-seq sample reads to the *Dictyostelium discoideum* genome**

To align the RNA-seq reads in the rnaseq.fastq file, you first need to index the *Dictyostelium discoideum* genome. Use bowtie2-build to create an index with name **dd** (if you didn't already build the index in the previous exercise):

$ bowtie2-build fasta\_file custom\_index\_name

After you created the index, you can align the reads by typing:

$ bowtie2 dd -U rnaseq.fastq.gz > dd.sam

The results are returned in SAM format and stored to the dd.sam file. How many reads align?

Use Bowtie2 manual (<http://bowtie-bio.sourceforge.net/bowtie2/manual.shtml>) to explore and change parameters. How do different parameters influence your results?

**\* Exercise 12: indexing a sam file**

Use the .sam file created from the previous exercise and make a .bam file (using samtools) and index it. How many reads align to chr1 region 50000-100000?

**System information, processes and other useful commands**

|  |  |
| --- | --- |
| **uname -a** | display system information |
| **man** *command* | display manual page of command |
| **df -h** | list mounted disks with available space |
| **du -h** *path* | show space usage |
| **top** | display running processes |
| **kill -9** *pid* | kill process |

**File and folder manipulation, compression**

|  |  |
| --- | --- |
| **pwd** | display current folder |
| **ls -l** *path* | list files and folders |
| **cd** *path* | change folder to path |
| **cd** ~ | change folder to home folder |
| **mkdir** *name* | make folder |
| **rmdir** *name* | remove folder |
| **cp** *source* *dest* | copy file/folder and all its contents |
| **less** *filename* | display file content |
| **wc** *filename* | count number of lines in file |
| **head** *filename* | shows first few lines of file |
| **tail** *filename* | shows last few lines of file |
| **gzip** *filename* | compress file with gzip (adds .gz extension) |
| **gunzip** *filename* | decompress filuncompress and remove .gz extension |
| **tar xfz** *filename.tar.gz* | uncompress files from tar.gz archive |
| **tar** **zcvf** *archive.tar.gz folder\_to\_compress* | creates archive.tar.gz |
| **unzip** filename.zip | unzip archive |

**Network and file transfer**

|  |  |
| --- | --- |
| **curl** URL -o filename | download URL to filename |
| **ssh** *username@host* | remote login to host with username |
| **sftp** username@host | remote login to host with username and transfer files |

**“vi” editor**

|  |  |
| --- | --- |
| **$ vi** *filename* | start editing file with vi |
| **i** | switch to “insert” mode |
| **ESC** | switch to “command” mode |
| **:w** | save |
| **:q** | quit |
| **:x** | save and quit |
| **/<pattern>** | search for pattern, <n> gives you the next match |
| **:q!** | quit without saving changes |