Intermediate Linux

Andreas W. Götz, PhD

Material by Robert Sinkovits, PhD

Director of Education

San Diego Supercomputer Center

SDSC Summer Institute
Monday, August 7, 2023

This session is intended for anyone who …

Works in a Linux environment and wants to acquire additional skills to more effectively use advanced cyberinfrastructure (CI)

- Linux philosophy
- Filesystem Hierarchy Standard
- Symbolic links and hard links
- Wildcards and globbing
- Aliases
- Environment variables
- Configuration files (~/.bashrc, ~/.bash_profile, /etc/bashrc, …)
- Pseudo-filesystems
- Bash scripting

Prerequisites

We assume that you already know the Linux basics, such as

- Listing files (ls)
- Removing files (rm)
- Navigating directories (cd)
- Listing file contents (cat and optionally head, tail, less, more)
- Ability to use one or more standard editors (vi, vim, emacs, nano)
- Relative and absolute paths

A note on *intermediate* Linux

In the Linux world, intermediate is in the eye of the beholder. Everything that we are covering in this session would be considered basic or elementary by Linux systems administrators.

But if you are attending this session, you are probably not a sys admin. Rather, it is more likely you are a student, educator or researcher who uses advanced CI as part of your work or studies. In which case, these topics really are intermediate in the context of Linux users instead of Linux professionals.

Filesystem Hierarchy Standard (FHS)

The FHS describes the conventional layout of a Linux system. Although there are some variations among different Linux distributions (distros), they tend to be pretty minor and mostly related to the /lib, /bin, /usr/lib and /usr/bin directories.

As a regular user (i.e., not a sys admin) you don't need to know too much about the FHS. We cover it here to round out your knowledge and make the Linux environment a little less mysterious.

- Overviews
  - https://en.wikipedia.org/wiki/Filesystem_Hierarchy_Standard
  - https://www.linuxjournal.com/content/filesystem-hierarchy-standard
- Deep dives
  - https://tldp.org/LDP/Linux-Filesystem-Hierarchy/html/index.html
  - https://refspecs.linuxfoundation.org/FHS_3.0/fhs/index.html

Filesystem Hierarchy Standard (FHS)

- **/home** contains user home directories
- **/bin** and **/usr/bin** traditionally contain essential and non-essential command binaries, respectively, but these are often merged. For example, on CentOS Linux /bin is a symbolic link to /usr/bin
- **/sbin** and /**usr/sbin** contain binaries for commands needed mostly by sys admins. Like /bin and /usr/bin, these are often merged
- **/lib** and **/usr/lib** contain libraries needed by the Linux commands (often merged)
- **/opt** is where software was traditionally installed, but with the rising popularity of package managers like Spack, this directory is not used much anymore

Filesystem Hierarchy Standard (FHS)

- **/etc** contains configuration files for Linux shells, schedulers, networking, accounts, package managers and others
- **/root** home directory for the root user
- **/var** log files
- **/dev** device files for terminals, disks, networking, etc.
- **/proc** and **/sys** are pseudofile systems (we will talk about this later) that contain information about running processes and hardware
- **/tmp** stores temporary files

Not everything fits into the FHS

Especially in high-performance computing environments, not everything fits into the FHS. This can include additional file systems and software installed using package managers. For example, on SDSC's Expanse resource

- Lustre parallel file systems live under **/expanse/projects** and **/expanse/scratch**
- Qumulo (hybrid cloud storage) is mounted at **/expanse/projects**
- Software installed using the Spack package manager are found under **/cm/shared/apps/spack**

Symbolic links and hard links

Symbolic links and hard links give you a way to refer to a file or an inode by a different name. They are useful when you want to avoid writing out a full or relative path name every time or if you want to have a generic executable name refer to a specific version.

We will discuss the difference between symbolic links (also known as soft links) and hard links in the next few slides along with some background on inodes. Both types of links are created with the ln command

```
$ ln -s file1 soft-file1 # Creating a soft link
$ ln file2 hard-file2    # Creating a hard link
```

Redirection and pipes

Most Linux commands write to standard output (stdout), which is the terminal if you are working on the command line. Output can be redirected to a file using the > symbol, with >> used to append output to a file.

```
$ date > outfile                   # Write date to file
$ cat outfile
Tue Jun 28 15:31:20 PDT 2022

$ date > outfile                   # Overwrites content of file
$ cat outfile
Tue Jun 28 15:31:25 PDT 2022

$ date >> outfile                  # Appends output to file
$ cat outfile
Tue Jun 28 15:31:25 PDT 2022
Tue Jun 28 15:31:30 PDT 2022
```

Redirection and pipes

A pipe (|) takes stdout from one command and directs it to stdin of a second command. An arbitrary number of commands can be connected using pipes.

```
# Count number of files in current directory
$ ls -1 | wc -l
13

# List the three longest running jobs on Expanse node exp-2-09
$ squeue | grep exp-2-09 | head -3
13777427    shared NGBW-JOB   cipres  R 5-06:14:20      1 exp-2-09
13816762    shared NGBW-JOB   cipres  R 5-03:14:11      1 exp-2-09
13816766    shared NGBW-JOB   cipres  R 5-03:13:39      1 exp-2-09
```

Redirection and pipes

Stdin can also be redirected. This is sometimes useful if you want to prevent a command from listing the name of the input file. In the example below we avoid having to pipe output from wc into the cut utility.

```
$ wc -l file1.txt                  # Output includes file name
9 file1.txt

$ wc -l file1.txt | cut -d' ' -f1  # Output contains just the line count
9

$ cat file1.txt | wc -l file1.txt  # Output contains just the line count
9

$ wc -l < file1.txt                # Output contains just the line count
9
```

Symbolic links and hard links - inodes

An inode is a data structure that stores information about a filesystem object, such as a regular file or a directory. It contains:

- Permissions
- File size
- Ownership
- Timestamps
- Location of data, but not the actual data.

A file is a link to an inode and a hard link is just another file that links to the same inode. An inode is only deleted when all the links to it have been deleted. If the original file is deleted, the data can still be accessed using the hard link.

A symbolic link is another name for the file. If the original file is deleted or moved, the symbolic link still exists, but can no longer be used to access the data.

Symbolic links and hard links example

```
$ ls
file1 file2
$ ln file1 file1-hard
$ ln -s file2 file2-soft

$ ls -i # -i option lists inode
157076717 file1
157076717 file1-hard
157076716 file2
157076768 file2-soft@

$ rm file1 file2
$ cat file1-hard
Contents of file1
$ cat file2-soft
cat: file2-soft: No such file or directory
```

Symbolic links and hard links – which should I choose?

You'll generally work with symbolic links since they let you span file systems and can be created even if the file being linked to does not yet exist.

```
$ ln file3 file3-hard
ln: file3: No such file or directory
$ ln -s file3 file3-soft

$ ln /expanse/lustre/scratch/sinkovit/file4 file4-hard
ln: failed to create hard link 'file4-hard' =>
'/expanse/lustre/scratch/sinkovit/file4': Invalid cross-device link
$ ln -s /expanse/lustre/scratch/sinkovit/file4 file4-soft

$ ls
file1  file1-hard  file2  file2-soft  file3-soft  file4-soft
```

Wildcards and globbing

Wildcards let you express a pattern for matching multiple file names using any combination of the following

- ? matches any single character
- * matches any string (including empty string)
- […] matches multiple characters or ranges of characters

Globbing is the operation that expands the pattern into a list of files. Globbing is done automatically by many Linux utilities that operate on files such as ls, rm, mv, cat, head, tail and file

Wildcards and globbing using ‘*’

```
$ ls
abcde.txt    cdefg.txt    efghi.txt    file2.dat    fileb.dat    ghijk.txt
axcde.exe    cxefg.exe    exghi.exe    file3.dat    filec.dat    gxijk.exe
bcef.txt     degh.txt     fgij.txt     file4.dat    filed.dat    hikl.txt
bxdef.exe    dxfgh.exe    file1.dat    filea.dat    fxhij.exe    hxjkl.exe

$ ls file*   # file names beginning with ‘file’
file1.dat    file3.dat    filea.dat    filec.dat
file2.dat    file4.dat    fileb.dat    filed.dat

$ ls f*      # file names beginning with ‘f’
fgij.txt     file2.dat    file4.dat    fileb.dat    filed.dat
file1.dat    file3.dat    filea.dat    filec.dat    fxhij.exe

$ ls *ghi*   # file names containing ‘ghi’
efghi.txt    exghi.exe    ghijk.txt
```

Wildcards and globbing using '?'

```
$ ls
abcde.txt     cdefg.txt     efghi.txt     file2.dat     fileb.dat     ghijk.txt
axcde.exe     cxefg.exe     exghi.exe     file3.dat     filec.dat     gxijk.exe
bcef.txt      degh.txt      fgij.txt      file4.dat     filed.dat     hikl.txt
bxdef.exe     dxfgh.exe     file1.dat     filea.dat     fxhij.exe     hxjkl.exe

$ ls file?.dat  # 'file' + any character + '.dat'
file1.dat     file3.dat     filea.dat     filec.dat
file2.dat     file4.dat     fileb.dat     filed.dat

$ ls a?cde.*.   # 'a' + any character + 'cde.' + any string
abcde.txt     axcde.exe

$ ls ?????.?x?  # Any 5 characters + '.' + any character + 'x' + any character
abcde.txt     bxdef.exe     cxefg.exe     efghi.txt     fxhij.exe     gxijk.exe
axcde.exe     cdefg.txt     dxfgh.exe     exghi.exe     ghijk.txt     hxjkl.exe
```

Wildcards and globbing using '[…]'

```
$ ls
abcde.txt     cdefg.txt     efghi.txt     file2.dat     fileb.dat     ghijk.txt
axcde.exe     cxefg.exe     exghi.exe     file3.dat     filec.dat     gxijk.exe
bcef.txt      degh.txt      fgij.txt      file4.dat     filed.dat     hikl.txt
bxdef.exe     dxfgh.exe     file1.dat     filea.dat     fxhij.exe     hxjkl.exe

$ ls file[13].dat    # multiple matches using list of integers
file1.dat     file3.dat

$ ls file[1-3].dat   # multiple matches using range of integers
file1.dat     file2.dat     file3.dat

$ ls [ac]x*.*        # multiple matches using list of letters
axcde.exe     cxefg.exe

$ ls [a-c]x*         # multiple matches using range of letters
axcde.exe     bxdef.exe     cxefg.exe
```

Aliases

Aliases are just shortcuts for Linux commands. They're useful for keeping you out of trouble (e.g., accidentally deleting all files) and abbreviating complex commands

```
# Staying out of trouble (e.g., ensure “rm *” won’t delete everything)
$ alias rm='rm -i’
$ which rm
alias rm='rm -i'
        /usr/bin/rm
```

```
# Abbreviating a long command (customized listing of Slurm partitions)
$ alias partitions='sinfo -o "%15P %6a %6D %11s %15F %7c %8m %10l %10L %10G”’
$ which partitions
alias partitions='sinfo -o "%15P %6a %6D %11s %15F %7c %8m %10l %10L %10G"'
        /cm/shared/apps/slurm/current/bin/sinfo
```

Aliases

Aliases can be listed by typing alias without arguments

```
$ alias
alias partitions='sinfo -o "%15P %6a %6D %11s %15F %7c %8m %10l %10L %10G"'
alias reservations='sinfo -T'
alias rm='rm -i'
alias spacktivate='spack env activate'
alias vi='vim'
alias int1core='srun --partition=shared  --pty --account=use300 --nodes=1 --
ntasks-per-node=1   -t 01:00:00 --wait=0 --export=ALL /bin/bash'
alias int1node='srun --partition=compute --pty --account=use300 --nodes=1 --
ntasks-per-node=128 -t 01:00:00 --wait=0 --export=ALL /bin/bash'
```

Command history

You can inspect your command history using `history`. You can navigate this history with the arrow keys so you can execute or modify a previous command so you do not have to remember and/or type everything again. You can also search for previous commands using emacs key bindings (ctrl+R for reverse search) or switch your bash shell to vi key bindings. There are special commands to execute previous commands:

```
# inspect command history
$ history
-- snip --
427 cd $HOME/test
428 ls file*.txt
429 rm file777.txt
430 echo "Hello"
```

```
# execute last command
$ !!
echo "Hello"
Hello
```

```
# execute last command that started
# with "ls"
$ !ls
ls file*.txt
file1.txt file2.txt file776.txt
```

```
# execute a specific command
$ !428
ls file*.txt
file1.txt file2.txt file776.txt
```

Navigating between directories

The pushd and popd commands are very useful to navigate between directories. You can use pushd to move into a different directory and store the current directory on a stack. You can then later use popd to move back to the original directory. This also allows you to move to different directories inbetween.

```
$ pwd
/scratch/agoetz/n2o5-scattering
# change directory and store current directory
$ pushd $HOME/foo/bar
~/foo/bar /scratch/agoetz/n2o5-scattering
$ pwd
/home/agoetz/foo/bar
# move to another directory
$ cd ../
# move back to the original directory
$ popd
/scratch/agoetz/n2o5-scattering
$ pwd
/scratch/agoetz/n2o5-scattering
```

Shell and environment variables

Shell and environment variables allow you to customize your environment and control how applications behave on a Linux system.

- **Shell variables** are only known within the current shell
- **Environment variables** are known globally and are inherited by processes and shells that are launched by the current shell. More often than not, you probably want to use environment rather than shell variables.

- Shell variables are set using the KEY=value[:value2:[:value3]…] syntax; space is not allowed around the equal sign and by convention the KEY is capitalized
- Shell variables are made into environment variables using the export command
- Value of shell/environment variables accessed using $KEY

Shell and environment variables

```
$ KEY1="abcd"            # set shell variable KEY1
$ KEY2="efgh"            # set shell variable KEY2
$ export KEY2            # make KEY2 an environment variable
$ export KEY3="ijkl"     # set environment variable KEY3
$ echo $KEY1             # display KEY1
abcd                     #
$ echo $KEY2             # display KEY2
efgh                     #
$ echo $KEY3             # display KEY3
ijkl                     #

$ /bin/bash              # launch a new shell
$ echo $KEY1             # display KEY1
                         # KEY1 was only known in the parent shell
$ echo $KEY2             # display KEY2
efgh                     # KEY2 was inherited
$ echo $KEY3             # display KEY3
ijkl                     # KEY3 was inherited
```

Shell and environment variables

Another way to assign shell and environment variables is with the declare command, using the -x flag to export. Another common option is -r to make the variable read only (within the current shell)

```
$ declare VAR1="abc"       # shell variable
$ declare -x VAR2="def"    # environment variable
$ echo $VAR1
abc
$ echo $VAR2
def

$ bash
$ echo $VAR1
                           # value not known by child shell
$ echo $VAR2
def                        # value exported to child shell
```

Shell and environment variables

All environment variables can be displayed using the env command. Partial output is shown below

```
$ env
PATH=/cm/shared/apps/sdsc/1.0/bin:/cm/shared/apps/sdsc/1.0/sbin:/cm/shared
/apps/slurm/current/sbin:/cm/shared/apps/slurm/current/bin:/home/sinkovit/
spack/bin:/usr/local/bin:/usr/bin:/usr/local/sbin:/usr/sbin:/home/sinkovit
/.local/bin:/home/sinkovit/bin:/home/sinkovit/anaconda3/bin:/opt/AMDuProf_
3.3-462/bin
LIBRARY_PATH=/cm/shared/apps/slurm/current/lib64/slurm:/cm/shared/apps/slu
rm/current/lib64
MANPATH=/cm/shared/apps/slurm/current/man:/usr/share/lmod/lmod/share/man::
/usr/local/share/man:/usr/share/man:/cm/local/apps/environment-
modules/current/share/man
LOGNAME=sinkovit
SHELL=/bin/bash
HOME=/home/sinkovit
```

Linux configuration files

Linux has a number of configuration files for customizing your environment: (/etc/profile, /etc/bashrc, ~/.bash_profile, ~/.bashrc and others)

To make things even more confusing, there is a hierarchy of scripts, scripts can execute (source) other scripts and some scripts only apply to login shells. Fortunately, you can usually get by with just two general rules

- **Customize environment variables in ~/.bash_profile**
- **Put aliases and functions in ~/.bashrc**

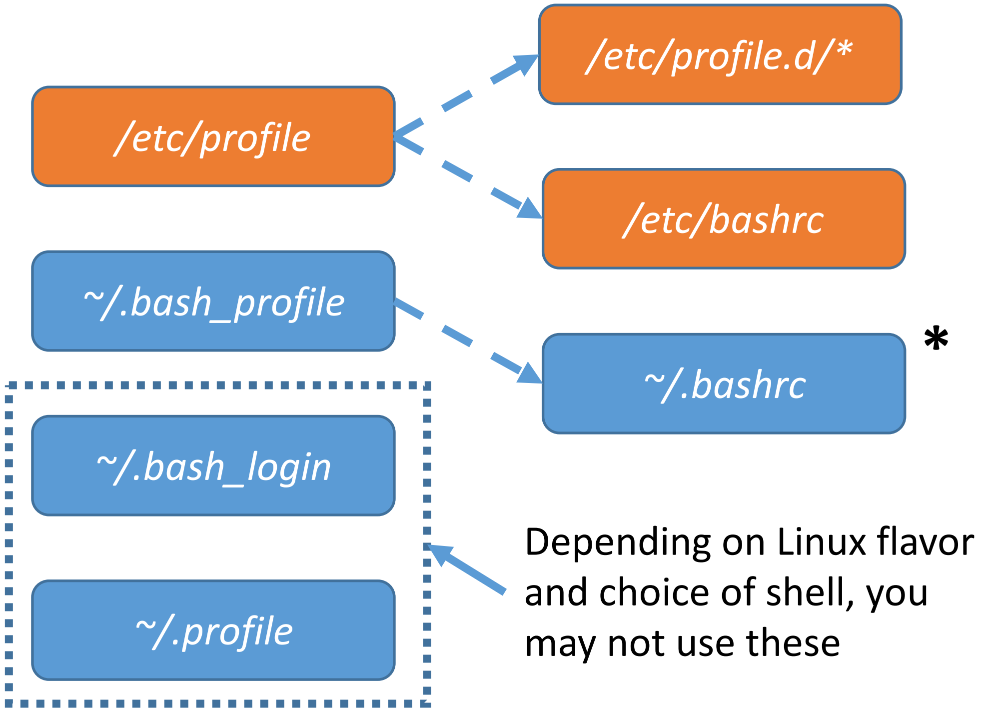
Interactive login, interactive non-login and non-interactive shells

- **Interactive shell** is one that reads from and writes to the user's terminal. *If you can enter commands at the command line and see output, you're in an interactive shell.*
  - **Interactive login shell** is launched using ssh, locally or when new shell is launched with the --login option. *This is where you'll usually work.*
  - **Interactive non-login shell** launched from a login shell by executing bash at the command line or opening new terminal from Linux desktop.

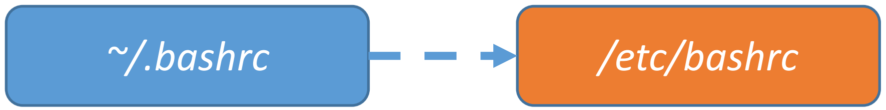
- **Non-interactive shell** is not associated with a terminal. Usually launched by a user executing a script or associated with automated process.

Order of execution

Interactive login shell



Interactive non-login shell



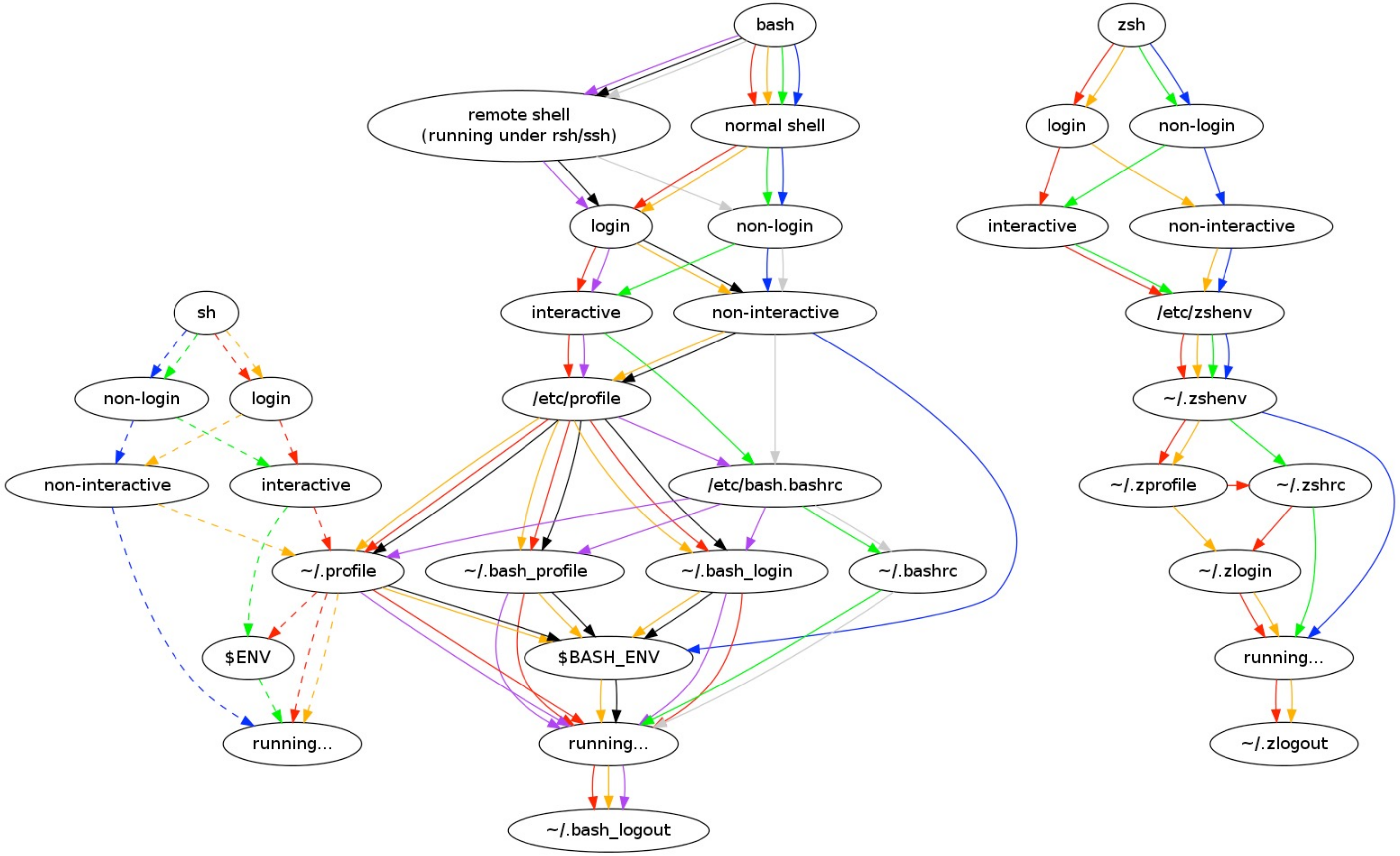
Legend

You don't need to worry about anything in the orange boxes. These are maintained by the sys admins.

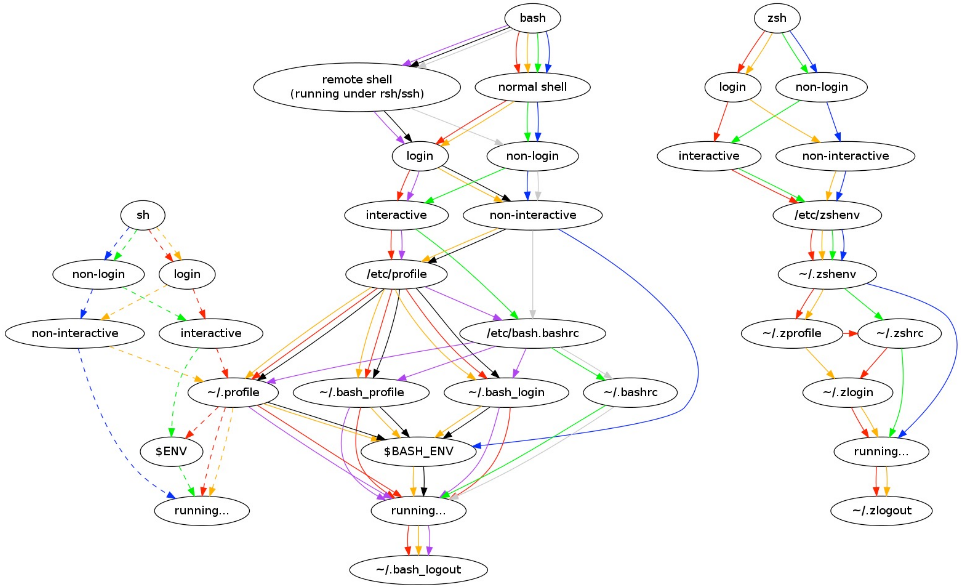
* Although ~/.bashrc normally sources /etc/bashrc, the latter contains logic to ensure that its content is not executed twice
See https://www.gnu.org/software/bash/manual/html_node/Bash-Startup-Files.html for more details on order

The bad news is …

If you do internet searches for Linux configuration files, you'll probably come across figures that look like this



But the good news is …



If you're using bash and you're not a sys admin, you probably just need to worry about two files

~/.bash_profile

~/.bashrc

Configuration files (~/.bash_profile)

Use ~/.bash_profile for commands that should be run only once, such as customizing environment variables (e.g., $PATH)

```
# .bash_profile
# Get the aliases and functions
if [ -f ~/.bashrc ]; then
. ~/.bashrc
fi

# User specific environment and startup programs
PATH=$PATH:$HOME/.local/bin:$HOME/bin
PATH=$PATH:$HOME/anaconda3/bin
PATH=$PATH:/opt/AMDuProf_3.3-462/bin
export PATH
```

This will usually be in your .bash_profile by default and sources your .bashrc file
Do not edit this part!

The first line will usually be present by default, subsequent line added to customize environment

Configuration files (~/.bashrc)

Use ~/.bashrc for commands that should run for every new shell

```
# .bashrc
# Source global definitions
if [ -f /etc/bashrc ]; then
        . /etc/bashrc
fi

alias rm='rm -i'
alias cp='cp -i’
reservations='sinfo -T’

module load slurm
module load sdsc
```

This will usually be in your .bashrc by default and sources /etc/bashrc
Do not edit this part!

Most often used for aliases and loading modules that you will want in every shell

Pseudo filesystems

Following the Linux philosophy of everything is a file, Linux distros provide pseudo filesystems that allow you to access dynamically generated content using the same tools that you use to access regular files. These are normally found in the **/proc** and **/sys** directories.

```
$ ls -l /proc/cpuinfo
-r--r--r-- 1 root root 0 Jun 10 14:42 /proc/cpuinfo # Note zero file size

$ less /proc/cpuinfo
processor        : 0
vendor_id        : AuthenticAMD
cpu family       : 23
model            : 49
model name       : AMD EPYC 7742 64-Core Processor
stepping         : 0
...
```

Shell scripting - introduction

Shell scripts are programs that are executed by the Linux shell. They can contain comments, simple commands and programming constructs for loops, branches, variable assignment, etc. Scripts can be run in three ways

- make the script executable (e.g. chmod 755 script.sh), then ./script.sh
- source script.sh
- . script.sh

We won't have time to go too deeply into shell scripting, but will show you a few of the essentials and then point you to additional resources.

Shell scripting - introduction

Shell scripts can be as simple as a list of Linux commands. The first line ("#!" is pronounced "shebang") defines the shell that will be used. It's a good idea to set this, but if you forget you'll normally get the bash shell. Note that on Linux systems /bin/sh is a symbolic link to bash

```
#!/bin/sh
# Script to execute head, tail and wc
echo "-- First two lines of file1.txt"
head -2 file1.txt
echo
echo "-- Last two lines of file1.txt"
tail -2 file1.txt
echo
echo "-- Number of lines in file1.txt"
wc -l file1.txt
```

```
$ . simple.sh
-- First two lines of file1.txt
Line 1 of file.txt
Line 2 of file.txt

-- Last two lines of file1.txt
Line 8 of file.txt
Line 9 of file.txt

-- Number of lines in file1.txt
9 file1.txt
```

Shell scripting - loops

Loops let you to iterate over the items in a list. The list can be defined in multiple ways, including a whitespace separated set of items, wildcard expansion or the backtick-captured output from a executing a command. Alternative to backticks is $()

```
for file in `ls -1 file*.txt`
do
    wc -l ${file} # get line count
done
```

```
for filename in file*.txt
do
    wc -l ${filename} # get line count
done
```

```
for x in file1.txt file2.txt file3.txt
do
    wc -l ${x} # get line count
done
```

```
for file in $(ls -1 file*.txt)
do
    wc -l ${file} # get line count
done
```

```
$ ./loop1.sh
9 file1.txt
5 file2.txt
6 file3.txt
```

All four versions produce same output, assuming directory contains file[123].txt

Shell scripting – iterating over a range of numbers

If you need to iterate over a range of numbers, you can use one of two approaches, either with the seq command or using a curly bracket notation. Note that if you define a stride, it is the second argument for the seq command, but the third argument in the curly bracket notation.

```
# concatenate content of file[1-3].txt
for i in $(seq 1 3)
do
    cat file${i}.txt >> $out
done
```

```
# concatenate content of file[1-3].txt
for i in {1..3}
do
    cat file${i}.txt >> $out
done
```

```
# concatenate content of even numbered
# files, counting down from 10 to 0
for i in $(seq 10 -2 0)
do
    cat file${i}.txt >> $out
done
```

```
# concatenate content of even numbered
# files, counting down from 10 to 0
for i in {10..0..-2}
do
    cat file${i}.txt >> $out
done
```

Shell scripting – command line arguments

Shell scripts can accept command line arguments, with $1, $2, $3 … storing the first, second, third … arguments ($0 stores the name of the script). Using command line arguments can make your scripts much more flexible

```
# Print character, word and line count in file
# Usage: counter.sh [FILE]

# wc returns character (-c), word (-w) and
# line (-l) counts; cut splits the output on
# spaces (-d' ') and prints the first field (-f1)

c=`wc -c $1 | cut -d' ' -f1`
w=`wc -w $1 | cut -d' ' -f1`
l=`wc -l $1 | cut -d' ' -f1`

echo "$1 contains"
echo "  $c characters"
echo "  $w words"
echo "  $l lines"
```

```
$ ./counter.sh file1.txt
file1.txt contains
  180 characters
  36 words
  9 lines

$ ./counter.sh file2.txt
file2.txt contains
  100 characters
  20 words
  5 lines
```

Shell scripting – command line arguments

This script is the same as the previous slide except that we used input redirection to simplify capturing the character, word and line counts and we use $() in place of backtics.

```
# Print character, word and line count in file
# Usage: counter.sh [FILE]

# wc returns character (-c), word (-w) and
# line (-l) counts

c=$(wc -c < $1)
w=$(wc -w < $1)
l=$(wc -l < $1)

echo "$1 contains"
echo "  $c characters"
echo "  $w words"
echo "  $l lines"
```

```
$ ./counter.sh file1.txt
file1.txt contains
  180 characters
  36 words
  9 lines

$ ./counter.sh file2.txt
file2.txt contains
  100 characters
  20 words
  5 lines
```

Shell scripting – conditional statements

```
if [ TEST ] ; then
    -- statements --
fi
```

Simple IF construct. Shell scripts are very picky about required white space around brackets

```
if [ TEST ]; then
    -- statements --
else
    -- statements --
fi
```

IF-ELSE construct

```
if [ TEST1 ]; then
    -- statements --
elif [ TEST2 ]; then
    -- statements --
else
    -- statements --
fi
```

IF-ELIF-ELSE construct. Note that each elif test must be followed by ‘; then’

Shell scripting – conditional statements

Variables containing numerical (integer) values can be compared using -lt, -le, -gt, -ge, and -eq

```
if [ $1 -lt $2 ]; then
    echo "$1 is less than $2"
fi
if [ $1 -le $2 ]; then
    echo "$1 is less than or equal to $2"
fi
if [ $1 -gt $2 ]; then
    echo "$1 is greater than $2"
fi
if [ $1 -ge $2 ]; then
    echo "$1 is greater than or equal to $2"
fi
if [ $1 -eq $2 ]; then
    echo "$1 is equal to $2"
fi
```

```
$  ./num-comp.sh 2 2
2 is less than or equal to 2
2 is greater than or equal to 2
2 is equal to 2

$ ./num-comp.sh 2 3
2 is less than 3
2 is less than or equal to 3

$ ./num-comp.sh 3 2
3 is greater than 2
3 is greater than or equal to 2
```

Shell scripting – conditional statements

Strings can be compared using ==, !=, \<, \> with the comparisons done using lexicographical (ASCII) order. Integer literals are treated as strings in this context

```
if [ $1 == $2 ]; then
    echo "$1 is equal to $2"
fi

if [ $1 != $2 ]; then
    echo "$1 is not equal to $2"
fi

if [ $1 \< $2 ]; then
    echo "$1 is less than $2"
fi

if [ $1 \> $2 ]; then
    echo "$1 is greater than $2"
fi
```

```
$ ./str-comp.sh abc def
abc is not equal to def
abc is less than def

$ ./str-comp.sh def abc
def is not equal to abc
def is greater than abc

$ ./str-comp.sh abc abc
abc is equal to abc

$ ./str-comp.sh 123 45
123 is not equal to 45
123 is less than 45
```

Shell scripting – conditional statements

Conditional statements can test if a file exists (-e), is a regular file (-f) or is a directory (-d). See https://tldp.org/LDP/abs/html/fto.html for full list of options

```
for file in file1.txt tempdir file4.txt
do
    if [ -e $file ]; then
        echo -n "$file exists and is "
        if [ -f $file ]; then
            echo "a regular file"
        elif [ -d $file ]; then
            echo "a directory"
        else
            echo "not regular file or directory"
        fi
    else
        echo "$file does not exist"
    fi
done
```

```
$ ./file-test.sh
file1.txt exists and is a regular file
tempdir exists and is a directory
file4.txt does not exist
```

Summary

Learning intermediate Linux skills will help you to become a more effective user of advanced cyberinfrastructure. This tutorial provides you with the tools to customize your environment, automate tasks and construct simple workflows.

While we covered most of what you should need to know, there are many resources on a wide range of topics in case you have to go deeper

- Bash scripting cheat sheet: https://devhints.io/bash
- Advanced bash scripting: https://tldp.org/LDP/abs/html/index.html
- Forums and tutorials: https://www.linux.org/forums/