

Introduction to the Command Line

Jan Moren, SCDA



OKINAWA INSTITUTE OF SCIENCE AND TECHNOLOGY GRADUATE UNIVERSITY

沖縄科学技術大学院大学

Set Up SSH

- Any OIST member can use the HPC resources. Apply here:
 - <https://scda-guide.oist.jp/request-access.html>
Select “Open Resources”
- OSX Users
 - You should already have SSH
 - Install “XQuartz” for graphics (reboot after installation)
- Windows Users
 - Install free “MobaXTerm”. Can use SSH and graphical applications.
- Linux and BSD Users
 - You already have everything.

<https://scda-guide.oist.jp/connect-clusters.html>

The Command Line

We have a perfectly fine graphical environment.
Why do we bother with a command line?!

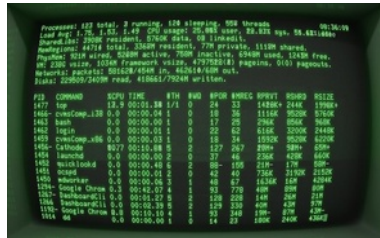


```
Processes: 123 total, 3 running, 120 sleeping, 558 threads      08:36:09
Load Avg: 1.75, 1.53, 1.49  CPU usage: 25.88% user, 28.83% sys, 55.6% idle
SharedLibs: 3908K resident, 5760K data, 0B linkedit.
MemRegions: 4471# total, 3358M resident, 77M private, 1118M shared.
PhysMem: 921M wired, 5288M active, 758M inactive, 6948M used, 1243M free.
VM: 238C vsize, 1034M framework vsize, 4797628(2) pageins, 0(0) pageouts.
Networks: packets: 581628/454M in, 462610/68M out.
Disks: 229809/3409M read, 418661/7924M written.

PID  COMMAND      %CPU  TIME    #TH  #WO  #PDR  #MREG  RPRVT  RSHRD  RSIZE
1477  top            12.9  00:01.38  1/1   0    24    33    1428K+ 214K  1998K+
1466  cvmsComp_i38  0.0   00:00.04  1     0    18    36    1116K  9528K  5760K
1463  bash           0.0   00:00.00  1     0    17    25    296K   856K   968K
1462  login          0.0   00:00.01  1     0    22    62    616K   3200K  2448K
1459  cvmsComp_x86  0.0   00:00.03  1     0    18    34    1592K  9528K  6220K
1486  Cathode       $077  00:10.88  5     2    127   267    28M+   92M+   68M+
1454  launchd       0.0   00:00.00  2     0    37    46    236K   428K   660K
1482  quicklookd    0.0   00:00.48  6     2    88-   155    21M-   17M-   58M-
1481  ocspd         0.0   00:00.01  2     0    42    40    736K   3192K  2152K
1450  mdworker      0.0   00:00.06  3     1    48    67    1636K  16M   4284K
1284  Google Chrom  0.3   00:42.07  4     1    93    778    48M   89M   80M
1267  DashboardCll  0.0   00:01.27  5     2    128   228    14M   26M   21M
1266  DashboardCll  0.0   00:02.39  5     2    129   330    40M   43M   97M
1192- Google Chrom 0.8   00:10.10  4     1    93   348    19M-   87M-  43M-
1014  dd            0.0   00:00.00  1     0    14    23    180K   240K  436K
```

The Command Line

GUIs (graphical user interfaces) and CLIs (command line interfaces) can both do everything



- Start applications:
browser, editors, simulations, games
- File and data management
organize, copy, send, clean up
- System stuff
config, wifi, hardware, preferences

Why The Command Line?



- It's Precise and Composable
Combine commands to quickly do very complex tasks
- It's Scriptable
Automate recurring tasks.
- It's Low Bandwidth
Access from anywhere, work on any device.

Why **not** The Command Line?



- It's Opaque and hard to Explore
What is even available?! What can you do?!
- It's Intimidating
Feels like you can break things at any time.



Command Line

We show our command line examples like this:

```
# log in to Deigo  
$ ssh -X your-name@deigo.oist.jp
```

- "\$" is a prompt where you can type things. We use it to show commands you run. Don't type the prompt.
- "#" starts a Bash comment.

Here is our introduction page on using the command line:

<https://scda-guide.oist.jp/command-line-introduction-bash>

We show our command line examples like this:

```
# log in to Deigo  
$ ssh -X your-name@deigo.oist.jp
```

- "\$" is a prompt where you can type things. We use it to show commands you run. Don't type the prompt.
- "#" starts a Bash comment.

Command Line Examples

```
# log in to Deigo
$ ssh -X your-name@deigo.oist.jp

# copy training data to your home
$ cp -r /apps/share/training/Bash .

# Edit a slurm file with the 'nano' editor
$ nano my-script.slurm

# run the firefox browser
$ firefox
```

Components:

- command
- options
- parameters

\$ ssh -X your-name@deigo.oist.jp

command

option

parameter

\$ cp -r /apps/share/training/Bash .

command

option

parameters



Command Line Examples

```
# log in to Deigo
$ ssh -X your-name@deigo.oist.jp

# copy training data to your home
$ cp -r /apps/share/training/Bash .

# Edit a slurm file with the 'nano' editor
$ nano my-script.slurm

# run the firefox browser
$ firefox
```

Components:

- **command**
- options
- parameters

Command: The name of an application

- Most are programs
- A few are built in to the shell itself

Command Line Examples

```
# log in to Deigo
$ ssh -X your-name@deigo.oist.jp

# copy training data to your home
$ cp -r /apps/share/training/Bash .

# Edit a slurm file with the 'nano' editor
$ nano my-script.slurm

# run the firefox browser
$ firefox
```

Components:

- command
- **options**
- parameters

Options: Changes how the program works

- One "-" and one letter ("-X"), or two "--" and a word ("--verbose")
- Short options can often be combined: "-s -t" → "-st"
- Options can have values ("--user jan-moren")

Command Line Examples

```
# log in to Deigo
$ ssh -X your-name@deigo.oist.jp

# copy training data to your home
$ cp -r /apps/share/training/Bash .

# Edit a slurm file with the 'nano' editor
$ nano my-script.slurm

# run the firefox browser
$ firefox
```

Components:

- command
- options
- **parameters**

Parameters: What the application should work on

- ssh: name of remote computer
- copy: source and destination
- text editor: name of file to edit

Let's Log In

Log in to Deigo:

```
$ ssh your-name@deigo.oist.jp
```

copy the slides, example scripts and programs to your home:

```
$ cp -r /apps/share/training/Bash .
```

Let's Log In

allow graphical applications across
this connection

Log in to Deigo:

Your user name

```
$ ssh -X your-name@deigo.oist.jp
```

The remote computer

copy the slides, example scripts and programs to
your home:

```
$ cp -r /apps/share/training/Bash .
```

Let's Log In

Log in to Deigo:

```
$ ssh -X your-name@deigo.oist.jp
```

copy entire folder,
not just one file

copy the slides, example scripts and programs to
your home:

```
$ cp -r /apps/share/training/Bash .
```

What to copy

Where to copy it

Let's Log In

Log in to Deigo:

```
$ ssh -X your-name@deigo.oist.jp
```

copy the slides, example scripts and programs to your home:

```
$ cp -r /apps/share/training/Bash .
```

Handy Tip:

Avoid typing with *tab* completion:

```
$ cp -r /a<tab>/sh<tab>/t<tab>/B<tab> .
```

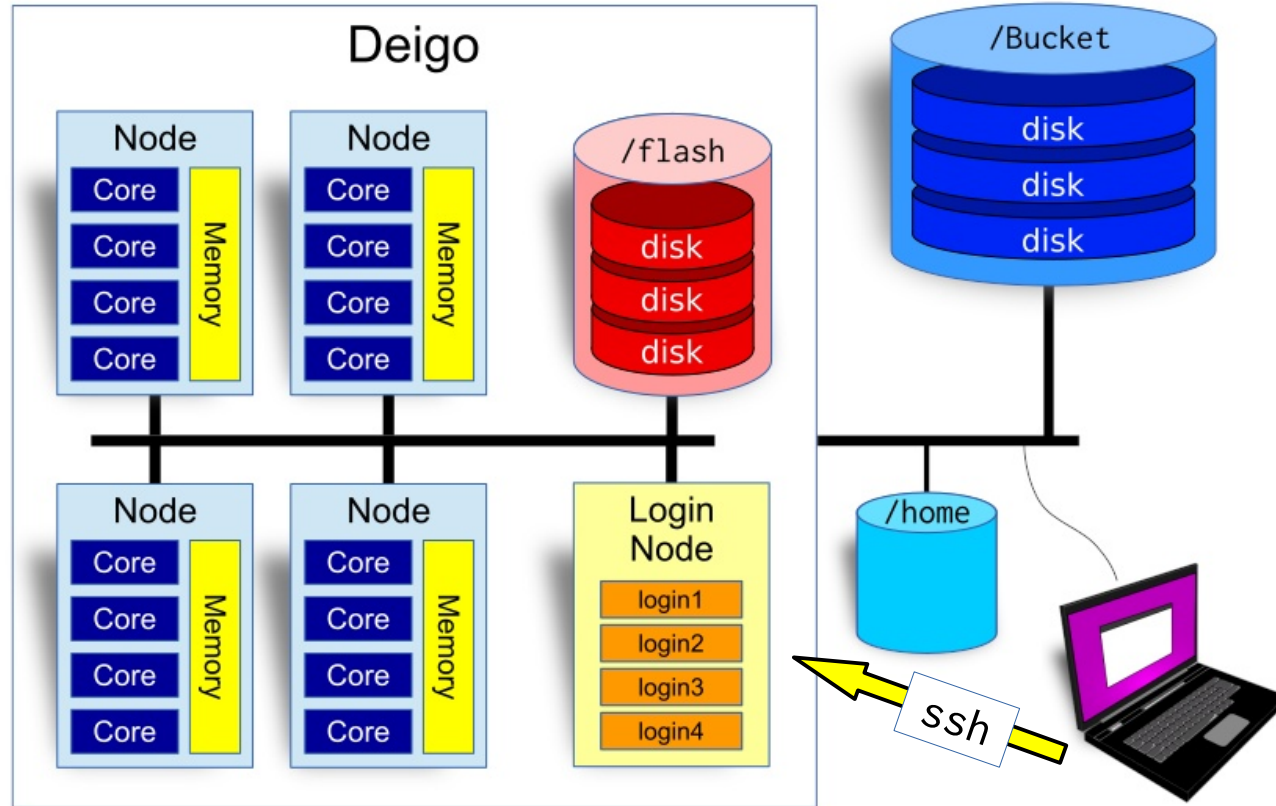


Press the *tab* key
to fill in the name

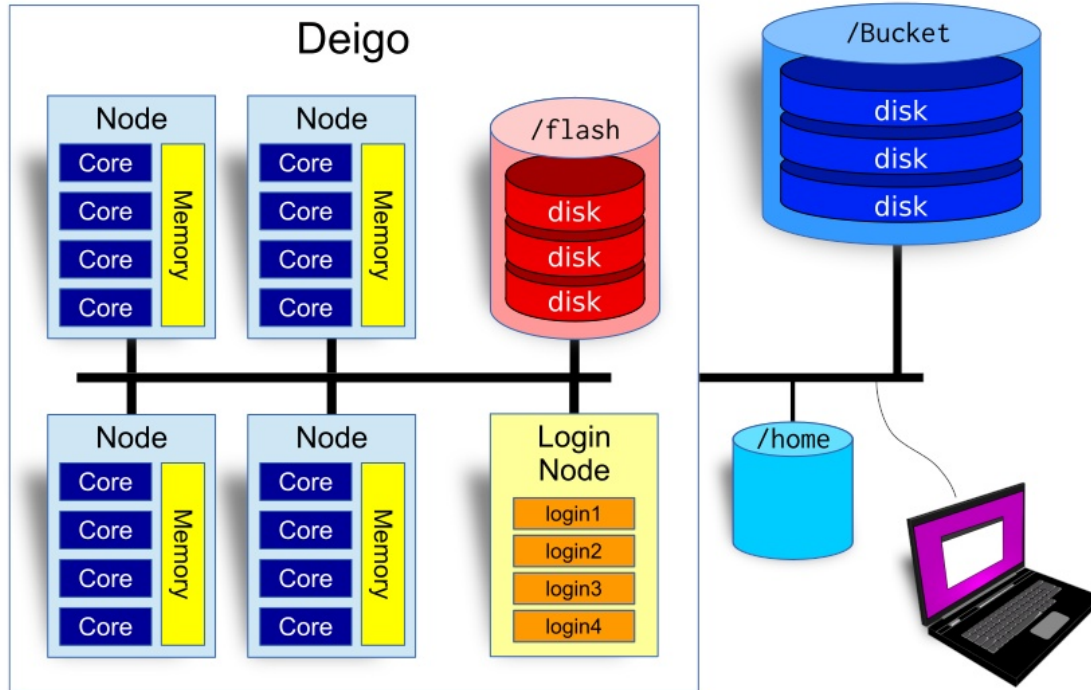
Press **once** to fill in unique parts.

Press **twice** to see matching alternatives.
This works with directories, files, programs and parameters.

The Deigo cluster



Deigo Storage

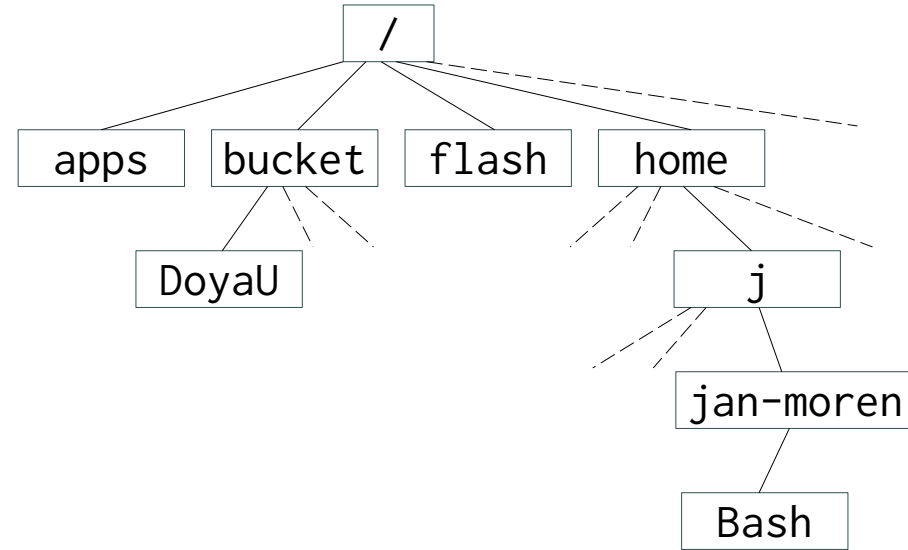


- /home Your home.
Small (<50GB), slow.
Use for: configuration files, source
- /flash In-cluster file system
Not big (10TB/unit), fast
Use for: running jobs
- /bucket Long-term storage file system
Big, backed up
read-only from computing nodes
Use for: storing data

The File System

Similar to MacOS, different from Windows

- A single tree (no "c:", "d:")
- "/" is the folder separator
- "directory" = "folder"



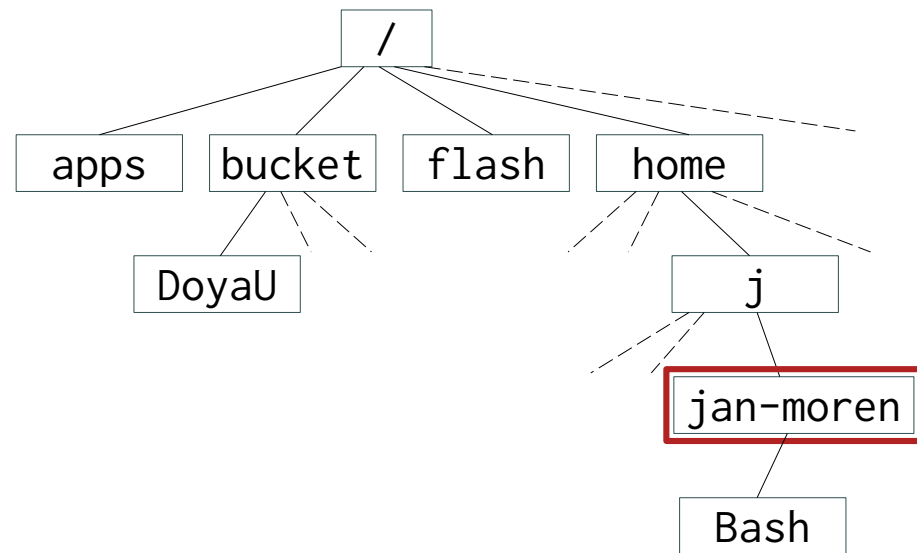
The File System

Similar to MacOS, different from Windows

- A single tree (no "c:", "d:")
- "/" is the folder separator
- "directory" = "folder"
- You are always in a directory
you start in your "home"
- See your current directory with "pwd":

```
$ pwd
```

```
/home/j/jan-moren
```



The File System

See your current directory with "pwd":

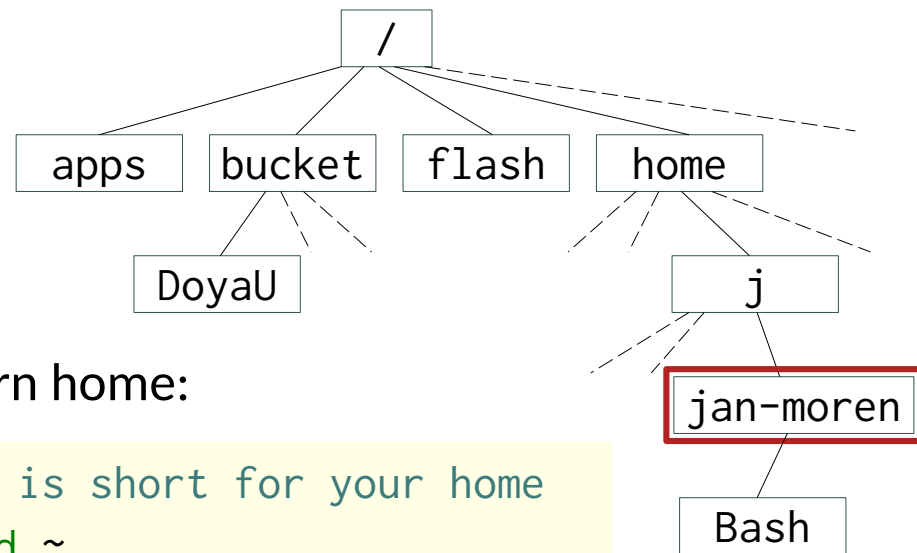
```
$ pwd  
/home/j/jan-moren
```

List current directory with "ls":

```
$ ls  
10.2 n13
```

Change directory with "cd":

```
$ cd Bash
```



Return home:

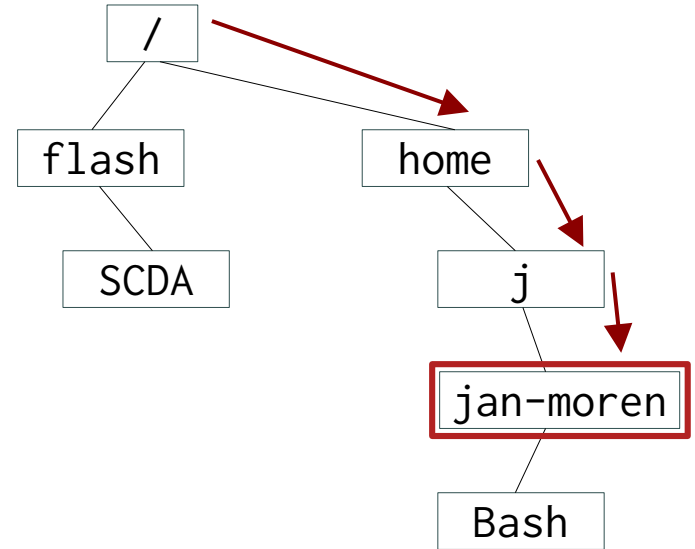
```
# ~ is short for your home  
$ cd ~  
# $HOME is also your home  
$ cd $HOME  
# just 'cd' returns you home  
$ cd
```

The File System

- "/" is the folder separator
- "directory" = "folder"
- You start in your "home"

Absolute path begins from the top with "/":

```
$ ls /home/j/jan-moren
```



The File System

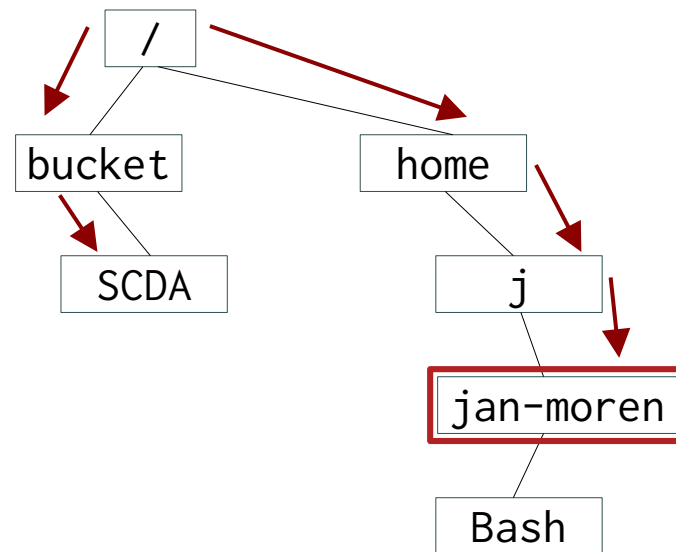
- "/" is the folder separator
- "directory" = "folder"
- You start in your "home"

Absolute path begins from the top with "/":

```
$ ls /home/j/jan-moren
```

```
$ ls /bucket/SCDA
```

try with your unit

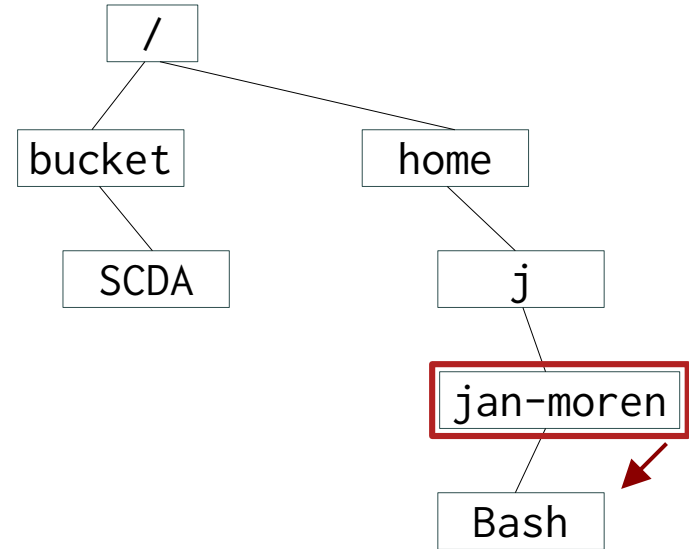


The File System

- "/" is the folder separator
- "directory" = "folder"
- You start in your "home"

Relative path begins from where you are:

```
$ ls Bash
```



The File System

- "/" is folder separator
- "directory" = "folder"
- You start in your "home"

Relative path begins from where you are:

```
$ ls Bash
```

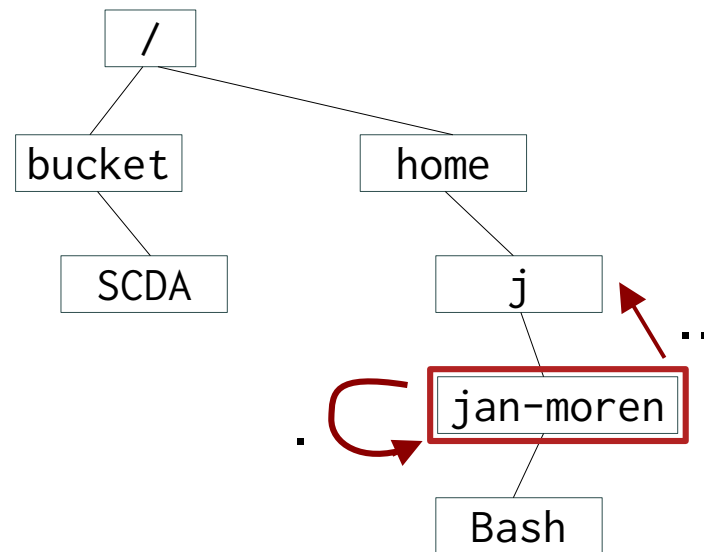
"." means here and ".." means one step up:

```
$ ls ..
```

```
# same thing:
```

```
$ ls
```

```
$ ls .
```



The File System

- "/" is folder separator
- "directory" = "folder"
- You start in your "home"

Relative path begins from where you are:

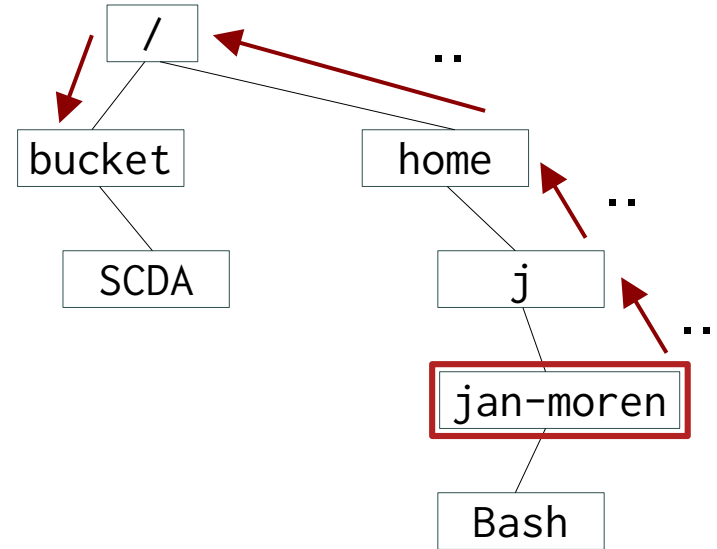
```
$ ls Bash
```

"." means here and ".." means one step up:

```
$ ls ../../../../bucket
```

```
# same thing:
```

```
$ ls ../../../../../../../bucket
```



The File System

Useful Commands

`ls <path>`

list contents of directory

`cd <path>`

go to directory

`pwd`

show your current directory

`cp <source> <dest>`

copy file or directory

`mv <source> <dest>`

move (rename) file or directory

`rm <thing>`

delete file or directory

`mkdir <dir>`

create new directory

`touch <file>`

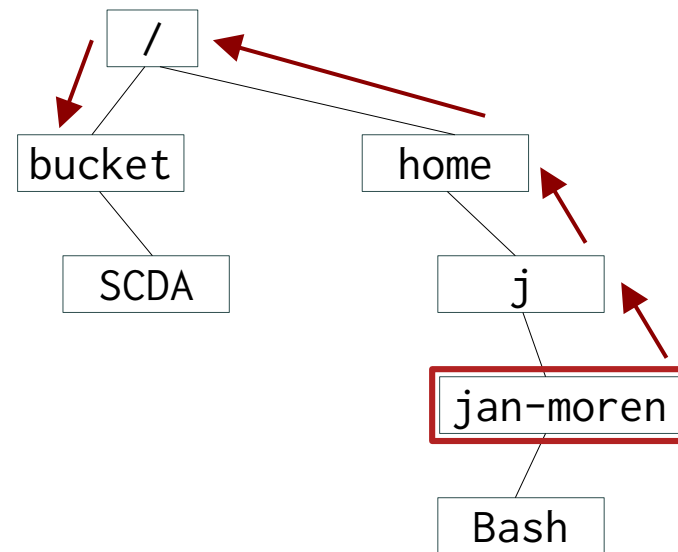
create new, empty file (or "update" a file)

`less <file>`

open and read a file

`nano <file>`

edit a file (limited, simple editor)



Wildcards

Match groups of files by name

- * stands for "any number of characters"
- ? stands for "any one character"

```
# any file ending with ".txt"
$ ls *.txt
macbeth.txt  bipp.txt      hello.txt

# a set of fasta files
$ ls name_ID????.fasta
name_ID0001.fasta  name_ID0002.fasta
```

Command line tips:

- **history** shows your command history
-  and  moves through your previous commands
- **ctrl+r** searches through your command history
- **!number** run command <number>

Help!!!!

1. Use the help options

Most (not all) commands have either
"-h" or "--help" for a short summary:

```
$ rm --help
Usage: rm [OPTION]... [FILE]...
Remove (unlink) the FILE(s).

  -f, --force    ignore nonexistent files and arguments, never prompt
  -i            prompt before every removal
  -I            prompt once before removing more than three files, or
               when removing recursively; less intrusive than -i,
...

```

Help!!!!

2. "man" pages

many applications have manual pages with terse
usage information in a standard format:

```
$ man rm
```

```
dRM(1)                                User Commands                                RM(1)

NAME
  rm - remove files or directories

SYNOPSIS
  rm [OPTION]... [FILE]...

DESCRIPTION
  This manual page documents the GNU version of rm. rm removes
  each specified file. By default, it does not remove directories.
  . . .
```

man uses "less" to show text:



move up and down

<space> page down

/ search for a string

q quit

Help!!!!

3. Search online

- Many, many pages, tutorials, forums online.
- Google is still least bad (maybe)
- Use "bash" as one keyword

**Do NOT just copy and paste
commands you find online!**

*This goes for LLMs
and coding agents too!*

- It might be **outdated**
- It might be **wrong**
- It might be **malicious**

→ Always try to understand what you're doing

Exercise

It's Your Turn!

- Here is the cluster training data directory:
/apps/share/training/Intro
 - Under that directory, find a file called "animals.txt"
1. Create a new directory called "from_intro" inside your Bash directory in your home
 2. Copy the file to your new directory
 3. Find out **how many Platypuses** we have

```
$ man <command>  
$ command --help
```

```
ls <path>  
cd <path>  
pwd
```

```
cp <source> <dest>  
mv <source> <dest>  
rm <thing>
```

```
mkdir <dir>  
touch <file>
```

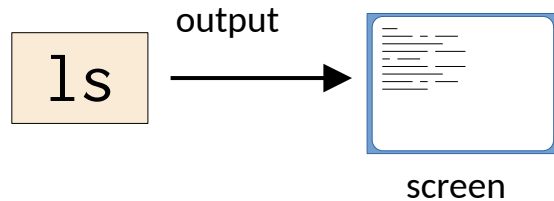
```
less <file>  
nano <file>
```

Redirection and Pipes

You want to read the too-long output of a command:

```
# List all programs in /usr/bin  
$ ls /usr/bin
```

Way too long...



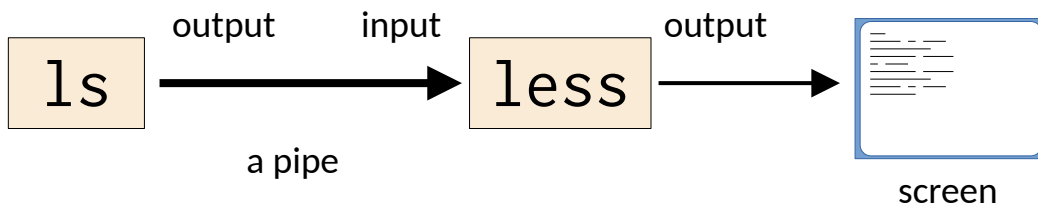
Redirection and Pipes

You want to read the too-long output of a command:

```
# List all programs in /usr/bin  
$ ls /usr/bin
```

Way too long... We want to use "less" to view it. We can do that with a *pipe*:

```
# List all programs in /usr/bin  
$ ls /usr/bin | less
```



The inputs and outputs have specific names:

input = stdin
output = stdout
error = stderr

Redirection and Pipes

You can redirect to and from a file with '<' and '>':

```
# List all programs in /usr/bin, redirect to file
$ ls /usr/bin >binfiles.txt

# Redirect the file to less
$ less <binfiles.txt
```

You can add to a file with '>>':

```
# add a monthly report to the year
$ cat September.txt >>year2023.txt
```

redirect both stdout and stderr with '&>':

```
# build some program, save the output for later:
$ make &>build.log
```

The inputs and outputs
have specific names:

input = stdin

output = stdout

error = stderr

Variables

- Variables contain values, typically text
- Set them like this:

```
# set a variable - no spaces around the "="  
$ value="42"
```

- Get the value with `${value}` or `$value`
- You can print stuff using "echo"

```
# get a variable value  
$ echo ${value}  
42  
  
# same but simpler to type (but a bit ambiguous)  
$ echo $value  
42
```

Variables

- First `${...}` expands into the *content* of the variable
- Then the line is evaluated

```
# Here's a value
$ value="42"

# Here's our line
$ echo "the number is" ${value}

# Substitute all variables with their values
$ echo "the number is" 42

# Now actually run the command
$ echo "the number is" 42
the number is 42
```

Variables

- You can use variables to create filenames or commands

But the { } brackets become important

```
# let's create a file name:  
$ value="42"  
  
# when run, ${value} is first replaced with 42:  
$ echo gene${value}a_01.fasta
```

Variables

- You can use variables to create filenames or commands

But the { } brackets become important

```
# let's create a file name:  
$ value="42"  
  
# when run, ${value} is first replaced with 42:  
$ echo gene42a_01.fasta
```

Variables

- You can use variables to create filenames or commands

But the { } brackets become important

```
# let's create a file name:  
$ value="42"  
  
# final filename  
$ echo gene42a_01.fasta  
gene42a_01.fasta
```

Variables

- You can use variables to create filenames or commands

But the { } brackets become important

```
# let's create a file name:  
$ value="42"  
  
# final filename  
$ echo gene42a_01.fasta  
gene42a_01.fasta
```

- If we used just \$value here:

```
# Now bash thinks the variable is valuea_01  
$ echo gene$valuea_01.fasta  
gene.fasta
```

Environment Variables

- Ordinary variables only visible to the script itself
- *Environment variables* visible to all child programs
- Used for various general settings

HOME	Your home directory
USER	Your user name
LANG	Your language
PATH	List of directories to look for programs
HOSTNAME	Name of the current computer / node

```
# make it an environment variable
$ export value
# see environment
$ env | less
```

Environment Variables

- Ordinary variables only visible to the script itself
- *Environment variables* visible to all child programs
- Used for various general settings

HOME	Your home directory
USER	Your user name
LANG	Your language
PATH	List of directories to look for programs
HOSTNAME	Name of the current computer / node

```
# Add a path to PATH:  
$ PATH="/new/path/name:$PATH"  
# export new PATH into environment  
$ export PATH
```

Configuration Files

Many apps can be configured with configuration files

- They're usually text format
- They are usually in your home:
 - A file beginning with a '.'
 - In an app-specific subfolder
 - In the '.config/' subfolder

Names that begin with '.'
are hidden by default

```
# the -a option shows hidden files/dirs
$ ls -a

.bash_history      # your shell history
.bash_profile     # Bash settings
.bashrc           # Bash settings
.cache/           # temporary data
.config/          # many apps settings
.inputrc          # commandline settings

.lmod.d/          # Your module cache
.matlab/          # matlab settings
.ssh/             # Your SSH folder
.vimrc            # Vim editor settings
```

An aside: Editors

We need an editor... ...a **code** editor. What is that?

Uses *only* plain text

Text editors change simple " to fancy “ and ”, plain – (minus) to fancy — and so on.

These are completely different characters, so your code breaks.

Supports syntax highlighting, autocompletion and so on

Text editors (avoid these):

- Word
- Wordpad
- TextEdit

Code editors:

- Notepad
- VSCode
- Zed
- Nano
- Vim

An aside: Editors

Lots of options on the cluster!

- **nano** - very simple, easy but limited
- **gedit** - modern, simple, mouse support but needs graphics support
- **vim** - classic, very powerful, available everywhere but difficult to learn
- **VSCode** - popular, run locally and edit on the cluster, a bit complex

Nano is already installed on MacOS and on Linux.

In MobaXterm use Notepad:

```
$ notepad
```

Nano benefits:

- Available everywhere
- Text mode – no graphics needed
- Runs fine across SSH
- Really simple

.bashrc

The main shell configuration file

- Set parameters
- create aliases
- set environment variables
- Change your prompt
- ...

File names starting with "." are normally hidden.

"ls -a" ("all") will show them

Don't:

- Don't load modules here
 - You will forget, then get confused
 - Load modules in your slurm scripts
 - Do enable unit-specific module areas
- Don't enable python environments
 - Same reason as above
- Don't print *anything*
 - When you use scp to copy data, it will break the data stream

.bashrc

The main shell configuration file

```
# Use "nano" to edit .bashrc
$ nano .bashrc

# use 'source' to re-read the file
$ source .bashrc
```

File names starting with "." are normally hidden.

"ls -a" ("all") will show them

```
# correct minor directory errors
shopt -s cdspell

# append to the history file
shopt -s histappend

# set history length
HISTSIZE=15000
HISTFILESIZE=15000

# Add a new path for programs to PATH
export PATH="/my/new/path:$PATH"

# create aliases
alias ll="ls -trlh"           # long form ls

# disable ctrl-s and ctrl-q
stty -ixon
```

Command Examples

```
# Size of all folders, in human readable
# format, sorted with largest at bottom
# du disk usage
#   -c count each folder separately
#   -h human readable output
#   ~ our home directory
# sort sorts its input
#   -h read human readable values
$ du -ch ~|sort -h

# date prints time and date
$ date
```

```
# See the newest files

# long list, with newest file at the end
# -t sorts by time
# -r reverse — newest at the bottom
# -l long format, with lots of detail
# -h print file sizes human readable
$ ls -trlh

# time measures time taken
$ time sleep 5
```

More Examples

sed edits text

```
# Replace 'Macbeth' with 'Bob':  
# 's' is the search and replace command  
# 'g' replaces all matches on a line  
# 'i' makes the search case-insensitive  
$ sed "s/macbeth/Bob/ig" macbeth.txt
```

AWK is a language, does lots of stuff

```
# print first and second word with comma  
# -e says script is the next argument  
$ awk -e '{print $1 ", " $2}' macbeth.txt
```

"grep" finds text

```
# find Platypus in our animal list  
$ grep "Platypus" animals.txt  
  
# Commands that begin with b in /usr/bin  
$ ls /usr/bin | grep "^b"  
  
# count lines with "Macbeth" in macbeth.txt  
# 'wc' counts characters, words and lines  
$ grep "Macbeth" macbeth.txt | wc -l  
  
# print lines without "Macbeth"  
# -v inverts the search  
$ grep -v "Macbeth" macbeth.txt
```

Regular Expressions

Immensely useful, so a very quick primer

It's a language for *matching patterns*

- Typically used on text:
documents, filenames, source code, data ...

Use to:

- Search
 - Replace
 - Delete
 - Count
- Many tools have Regular Expressions built in
 - Often called “Regexps” for short

character	Any normal character matches itself
^ and \$	Match beginning, end of line
.	Match any character
?	Matches 0 or 1 of a character - may need \"?\"
*	Matches any number of a character
[...]	List of possible characters. ^ first negates it.
... ...	Match either or - may need \" \"

grep uses regular expressions

```
# Find all mentions of "Macbeth" or "Macduff" in macbeth.txt
$ grep "Macbeth\\|Macduff" macbeth.txt
...
```

character	Any normal character matches itself
^ and \$	Match beginning, end of line
.	Match any character
?	Matches 0 or 1 of a character - may need \"\?"
*	Matches any number of a character
[...]	List of possible characters. ^ first negates it.
... ...	Match either or - may need \"\ \" and

Macbeth	"Macbeth" anywhere
M.cbeth	"Macbeth", "Mocbeth", "M#cbeth"...
^M.*h	M at the start of a line, any number of any char, then h
Mac[^]*	"Macbeth", "Macduff", "Macaroni", "Macaronis" ...
[Mm]acbeth	"Macbeth" or "macbeth"
Mac\\(beth\\ duff\\)	"Macbeth" or "Macduff"
^\$	an empty line (start and end with nothing inbetween)

character	Any normal character matches itself
^ and \$	Match beginning, end of line
.	Match any character
?	Matches 0 or 1 of a character - may need '\?'
*	Matches any number of a character
[...]	List of possible characters. ^ first negates it.
... ...	Match either or - may need '\ '

Useful grep options:

-i	Ignore case: "Macbeth", "macbeth", "MACBETH", "MacBeTh"
-w	Find whole words only.
-v	Invert match: print lines that <i>don't</i> match your regexp
-o	Print only the matching part, not the rest of the line
-Cn	Print n lines before and after each match

Questions?

Find our "introduction to Bash" page in our documentation:

<https://scda-guide.oist.jp/command-line-introduction-bash>

Our "advanced Bash" page covers useful tools and patterns:

<https://scda-guide.oist.jp/advanced-bash>