Introduction to Linux Command Line

Shuo Zhang
Penn Epigenetics Institute
03/11/2024

Description

This tutorial is designed to help you get familiar with Linux command line, which is a text-based interface to interact with a Linux system. The interface, often referred to as the shell, provides a powerful approach to perform computationally intensive work, such as RNA-seq and ChIP-seq data analysis. The workshop will introduce how to utilize a Linux platform: the high-performance computing (HPC) at Penn Medicine. Specifically, you will learn:

- How to login and exit HPC
- How to navigate the Linux file system, such as creating files and directories
- How to view the contents of a text file and process the file
- How to transfer data between the HPC and your personal computer
- How to write shell scripts
- How to submit jobs to the HPC

Prerequisites

- A personal computer (Windows, MacOS, or Linux)
- An HPC account

Login HPC

- For macOS, open spotlight 🔍 -> search Terminal to open it.

  Then, connect to the HPC using: ssh username@consign.pmacs.upenn.edu

- For Windows, install MobaXterm: https://mobaxterm.mobatek.net
    MobaXterm -> Session -> SSH
    Remote host: consign.pmacs.upenn.edu
    Check the box "Specify davidname"
    davidname: PMACS ID

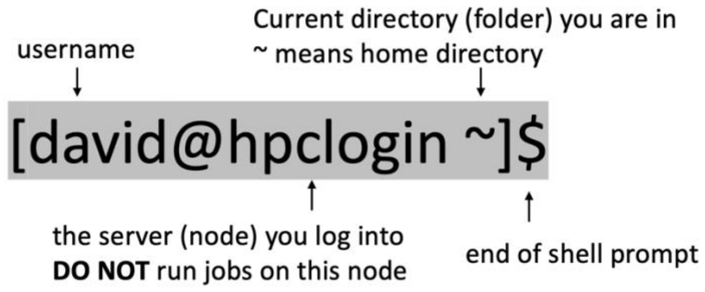
Note: to login HPC, you need to connect to UPenn's wifi. If you are off compus, please use VPN: https://hpcwiki.pmacs.upenn.edu/wiki/index.php/HPC:Login. If this is your first login, connect to mercury.pmacs.upenn.edu to initialize your home area.

Exit HPC

Use exit command to exit the command line interface:

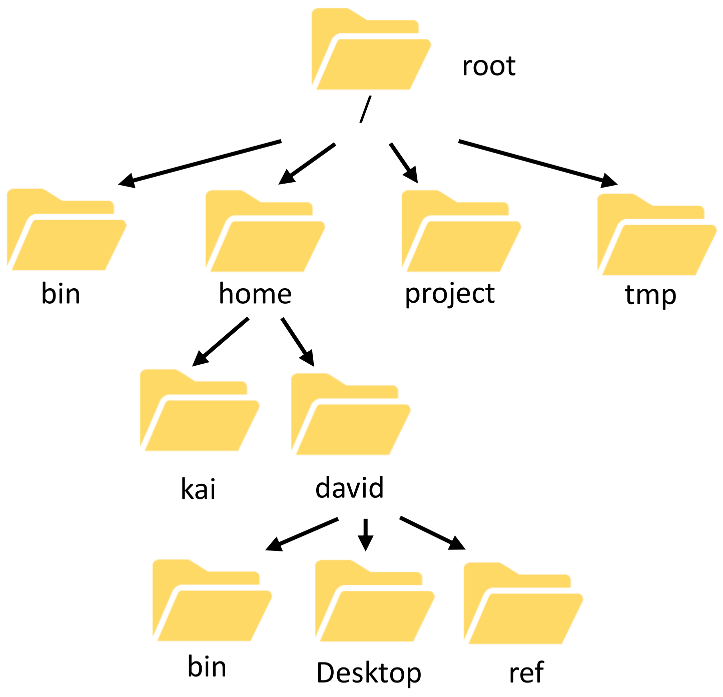
```
(base) [david@hpclogin ~]$ exit
logout
Connection to consign.pmacs.upenn.edu closed.
```

The shell prompt


username
Current directory (folder) you are in
~ means home directory
[david@hpclogin ~]$
the server (node) you log into
DO NOT run jobs on this node
end of shell prompt

Commands to manage the file system

The Linux file system is a tree-like structure that starts from root:



The pwd command prints the current working directory:

```
$ pwd
/home/david
```

ls -- lists directory contents:

```
$ ls
bin Desktop ref
```

mkdir -- creates a new directory:

```
$ mkdir test
$ ls
bin Desktop ref test
```

cd -- changes working directory:

```
$ cd test
$ pwd
/home/david/test
```

touch -- creates an empty file:

```
$ touch hi.txt
$ ls
hi.txt
```

cp -- copy a file/directory:

```
$ cp hi.txt Hi.txt                # Linux is case sensitive: hi.txt and Hi.txt are different
$ ls
hi.txt Hi.txt
```

- '#' and the rest of line are comments.

mv -- rename a file/directory or move a file/directory to a new directory:

```
$ mv Hi.txt hello.txt                  # Caution: it will overwrite hello.txt if it exists
$ ls
hello.txt hi.txt
$ mv hello.txt /home/david             # move to home directory
$ cd /home/david                       # change to home directory
$ ls
bin Desktop ref test hello.txt
```

rm -- remove files and directories:

```
$ rm hello.txt        # remove a file
$ ls
bin Desktop ref test
$ rm -r ref                  # remove a directory with -r
$ ls
bin Desktop test
```

Note: removed files and directories are very very hard to be recovered. The -i option can be used to request confirmation before removing each file.

Table 1: summary of frequently used commands for managing the file system.

Command	**Function**
pwd	print working directory
ls	list directory content
mkdir	make a directory
cd	change directory
touch	create an empty file
mv	move or rename a file/directory
cp	copy a file/directory
rm	remove a file/directory

Use man command (for example: man mv) to see the details of each command. ‘mv’ and ‘rm’ commands should be used with caution as they can overwrite or remove files.

Table 2: keyboard shortcuts

shortcut	**Function**
[TAB]	Auto completion
↑	Last history command
↓	Next history command
Ctrl + c	Kill running command
Ctrl + u	Clear all before the cursor
Ctrl + a	Move cursor to the beginning
Ctrl + e	Move cursor to the end

Table 3: wildcards

shortcut	**Function**
?	Match any single character
*	Match any characters (0 or more times)
[]	Match a range
[!]	Match characters not in the range

Absolute and relative paths

```
$ pwd
/home/david/test       # This is an absolute path that starts from the root ('/') directory
$ ls -a
. .. hi.txt
$ cd ..                # change to parent directory
$ pwd
/home/david
$ cd ./test            # the same as 'cd test'
$ pwd
/home/david/test
```

- -a: controls the behavior of ls command, i.e., displays all contents. Most commands have this syntax: **cmd [options] <arguments>**
- '.': current working directory
- '..': the parent directory of current working directory
- absolute path starts from root ('/'). '.' and '..' are used in relative path.

Table 4: frequently used cd commands

Command	Function
cd ..	Go to parent directory
cd -	go to the previous directory
cd	go to home directory
cd ~	go to home directory

Commands to view and process files

cat -- displays the whole content of a file:

```
$ cat gene_list.txt
gene_name        chr    start  end
Xkr4  chr1  3205901    3671498
Rp1   chr1  3999557    4409241
Sox17   chr1     4490931         4497354
…
```

head -- display the first (head) or last (tail) n lines of a file, respectively:

```
$ head -n 3 gene_list.txt
gene_name        chr    start  end
Xkr4  chr1  3205901    3671498
Rp1   chr1  3999557    4409241
```

-n: set the number of lines (10 by default) to be printed 10. The command head -n 4 is equivalent to head -4.

Table 5: summary of commands to view and process a file.

Command	**Function**
cat	concatenates file together, can be used to print a file
head	display the first n (10 by default) lines of a file
tail	display the last n (10 by default) lines of a file
less	display a file one page per time, useful for viewing large files; press ‘q’ to exit
wc	count a file’s line, word, character
cut	cut out parts of each line; -f for column, -d for delimiter
grep	Search for a pattern of each line
sort	display the last n (10 by default) lines of a file
uniq	remove duplicates lines in a file
sed	stream editor: search, replace

Pipe and redirection

pipe '|' connects two commands:

```
$ grep 'chr2' gene_list | sort -n -k3 | head -5     # first 5 genes on chr2 sorted by position
Fam171a1 chr2  3114224     3227806
Nmt2chr2  3284212     3328877
Rpp38       chr2  3328949     3332643
Acbd7       chr2  3336168     3340993
Olah     chr2     3341982         3397210
```

Redirection

```
$ grep 'chr2' gene_list | sort -n -k3 | head -5 > chr2_genes.txt
$ ls
chr2_genes.txt gene_list.txt
$ grep 'chr2' gene_list | sort -n -k3 | tail -5 >> chr2_genes.txt
```

- '>': create a new file if the file doesn't exist; overwrite if the file exists
- '>>': create a new file if the file doesn't exist; append if the file exists

vim editor

```
$ vim hi.txt                              # create a file if it doesn't exit
~
~
~
~
"hi.txt" 0L, 0C
```

Table 6: frequently used keys for vim

key	**Function**
i	Enter insert mode
[esc]	Leave insert mode
:wq	Save editing and leave vim
:q!	Leave without saving

Data transfer between HPC and your local computer

- Command line tools: scp, rsync
- Graphical tools: FileZilla

File Compression

gzip -- compress a file

```
$ ls
hi.txt
$ gzip hi.txt
$ ls
hi.txt.gz
```

- Compressed files is ~4-5 time less in size

gunzip – decompress a file

```
$ gunzip hi.txt.gz
$ ls
hi.txt
```

The commands below show how to concatenate fastq files from 4 lanes

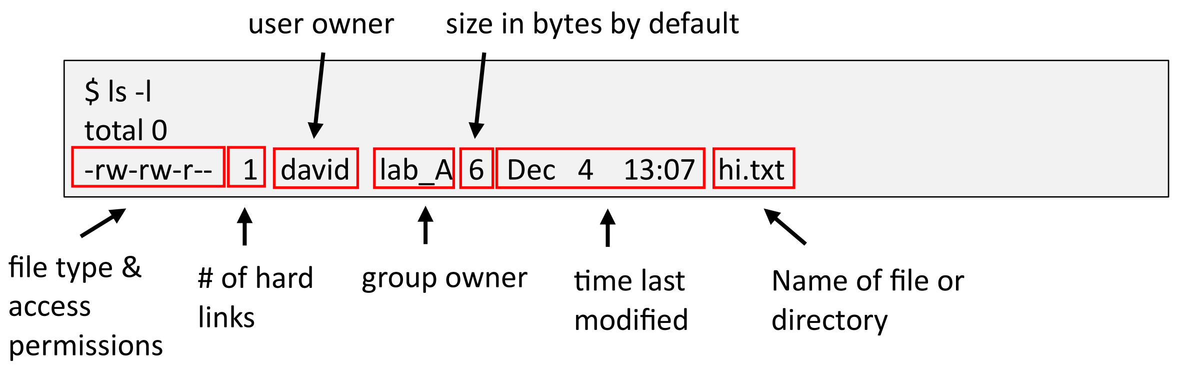
```
$ ls
S1_L001_R1.fastq.gz S_L002_R1.fastq.gz S1_L003_R1.fastq.gz S1_L004_R1.fastq.gz
$ gunzip -c S1_L001_R1.fastq.gz > S1_R1.fastq       # create a new file for the first lane
$ gunzip -c S1_L002_R1.fastq.gz >> S1_R1.fastq      # append the second lane
$ gunzip -c S1_L003_R1.fastq.gz >> S1_R1.fastq      # append the third lane
$ gunzip -c S1_L004_R1.fastq.gz >> S1_R1.fastq      # append the fourth lane
$ gzip S1_R1.fastq                                  # compress the combined file
```

- -c: write output on standard output (usually the screen)

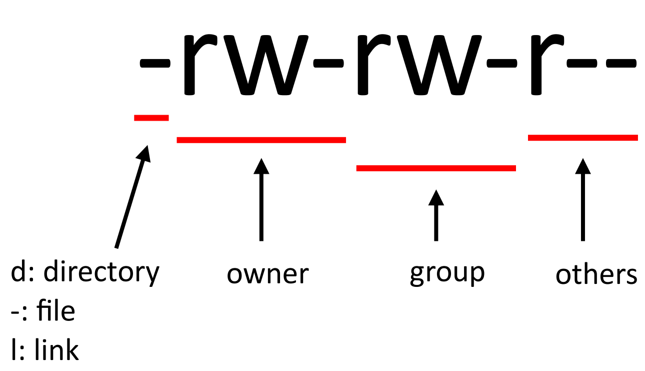
Table 7: frequently used file compression commands

key	Function
gzip	compress
gunzip	decompress
tar	create compressed and archived files

File permission



```
$ ls -l
total 0
-rw-rw-r--  1  david  lab_A  6  Dec   4   13:07  hi.txt
```



letter	permission
r	read
w	write
x	execute
-	no permission

chmod -- change file/directory mode (permission):

```
$ chmod u+x hi.txt                  # add execute permission to owner
$ ls -l
total 0
-rwxrw-r--    1   david    lab_A 6   Dec   4    13:07   hi.txt
$chmod g-w hi.txt                   # remove write permission from group
$ ls -l
total 0
-rwxr--r--    1   david    lab_A   6   Dec   4    13:07   hi.txt
```

letter	class
u	owner
g	group
o	others
a	all

operator	operation
+	add
-	remove
=	Set equal to

Practice project: use HPC for read mapping
Henikoff Lab present CUT&Tag for epigenomic profiling (Kaya-Okur et al., 2019). In the study, they showed *E. coli* DNA derived from transposase protein production could be used for between sample normalization, acting like a spike-in. To perform the normalization, we need to know the read counts mapped to *E. coli* and the target genome (human in the study), respectively. In this project, we will figure out how to get the read counts aligned to *E.coli*.

As this is a compute-intensive task, we need to use an interactive node:

```
$ bsub -Is bash
```

- This command asks the system to dispatch resources (one CPU core and 60 GB memory by default) for your private usage.

Download the required data:

```
$ git clone https://github.com/szhang32/Shell_tutorial.git
```

The test data (SRR8383505_1M_R1.fastq.gz and SRR8383505_1M_R2.fastq.gz) contain 1 million read pairs for H3K27me3 profiling from 60 K562 cells (SRR8383505)(Kaya-Okur et al., 2019). We can use bowtie2 to map the reads to *E.coli* genome (in the 'ref' directory).

We first add the version of bowtie2 we would like to use to our environment:

```
$ module add bowtie2/2.3.4.3
```

- Software and its settings are organized into modules for easy usage

Table 8: commands for module usage

Command	**Function**
module avail	List all available modules
module load XXX	Load module 'XXX'
module add XXX	The same as module load
module unload XXX	Unload module 'XXX'
module list	List all loaded modules

Then, paired-end reads are mapped to *E.coli*:

```
$ bowtie2 -x ref/ecoli -1 raw_fastq/SRR8383505_1M_R1.fastq.gz -2
raw_fastq/SRR8383505_1M_R2.fastq.gz -S ecoli_alignment
```

- -x: the basename of the index for the reference genome
- -1: read 1 of the paired-end reads

- -2: read 2 of the paired-end reads
- -S: output file in SAM format

Put all commands into a single file:

```
$ cat main_interactive.sh
#!/bin/bash
cd ..
module add bowtie2/2.3.4.3
bowtie2 -x ref/ecoli -1 raw_fastq/SRR8383505_1M_R1.fastq.gz -2
raw_fastq/SRR8383505_1M_R2.fastq.gz -S ecoli_alignment
```

- The first line '#!/bin/bash' instructs the system how to interpreter the rest of commands. It starts with a shebang, which consists of number sign ('#') and exclamation mark ('!').
- Putting all commands in a file helps us memorize what we have done, which version a software is used, which parameters are used, and others.

Run all commands one-by-one on the interactive node:

```
$ bash main_interactive.sh
```

Running can also be accomplished by giving the script execute permission:

```
$ chmod u+x main_interactive.sh
$ ./main_interactive.sh
```

Submit non-interactive jobs:

```
$ cat main_lsf.sh
#!/bin/bash
#BSUB -J main_lsf                    # job name
#BSUB -oo main_lsf.o                 # standard output
#BSUB -eo main_lsf.e                 # standard error output
#BSUB -M 2000                        # memory limit in Mb
cd ..
module add bowtie2/2.3.4.3
bowtie2 -x ref/ecoli -1 raw_fastq/SRR8383505_1M_R1.fastq.gz -2
raw_fastq/SRR8383505_1M_R2.fastq.gz -S ecoli_alignment

$ bsub < main_lsf.sh                 # submit the script for running
```

- IBM LSF (load sharing facility) is a resource management platform for HPC.

Table 9: summary of bsub commands

Command	Function
bsub	Submit a job
bjobs	Display job information
bkill	Sends a signal to a job

Shell variables
A variable is a named container that stores a single value (scalar variable) or multiple values (array variable).
Rules for variable name: could only contain alphabets (a-z, A-Z), digits (0-9), and underscore (_).
A variable name must start with an alphabet or underscore.

```
# valid variable names                  # invalid variable names
myvar                                   my$var
my_var                                  123myvar
MY_VAR
_myvar123
```

Define variables

```
variable_name=value
             ↑↑
          no space
```

```
$ first_name="David"          # "" is usually used to represent a string
$ echo "$first_name"          # echo command is used to display message
David
$ last_name="love"
$ name="$first_name $last_name"    # create a new variable using existing variables
$ echo "$name"
David Love
```

- $: extract the value of the variable

Environmental variables:

```
$ echo "$HOME"                # home directory
/home/david
$ echo "$PATH"                # a list of paths to search for executables
```

- environmental variables affect how programs run

Read file line-by-line

read -- reads text from standard input (usually keyboard)

```
$ read first_name                        # save user input to a variable
David
$ echo "$first_name"
David
$ read first_name last_name              # split user input into multiple variables
David Love
$ echo "$first_name"
David
$ echo "$last_name"
Love
```

- Default split delimiter is a space " ", which can be set by IFS.

while loop

```
while [ condition ]
do
        commands
done
```

- one-linear syntax: while [condition]; do commands; done

combine while loop and read command:

```
while read line
do
        echo $line
done < infile.txt
```

- '<': input redirector

for loop

```
Syntax:
for <var> in <a list of items>
do
        commands
done
```

```
Example:
for i in {1..10}
do
        echo "$i"
done
```

if-statement

```
Syntax:
if [ condition ]
then
        commands
fi
```

```
Example:
if [[ 1 -lt 2 ]]
then
        echo "1 is less than 2"
fi
```

Table 10a: operators to compare two numbers

operator	**Function**
-eq	Check if two numbers are equal
-ne	Check if not equal
-gt	greater than
-lt	less than
-ge	greater than or equal
-le	less than or equal

Table 10b: operators to compare two strings

operator	**Function**
==	Check if two strings are equal
!=	Not eqaul
>	greater than
<	less than

- '==' is the same as '='

References and additional resources

- UPenn HPC login: https://hpcwiki.pmacs.upenn.edu/wiki/index.php/HPC:Login
- PMACS service Desk: https://helpdesk.pmacs.upenn.edu
- Linux command line book: http://linuxcommand.org/tlcl.php
- Platform LSF Quick Reference: https://www.med.upenn.edu/hpc/assets/user-content/documents/lsf-quick-reference_user_commands.pdf
- Kaya-Okur, H. S., Wu, S. J., Codomo, C. A., Pledger, E. S., Bryson, T. D., Henikoff, J. G., ... & Henikoff, S. (2019). CUT&Tag for efficient epigenomic profiling of small samples and single cells. *Nature communications*, *10*(1), 1930.