First Steps with UNIX and Science Cluster

BIO298 Microbial bioinformatics block course
Fanny Wegner
2023-03-21

Learning outcomes

- Connecting to Science Cluster
- Get familiar with the UNIX environment
- Use the most common UNIX commands
- How to write simple bash scripts
- Run singularity containers
- How to submit a job to Science Cluster

Connecting to Science Cluster

- High performance computing environment of UZH
- Many tools with different advantages available for remote connection
- To connect on command line: `ssh <shortname>@s3it.cluster.uzh.ch`

  Hostname
- However, graphical tools with SFTP make things a lot easier:
  For Mac e.g., **Termius**
  For Windows e.g., **MobaXterm**
- Transfer of files on command line:
  ```
  scp my_local_file.txt <shortname>@cluster.s3it.uzh.ch:<target directory>
  ```
- To disconnect just type `exit`

Science Cluster

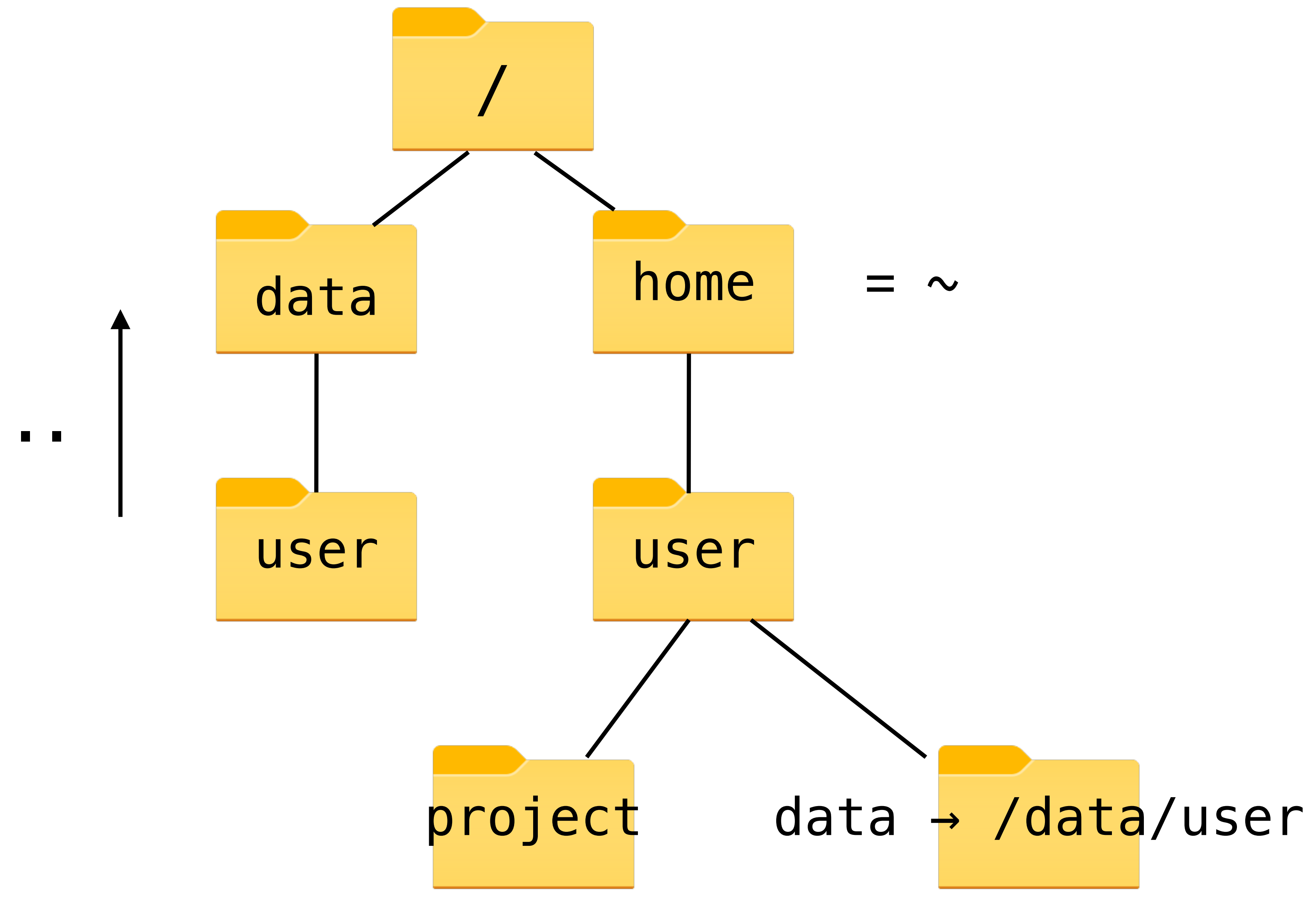
- Science Cluster uses **bash** as shell (= command line interface)
  others: zsh (Mac), csh, sh, tcsh …

- Four filesystems for data storage
  your **home** filesystem: `/home/cluster/$USER` or `~` 15 GB/100k files
  your personal **data** filesystem: `/data/$USER` or `~/$USER` 200 GB
  scratch filesystem: `/scratch/$USER` 20 TB
  group-specific shared filesystem Do you all see `amr.imm.uzh`?

- SSH connection is to login nodes, computations are done on computational nodes

Navigating

- `pwd` - print working directory
- `cd` - change directory
  ```
  cd /
  cd ~
  cd ..
  ```
- `ls` - list files
- Directories behave similar to files in UNIX
- Symlinks can be used for files and directories
- [tab] will autocomplete any commands or available files



EXERCISE 1

Files and directories

- `cp` - copy files and directories
- `mv myfile target_directory/` - move file (or directory)
  `mv myfile newname` - rename file (or directory)
- `mkdir` - create a new directory
- `rm` - delete files and directories
- `ln -s file link_to_file` - create a symbolic link for a file (or directory)

EXERCISE 2

Read and manipulate files

- `cat` - print all content of a file
- `less` or `more` - print some content
  for `less`: `g` - go to top of file, `SHIFT+g`, go to bottom of file, `/word` to search for 'word'
- `head` - look at the first 10 lines
  `tail` - look at the last 10 lines
- `touch` - create an empty file or change the “modified by” date of existing file
- `nano` - simple editor
  `vi` or `vim` - advanced editor
- `wc` - count lines, words and bytes
  `–l` lines
  `–w` words
  `–c` characters
- `cut –d [delimiter] –f [field]` - cut out a specified field by separated by delimiter
- `grep` - print lines in a file that match pattern

Input and output

- `echo “some statement”`- print statement to standard output
- `echo “Hello world!” > hello.txt` - print statement and direct to (new) file
- `cat myfile.txt > mynewfile.txt` - print content of file and direct it to new file
  `cat anotherfile.txt >> mynewfile.txt` - print content and concatenate it to existing file
- `command1 | command2` - Directs output from command1 as standard input into command2 (“piping”)
  example: count number of files in current directory
  `ls | wc -l`

EXERCISE 3

Variables

- You can store text or numbers in variables to use in scripts or on the CLI
- Definition `myvar=“Hello world”`

  No space around the “=“!
- Calling `$myvar` or `${myvar}`

  example:
  ```
  echo $myvar
  “Hello world”
  ```
- `Pre-defined variables in UNIX: $USER, $HOME, $PATH etc.`
- Can be concatenated with other variables or strings, for example
  ```
  sample_id=“ESC00142”
  ./make_alignment.sh ${sample_id}.fastq.gz ${sample_id}_reference.fasta
  ```
- Output from commands can also be stored into a variables
  ```
  output=$(command)

  numfiles=$(ls | wc -l)
  ```

Control structures

- **for loop** - do something for a specified number of times

```
for i in {1..n}
do
    command $i
done
```

Defined sequence of numbers

Is a number

```
for i in *
do
    command $i
done
```

For all files in your working directory

Is a filename

example1: print the names of all files starting with A

```
for i in A*; do echo $i; done
```

One-liner syntax

- **while loop** - do something while some condition is TRUE

```
while [ condition ]
do
    command1
    command2
done
```

Space around the [ ] !

Ideally one of these commands changes the condition as some point, otherwise you'll end up with an infinite loop

Control structures

- **if else** - do something if a condition is true

```
if [ condition ]
then
   command1
   command2
fi
```

```
if [ condition ]
then
   command1
else
   command2
fi
```

```
if [ condition ]
then
   command1
elif [ condition2 ]
   command2
else
   command3
fi
```

- Also possible to nest if statement

Arrays

- Data structure that stores multiple elements

- Definition

```
myarray=(1 2 3 4 5 6)
```

- Calling

```
echo ${myarray[@]}
1 2 3 4 5 6
echo ${myarray[0]}
1
```

Curly brackets required

All elements are accessed with “@“

Indexing starts at 0, so this accesses the first element

- Looping through array elements

```
for i in ${myarray[@]}
do
   command $i
done
```

Looping through array indices

This expression gives you the indices of each element

```
for i in ${!myarray[@]}
do
   command ${myarray[$i]}
done
```

Writing simple scripts

- A script is just the concatenation of a series of commands that are executed one after the other

- A shell script always starts with the shebang #!, defining the shell it will operate in

```
#!/usr/bin bash

command1
command2    # this will do X
command3
```

→ myscript.sh

You can explain your code (to yourself and others) using comments (good practice!)

- To run a script

```
./myscript.sh
```

A quick note: Scope of variables

- **Local variables**: If you define variables on your command line, it will only be available in this instance of your shell, not anywhere else or when you re-login in

- Similarly, variables defined within scripts only exist in there

- **Global variables**: Valid everywhere. For example, $USER

EXERCISE 4

Singularity containers

- Singularity = container platform, that packages up pieces of software
- A container = an image (.sif or sometimes .img)
- Portable and reproducible, safe
- To execute the software from an image: `singularity_image.sif command [parameters]`
- On Science Cluster, we “install” most our software as singularity images.
  To run them, you need to load a required module
  ```
  module load singularityce
  ```
- Note: singularity does not work for Mac command line, only on UNIX/linux systems!

Submission of jobs on Science Cluster

- Jobs can either be single commands or scripts
- Submissions scripts are bash scripts with a special header defining parameters for the job we request to run

```
#!/usr/bin/env bash
#SBATCH --cpus-per-task=[number]
#SBATCH --mem=[memory]
#SBATCH --time=[hr:min:sec]
#SBATCH --job-name=[name]
#SBATCH --output=[name]_%j.out
#SBATCH --error=[name]_%j.err

# load any required modules
module load [module name]

command1
command2
command3
```

More specific example for our use case

```
#!/usr/bin/env bash
#SBATCH --cpus-per-task=8
#SBATCH --mem=5G
#SBATCH --time=02:00:00
#SBATCH --job-name=myJob
#SBATCH --output=myJob_%j.out
#SBATCH --error=myJob_%j.err

# load any required modules
module load singularityce

path/to/singularity/module.sif command [parameters]
```

Submission of jobs on Science Cluster

- Submission: `sbatch myJob.sh`
- What's happening: `squeue —u $USER`
- Output from the script (which is normally printed to standard output) can be directed to `.out` and `.err` files

EXERCISE 5