

Introduction to Linux shell

Contents

Command Line	3
Introduction	3
Secure shell.....	3
Data storage	6
Exercises	9
Getting around	9
Exercises	13
Technicalities.....	16
Transferring files	19
Exercises	21
Composition.....	22
Introduction.....	22
Viewing and editing	23
Exercises	26
Regular expressions and globing	28
Regular expressions and globing.....	28
Redirection and pipes	32
Exercises	35
Automation.....	36
Introduction.....	36
Scripting.....	37
Exercises	42
Submitting jobs.....	45
Loops	49
Exercises	51
UNIX Reference	52
Key features.....	52
File system	52
Root (.....	53
/	53
)	53
User (.....	55
/usr	55
)	55
User Local (.....	56
/usr/local	56

)	56
Alliance supercomputers.....	57
Devices.....	57
Commands.....	57
Help	58
Directories.....	59
Files	60
Permissions	61
View Files	63
Comparison	66
Searching	67
Process	68
Remote	69
Other.....	70
Editors.....	71
Vi	72
Emacs.....	74
Command line.....	75
Arguments	76
Quoting	78
Environment	79
Input and Output.....	80
Status.....	81
Job Control	81
Multiple Commands	81
Scripting.....	83
Parameters.....	83
Programming.....	84
Regular expressions	87
Anchoring	88
Characters	88
Combining	88
Repetition.....	89
Replacement.....	89

Command Line

Introduction

Completion requirements

The [command line](#) is a text interface on a computing system into which we type [commands](#) to tell the computer what to do. In times past this would be done via a dedicated monitor and keyboard combination called a terminal. This has been replaced with a program that emulates these old system and provides us a window into which we can type.

The [command line](#) program that we will be learning is the Bash shell. It is the descendant of a long line of shell programs, and it is specialized to help us manipulate our file, start programs, and automate tasks involving these things. It is also the only way to run jobs on the Canadian supercomputers.

I will be demonstrating everything on the supercomputers in order to help you become more familiar with them. Almost everything in this course does not require the supercomputers though, so there is no need for you to use them unless you would like the practice.

Under both Linux and Mac OS X, you can start up a Bash [command line](#) session by starting a terminal program (search for terminal in your applications menu). The default [command line](#) interface in Windows is the older Command Prompt or the newer Power Shell. To use Bash you need either install the Windows Subsystem for Linux (a full Linux installation) or Cygwin (a collection of UNIX programs, including Bash, compiled for Windows).

For this course, it is sufficient to install the free version of MobaXterm (a popular with our Windows users). It includes a basic Cygwin installation along with a graphical file transfer program that lets you click-to-edit and drag-and-drop to transfer files when connected to the supercomputers. An even lighter option is to install the [secure shell installable component](#) (this requires Windows 10 or later, for earlier versions you can use the standalone PuTTY program). Then you can [secure shell](#) from the Windows Command Shell to the supercomputers and use Bash there.

- MobaXterm - <https://mobaxterm.mobatek.net/>
- [Secure Shell](#) - https://docs.microsoft.com/en-us/windows-server/administration/openssh/openssh_install_firstuse
- PuTTY - <https://www.chiark.greenend.org.uk/~sgtatham/putty/latest.html>

Secure shell

Completion requirements

The Canadian supercomputers can be access from anywhere there is internet access using the address `<supercomputer>.alliancecan.ca` and the *secure shell* client suite of [commands](#)

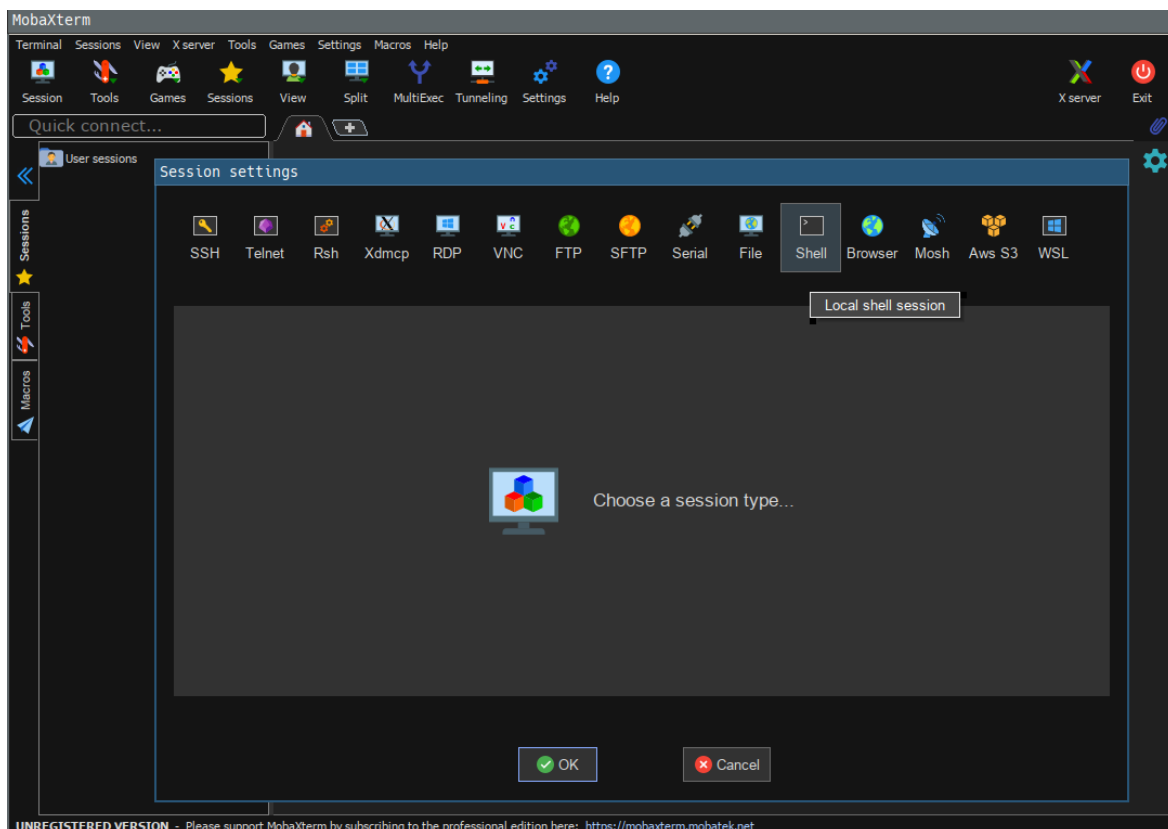
- `ssh`
 - used to run [commands](#) (secure shell)
- `scp`
 - used to copy files (secure copy)
- `sftp`
 - alternative to copy files (secure file transfer protocol)

As a demonstration of our first command, I am now going to show how to connect to the SHARCNET supercomputer graham from a local shell session on your computer using the

`ssh`

command. Connecting to the supercomputer is not required for this course (unless you are using Windows and haven't installed MobaXterm), but it is required to use the supercomputer, so it is good practice.

First we need to open our terminal program. For Linux and Mac OS X, the terminal program can be found by searching for *terminal* under the applications menu. For Windows, if you installed MobaXterm, click the *Sessions icon* in the upper-left, and then click the *Shell icon* the middle-right. If you installed the *secure shell installable component*, start the *Command Shell* under the applications menu. If you have installed PuTTY, start it.



To connect with PuTTY, we type

`graham.alliancecan.ca`

into the *Host Name* box and then press *Open*. It will then prompt for my username and password and then give us a terminal that is connected to graham and running the bash (the default Graham shell). For all other methods we start with a terminal on our local computer running our local shell (generally bash or a bash based derivative). The shell, which is also referred to as the command line or the terminal, does what is known as a read-evaluate-print-loop (REPL). That is, it interacts with us by

- (R)eading a command from me,
- (E)valuating the command,
- (P)rinting the results of the command, and
- (L)ooping (reads the next command, etc.)

A Bash command is generally the name of the program to run followed by any information that program needs to be told in order to do its thing. Assuming we aren't running PuTTY and already connected to graham, we need to run

```
ssh
```

and tell

```
ssh
```

to connect to `graham.alliancecan.ca` with our graham username. So I type

```
[tyson@tux:~]$ ssh tyson@graham.alliancecan.ca
```

(tyson is my graham username) and press enter. This brings us to the second step. The computer runs the

```
ssh
```

command. The

```
ssh
```

command prompts for a password (note there are no stars when typing it in), logs into the graham supercomputer, and starts a new [command line](#) session inside my existing one.

When we are done running [commands](#) on graham, type

```
[tyson@gra-login3 ~]$ exit
```

This causes the [command line](#) session on graham to complete, and, in turn, the

```
ssh
```

command ran on our local computer to also complete. This brings the local computer to the loop phase, and it will then prompt for our next command. We can enter another command for the local computer or type

```
exit
```

, which will cause the the local session to also end and close the terminal application.

Last modified: Friday, 24 May 2024, 11:59 AM

Data storage

Completion requirements

Now we are going to review the basics of data storage on computers. Data is stored in a hierarchical tree structure. For this course we will be working with some data we can download using the

```
wget
```

(web get) command

```
[tyson@gra-login3 ~]$ wget <a href="https://staff.sharcnet.ca/tyson/flights.zip"
```

```
class="_blanktarget">https://staff.sharcnet.ca/tyson/flights.zip</a>
```

You don't need to exit from graham at the end as I did as will continue using it with our next exercise. Note also that the alliance wiki page contains details pages on using ssh, PuTTY, and MobaXterm. or the

```
curl
```

command (in the event your system doesn't have

```
wget
```

```
)
```

```
[tyson@gra-login3 ~]$ curl -LO <a href="https://staff.sharcnet.ca/tyson/flights.zip"
class="_blanktarget">https://staff.sharcnet.ca/tyson/flights.zip</a>
```

We can then unpack this file using the

```
unzip
command
```

```
[tyson@gra-login3 ~]$ unzip flights.zip
```

which leaves me with this tree of files and folders (much of the details outside of the unpacked flights folder are specific to Linux and our supercomputer storage layout and will be different for your personal computer)

```
/
├── home
│   ...
│   ├── tyson
│   │   ├── flights.zip
│   │   ├── flights
│   │   │   ├── 0144f5b1.igc
│   │   │   ├── 2191bc99.igc
│   │   │   ...
│   │   ├── nearline
│   │   │   ├── def-tyson -> /nearline/6001152
│   │   │   └── def-tyson-ab -> /nearline/6023753
│   │   ├── projects
│   │   │   ├── def-tyson -> /project/6001152
│   │   │   └── def-tyson-ab -> /project/6023753
│   │   └── scratch -> /scratch/tyson
│   ...
...
├── nearline
│   ...
│   ├── 6001152
│   ├── 6023753
│   ...
...
├── project
│   ...
│   ├── 6001152
│   ├── 6023753
│   ...
...
├── scratch
```

```
| ...  
| └─ tyson  
| ...
```

- file - named piece of data
- folder - container holding files and further folders
- link - a reference to another file or folder

Before the graphical analogy to a filing system, folders were called directories, and this is reflected in the names of [command line commands](#) (e.g., change directory, print working directory, etc.), so we will use that.

To specify a piece of data, we need to specify both the file name the data is stored under and the series of directories (folders) that that filename is stored under. It isn't sufficient to tell someone (or the computer) just the filename as they would then have to look through all the folders to find it, and they very well might find another file with the same name in a different directory (folder).

To uniquely specify a piece of data we, therefore, have to specify all the parts from the start. For example

- start at the start
- under the *home* directory
- under the *tyson* directory
- under the *flights* directory
- the data is in the *0144f5b1.igc* file

When writing this down, we separate all the components with a / and call it a path because it gives the path to follow through the directories to locate the file. A **leading /** says the path is absolute as it starts at the **very start**. The above example would be */home/tyson/flights/0144f5b1.igc*.

Frequently we are only referring to files relative to some common starting point, such as the location of my person storage */home/tyson*, in which case we call it a relative path and do not include the leading /. The above example would be *flights/0144f5b1.igc*.

When working with data on another computer, we need to specify not only the full file path to the data, but also computer it is on, and the username to use to login to that computer to get it. An example might be ***tyson@graham.alliancecan.ca:/home/tyson/flights/0144f5b1.igc***

You may be more familiar with a Windows file specifications, which would look something more like *C:\Users\Tyson\Desktop\flights\0144f5b1.igc*. The key difference here is that in Windows uses \ instead of / to separate the components and explicitly specifies the physical location of the storage at the start of the path with a drive letter like C:.

There are also no drive letters with Linux. The physical storage is implicit in the path. When I plugin a USB stick, for example, a new path like `/media/tyson/80BC-6336` shows up under which I can access all the files and directories on that USB stick. The

`mount`

command can be used to view what storage is under what paths, but we won't be plugging any USB sticks into the supercomputers, so we will leave it at that.

Last modified: Wednesday, 22 May 2024, 5:59 PM

Exercises

Completion requirements

Doing the following [exercises](#) help ensure you understand the course material. The results of some of the later ones are needed to answer the course completion quizzes.

1. The

`tree`

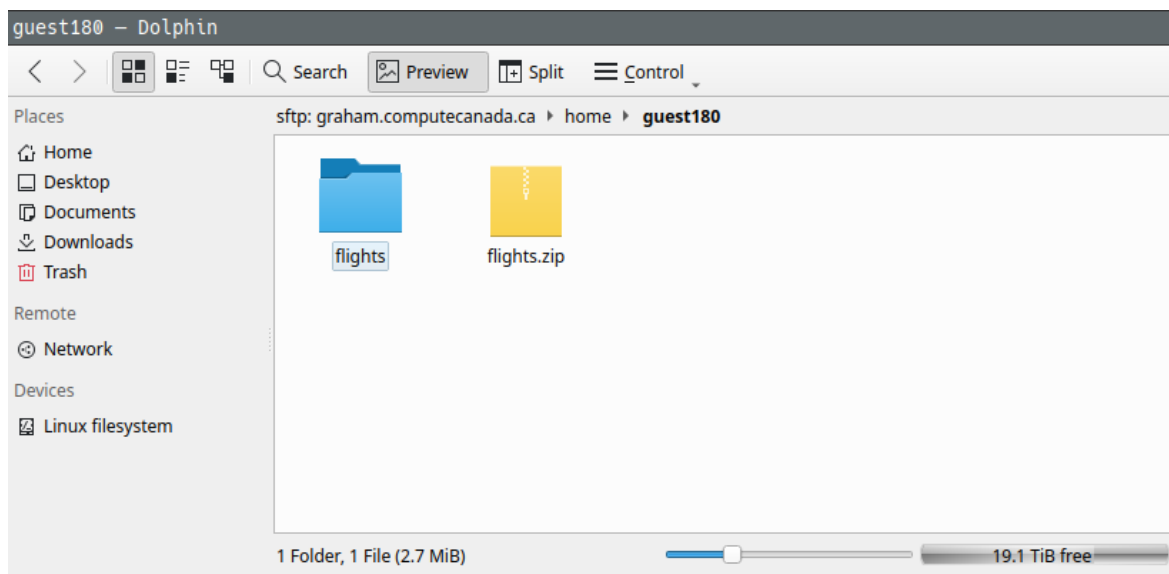
command displays a tree view of your directories and files. Run it and see if the results are what you expect (note this command may not be available on your personal computer).

Last modified: Wednesday, 22 May 2024, 6:00 PM

Getting around

Completion requirements

Under a GUI we navigate our folders (directories) and files with a file manager. A typical graphical file manager shows us the path of the folder we are viewing and its contents as a series of icons



Like a file manager, the [command line](#) has a directory (folder) that it is currently in. We call this the working directory, and the

`pwd`

(print working directory) command will tell us what it is

```
[tyson@gra-login3 ~]$ pwd
```

```
/home/tyson
```

We so frequently want to refer to files relative to our **home directory**, that the [command line](#) provides

`~`

as shortcut to means

`/home/tyson`

. With this information, you can see that the [command line](#) is actually configured to tell us exactly where we are every time we enter a command. That is, `[tyson@gra-login3 ~]$` is saying the command you enter is going to run

- under the user *tyson*
- on the computer *gra-login3*
- in the directory */home/tyson*

If you become a power user with many terminals open at once, you will appreciate this information in your face every time you enter a command.

The file manager also shows us each of the files and folders (directories) in its working directory. The

`ls`

(list) command does the same in the [command line](#)

```
[tyson@gra-login3 ~]$ ls
flights flights.zip nearline projects scratch
```

If we hover our mouse over a file or folder or right click and picking *properties*, we can get extra details about a file or folder, such as the day it was created, its size, and the access permissions. The

```
ls
command will also provide this information to us if we ask it to with the
```

```
-l
(long) switch
```

```
[tyson@gra-login3 ~]$ ls -l
total 2804
drwxr-xr-x 2 tyson tyson 141 Mar 1 2018 flights
-rw-r----- 1 tyson tyson 2794605 Mar 2 2018 flights.zip
drwxr-xr-x 2 root tyson 4 May 24 23:47 nearline
drwxr-xr-x 2 root tyson 4 May 24 23:47 projects
lrwxrwxrwx 1 tyson tyson 14 May 24 23:47 scratch -> /scratch/tyson
```

The file manager lets us open the file by double clicking on it, or right clicking and picking *open with*. For example, double clicking on the *flights.zip* will likely open it in the zip extractor program and let us unpack it. We have already seen how to do this with the [command line](#) when we ran the command

```
unzip flights.zip
```

```
.
```

The file manager also lets us go into the other folders (directories) in the current folder (working directory) by single or double clicking on them. The [command line](#) provides a

```
cd
(change directory) command to do this. For example, the equivalent of going into the
flights folder and looking around would be
```

```
[tyson@gra-login3 ~]$ cd flights
[tyson@gra-login3 flights]$ pwd
/home/tyson/flights
[tyson@gra-login3 flights]$ ls
0144f5b1.igc 2191bc99.igc 4620f232.igc ...
```

You will note that when we moved into the flights directory, the prompt changed from ~ (the abbreviation for */home/tyson*) to *flights* to reflect the fact that we are now in the flights folder. In the file manager, we can click a prior part of the path (or the back arrow) to return to where we were. The [command line](#) provides a special folder called

```
..
```

that refers to the parent folder to allow you to go back

```
[tyson@gra-login3 flights]$ cd ..  
[tyson@gra-login3 ~]$ pwd  
/home/tyson
```

This is actually baked right into the operating system, it just isn't normally shown as files and folders that begin with a period are not shown unless the

```
-a  
(all) flag is used  
[tyson@gra-login3 ~]$ ls -a  
. .. .bash_history .bash_logout ...
```

It is good to know this as many time special things like configurations are stored under files or folders with a leading dot in order to not cluttering up your regular listing. You can also see there is a `.` in addition to the `..` directory. The `.` directory is the directory itself. This is convenient as we frequently want to tell a command to do something to this directory (i.e., copy the files from some place to this directory).

With the file manager we could create a new folder called

```
downloads  
by right clicking and picking create new -> folder and then copy the flights.zip file to it  
by dragging it over and dropping it on the new downloads folder icon. With  
the command line we can make a new folder with the
```

```
mkdir  
(make directory) command and copy the flights.zip file to it with the
```

```
cp  
(copy) command
```

```
[tyson@gra-login3 ~]$ mkdir downloads  
[tyson@gra-login3 ~]$ cp flights.zip downloads/
```

where for copy like [commands](#) you generally specify one or more source followed by a destination separated by spaces. The trailing `/` on the destination is optional, but it makes it unambiguous that *downloads* is suppose to be a directory to a copy of the *flights.zip* file in. Without the trailing `/` the

```
cp  
command determines whether downloads is a folder to put it based on checking to see  
if downloads is an existing directory or not.
```

We have put together a quick reference guide to many of the common (and not so common) [commands](#) and options for you to refer to (see the reference link in the course index) as the goal of this course it not to put you to work memorizing a bunch of [commands](#). The [commands](#) you frequently use will commit to your memory soon enough through regular usage without any effort on your part. You can look up the others when you need to.

Last modified: Friday, 24 May 2024, 12:00 PM

Exercises

Completion requirements

Doing the following [exercises](#) help ensure you understand the course material. The results of some of the later ones are needed to answer the course completion quizzes.

These [exercises](#) assume the following file and directory layout that exists after the previous demonstration (adjusting *tyson* to your username)

```
/
├── home
│   ...
│   ├── tyson
│   │   ├── downloads
│   │   │   └── flights.zip
│   │   ├── flights.zip
│   │   ├── flights
│   │   │   ├── 0144f5b1.igc
│   │   │   ├── 2191bc99.igc
│   │   │   ...
│   │   ...
│   ...
...
```

Discuss your answers and test them out to verify if you are correct or not.

1. The full set of options for a command can be found in the manual page. The command

```
man <command>
```

(e.g.,

```
man ls
```

,

```
man cp
```

, etc.) will bring up the manual page for a command. Use the arrows and page up/down keys to scroll around,

q

to quit, and

/<text>

to search for . Using the

ls

manual page, answer the following questions

a. What does the command

ls -lh

do?

b. What does the command

ls -R

do?

c. How do you sort by last modified date?

2. Starting from */home/tyson/flights directory*, which of the following [commands](#) can be used to switch to the home folder (remember `..` goes the parent directory and `.` stays in the same place)?

a.

cd .

b.

cd /

c.

cd /home/tyson

d.

cd ..

e.

```
cd ~
```

f.

```
cd home
```

g.

```
cd ~/flights
```

h.

```
cd
```

i.

```
cd ../../tyson
```

3. If

```
pwd
```

displays */home/tyson/flights*, what does

```
ls ../downloads
```

display?

a. *ls: cannot access '../downloads': No such file or directory*

b. *downloads flights flights.zip*

c. *0144f5b1.igc 2191bc99.igc ...*

d. *flights.zip*

4. The

```
rmdir
```

(**remove directory**) command removes a directory. Trying to remove the *downloads* directory gives

```
[tyson@gra-login3 ~]$ rmdir downloads
```

```
rmdir: failed to remove 'downloads': Directory not empty
```

This implies we have to empty the directory first with the

```
rm
```

(remove) command. Fortunately

```
rm
```

has an option that will remove everything in one go. Use the manual page to figure out the required command.

5. How can the

```
mv
```

command be used to rename *flights.zip* in */home/tyson* to *flights-downloaded.zip*?

Last modified: Wednesday, 22 May 2024, 6:03 PM

Technicalities

Completion requirements

Now that we have run some [commands](#), and had a look at some of the manual pages, we are going to take a moment to step back discuss some of the technicalities and syntax. The [command line](#) we are using is called *bash*. This is an acronym for Bourne-again shell, which is a word play on the original Bourne shell from which it descended. There are many shells, including the original *sh*, *ksh*, *cs**h*, their decedents *ash*, *bash*, *dash*, *tcsh*, and the even newer *zsh* and *fish*.

The subject of what shell to use can be somewhat of an almost religious issue for some. We are learning *bash* as it is the most widely used and the default on most system. It is bit cruffy due to its extended history, but works well. The biggest gotchas with *bash* is that variable expansion also undergoes word splitting and pathname expansion unless quoted. This means many people's scripts do not properly handle filenames with spaces in them. Neither *zsh*, which is very compatible with *bash*, nor *fish*, which is not, have this issue.

In the [command line](#) quick reference we have stated that programs are run by specifying the command followed by the arguments separated by spaces. That is

```
[tyson@gra-login3 ~] <program> [argument] ...
```


When we write something this way, it is not something you are suppose to type in literally. Rather it is a syntax specification that tells you how to put together the required components when specifying your command. You need to replace the items in the angle and square brackets with what they describe.

That is,

`<program>`

should be replaced by the *the name of the program you wish to run* (e.g.,

`ls`

), and

`[argument]`

should be replaced by the *argument you wish to provide to the program* (e.g.,

`-l`

). **The difference between the**

`<>`

s and

`[]`

s is that the former has to be present while the later is optional. That is, a command must include a program to run, it does not need to include an argument. The full syntax is

- `<xyz>`
 - xyz is required
- `[xyz]`
 - xyz is optional
- `<xyz> ...`
 - xyz is optionally repeated (more of the same)
- `<xyz> | <uvw>`
 - either xyz or uvw but not both

With this in mind, we can now see that saying the syntax for a [command line](#) is

`<program> [arguments] ...`

means a command is a required program name followed by any number of optional arguments (including none) separated by spaces.

Sometimes type faces or capitalization are also used to indicate what parts of a statement are suppose to be typed exactly as is and what parts are suppose to be substituted. Running

```
man cp
```

to bring up the manual page for the

```
cp
```

command gives the following three ways the

```
cp
```

command can be used

```
cp [OPTION]... [-T] SOURCE DEST
```

```
cp [OPTION]... SOURCE... DIRECTORY
```

```
cp [OPTION]... -t DIRECTORY SOURCE
```

We can see that this manual page is using capitalization instead of angle brackets to specify what parts are suppose to be substituted with what they describe. From this we see there are actually three distinct modes in which

```
cp
```

can run, and all three allow any number of the options (e.g.,

```
-a
```

```
,
```

```
-b
```

```
,
```

```
-d
```

```
,
```

```
-f
```

, etc.) to be specified. The first is when you specify only a source and destination file name, as in

```
[tyson@gra-login3 ~] mkdir example
```

```
[tyson@gra-login3 ~] cp -T flights-downloaded.zip example/flights.zip
```

This make a copy of the *SOURCE* file called *DEST*. We have provided the optional

```
-T
```

parameter in this example. This doesn't do anything unless *DEST* happens to exist as a directory. In this case

```
cp
```

will provide an error instead of assuming you are invoking the second variant of the command.

```
[tyson@gra-login3 ~] cp -T flights-downloaded.zip example
```

```
cp: cannot overwrite directory 'example' with non-directory
```

Without this option, we would inadvertently invoke the second form of the

```
cp
```

command which copies one or more files into a destination directory, as in

```
[tyson@gra-login3 ~] cp flights/0144f5b1.igc flights/2191bc99.igc example
```

The final is the same as the second except you specify the destination directory first

```
[tyson@gra-login3 ~] cp -t example flights/0144f5b1.igc flights/2191bc99.igc
```

again this is provided only to ensure you don't accidentally invoke the first form when you really wanted the second form.

```
[tyson@gra-login3 ~] cp -t examplee flights/0144f5b1.igc flights/2191bc99.igc
```

```
cp: failed to access 'examplee': No such file or directory
```

One final nice feature of bash is that it has a history of prior run [commands](#) and does completion of [commands](#) and filenames. Pressing the up and down arrow keys will scroll through your previously run [commands](#) so you can edit and rerun them without having to type them all back in. Pressing the *tab* key partway through a filename or command will complete it up to the first ambiguity. Pressing *tab* key again will display all possibilities. For example

```
[tyson@gra-login3 ~] r<PRESS TAB TWICE>
```

```
Display all 158 possibilities? (y or n) y
```

```
ranlib reduce_test ...
```

```
[tyson@gra-login3 ~] rm -fr exa<PRESS TAB ONCE>
```

```
[tyson@gra-login3 ~] rm -fr example/
```

where we are abusing our angle bracket and capital notation to tell you to press the *tab* key. I would strongly recommend forcing yourself to use *tab competition* throughout this workshop as, once it becomes second nature, it will vastly improve the speed with which you can run [commands](#).

Last modified: Wednesday, 22 May 2024, 6:03 PM

Transferring files

Completion requirements

The other use of the [secure shell](#) client programs is for transfer files. The

```
scp
```

(secure copy) command is basically a version of the

```
cp
```

command where you can specify a remote computer as the source or destination. As an example of this command, I will end my session on graham, which returns me to the [command line](#) on my local Linux computer, and use the

```
scp
```

command to copy the `/home/tyson/flights/0144f5b1.igc` file from graham

```
[tyson@gra-login3 ~] exit
```

```
[tyson@tux:~]$ scp tyson@graham.alliancecan.ca:~/flights/0144f5b1.igc .
```

Note that I've used `.` as the location to copy the file to, which means the working directory.

There are also many graphical applications, such as MobaXterm and WinSCP under Windows, that use the *secure file transfer protocol (sftp)* in the background to let you simply drag and drop files between your computer. Most Linux file managers also have this ability built into them and you simply need to specify the remote path to accessing using the special

```
sftp://<user>@<computer>/<path>
```

URL

You may need to setup your *secure shell* client with a *secure shell* key so it can login to graham without using a password for this work. See the alliance documentation wiki for details on how to do this.

Linux and Mac OS X (if you install *FUSE for macOS*) also have the ability to splice a remote *file system* into your local *file system* using the

```
sshfs
```

command. For example,

```
[tyson@tux:~]$ mkdir graham
```

```
[tyson@tux:~]$ ls graham
```

```
[tyson@tux:~]$ sshfs tyson@graham.alliancecan.ca:/home/tyson graham
```

```
[tyson@tux:~]$ ls graham
```

```
flights flights-downloaded.zip
```

This is quite powerful as you can then do anything you can do with local files on the remote files. Examples include simply using the standard

```
cp
```

(copy) command to copy them to a local path

```
[tyson@tux:~]$ cp graham/flights-downloaded.zip .
```

or even directly edit them in your standard editor. Of course they will take longer to access though as they are actually being transferred back and forth under the hood using the *secure shell* system. The

```
fusermount
```

command with the

-u

option un-splices the remote [file system](#)

```
[tyson@tux:~]$ fusermount -u graham
```

```
[tyson@tux:~]$ ls graham
```

Last modified: Wednesday, 22 May 2024, 6:04 PM

Exercises

Completion requirements

Doing the following [exercises](#) help ensure you understand the course material. The results of some of the later ones are needed to answer the course completion quizzes.

1. Transfer some of the *.igc* files in the *flights* directory to your computer. Have a look at them in your text editor and view them in the [online IGC file viewer](#). We will be using command line tools to automate processing of these files shortly.
2. The

rsync

command is very useful when working on multiple computers. An example of typical usage might be

```
[tyson@tux:~]$ rsync -e ssh -auvP tyson@graham.alliancecan.ca:/home/tyson/flights .
```

Use the

rsync

manual page to describe what this command does.

3. What advantage does

rsync

have over using

-r

with

scp

to recursively copy all files and directories?

4. Would it be easier to automate dragging and dropping new files or running an `rsync` command for weekly updates?

Last modified: Wednesday, 22 May 2024, 6:05 PM

Composition

Introduction

Completion requirements

As has already been hinted at, the power of the [command line](#) system is not so much how amazing (or not) the individual [commands](#) are, but how well those [commands](#) can be easily combined together to give new desired behavior. The reason for this is, in any graph, the number of edges can grow much faster than the number of nodes.

That is, consider a system with just 10 programs. If these programs can only be used independently of each other, as is often the case with GUI programs, then we have 10 programs that we can run. If each of these programs can be combined with one of the others though, then we also have $10 \times 9 = 90$ compositions we can run. Further, when we get another program, if it only stands on its own, then we have increase our options by 1 (from 10 to 11). If it can be combined with the others though, then our combinations increases by 20 (from $10 \times 9 = 90$ to $11 \times 10 = 110$).

In order for programs to compose, they need to talk a common language. In Linux, with its Unix heritage, this is text. As Doug McIlrory, who invented Unix pipes said,

This is the Unix philosophy: Write programs that do one thing and do it well. Write programs to work together. Write programs to handle text streams, because that is a universal interface

The other universality you will find in Linux is a huge amount of information about the operating system itself is exposed as text files. This allows all programs to consume and manipulate this information. To quote the Unix Architecture page on Wikipedia

With few exceptions, [devices](#) and some types of communications between processes are managed and visible as files or pseudo-files within the [file system](#) hierarchy. This is known as *everything's a file*.

As a simple example of this lets look at the `/proc/cpuinfo` and `/proc/meminfo` files using the

```
cat
```

command which prints all the files it is given to the screen one after another (concatenates them together)

```
[tyson@gra-login3 ~] cat /proc/cpuinfo
processor : 0
vendor_id : GenuineIntel
cpu family : 6
model : 79
model name : Intel(R) Xeon(R) CPU E5-2667 v4 @ 3.20GHz
```

...

```
[tyson@gra-login3 ~] cat /proc/meminfo
MemTotal: 131624960 kB
MemFree: 27484820 kB
MemAvailable: 53464260 kB
SwapCached: 42912 kB
```

...

Having all this information available to us as simple text files gives us a lot of power in our shell. Things that would be only available by writing a program to make a special operating system (OS) call in another OS, can be retrieved and manipulated with our basic file [commands](#) like

```
cat
```

in our shell.

Last modified: Wednesday, 22 May 2024, 6:07 PM

Viewing and editing

Completion requirements

One of the challenges in preparing a course is to come up with interesting examples. Several years ago, my wife's cousin took her flying in a glider. She really enjoyed the experience, and a couple of years ago we joined the London Soaring Club to learn how to fly gliders too. This has proven to be a lot of fun. To give you a better visual, here is a random picture from the internet of a trainee in a two-seater trainer (the instructor sit in the seat behind you).

The gliders all have GPS trackers in them, and the club has a variety of awards that are awarded each year based on the traces (e.g., who gained the most altitude, who flew the furthest, etc.). Going through these files by hand is a tedious chore, but it is easy to automate with the [command line](#), and is typical of the sort of data and log file real-world pre- and post-processing we frequently have to do as researchers.

The *flights.zip* file we have downloaded and unpacked contains a variety of glider flight traces and we are going to use them for our [exercises](#) now. We have already introduced the

cat

command to display the entire contents of a file (or multiple files). Often we are only interested in the start or end of a file though. The

head

and

tail

[commands](#) allow us to extract just the start or end of a file. Both show ten lines by default, but can be told to show an arbitrary number with the

-n <number>

option. Using this to look at one of the igc files

```
[tyson@gra-login3 ~]$ head flights/0144f5b1.igc
```

```
AXCSAAA
```

```
HFDTE030816
```

```
HFFXA050
```

```
HFPLTPILOTINCHARGE:Lena
```

```
HFGTYGLIDERTYPE:L23 Super Blanik
```

```
...
```

```
[tyson@gra-login3 ~]$ tail flights/0144f5b1.igc
```

```
B2116474309137N08057022WA002870042000309
```

```
B2116524309138N08057024WA002870041800309
```

```
B2116574309138N08057024WA002870041800309
```

```
B2117024309138N08057024WA002870041800309
```

```
B2117074309138N08057024WA002870041800309
```

```
...
```

We see the files are composed of a series of records. Googling will give the full igc file specification, but, for our purposes, the records of interest are the date record, the pilot record, the plane record, and GPS position record

HFDTE<DD><MM><YY> (date: day, month, year)

HFPLTPILOTINCHARGE:<NAME> (pilot: name)

HFGIDGLIDERID:<CALLSIGN> (plane: call sign)

B<HH><MM><SS> (time: hour, minute, second,

<DD><MM><mmm>N (latitude: degrees, minute, decimal minutes)

<DDD><MM><mmm>W (longitude: degrees, minutes, decimal minutes)

A<PPPPP><GGGGG> (altitude: pressure, gps)

<AAA><SS> (gps: accuracy, satellites)

Linux also comes with a variety of [text editors](#). The two most common ones are

emacs

and

vi

, both of which were created in 1976, have an almost cult-like following, and will seem quite strange to the uninitiated. The quick reference guide contains the basic key strokes/[commands](#) to use these [editors](#). For a new user, the most important thing to know is how to **exit** these [editors](#) if you accidentally get into them

- emacs

- press

CTRL+x CTRL+c

- vi

- type

:q!

where

CTRL+<key>

means to press while holding down the *control key*. Other ways you will see

CTRL+c

written is

^c

and

C-c

.

For a first time user,

nano

is likely a good choice for making simple edits. It is a basic no-frills editor that lets you move around with the cursor keys and make edits as most people expect. The key sequences to exit, save (write out) the file, and so on are printed at the bottom of the screen using the

^<key>

format (e.g., press while holding down the *control key* to exit), so you won't forget them. Sometimes, when you run a command, it will start an editor for you to edit something. Chances are this editor will be

vi

and you will need to know the

:q!

sequence to get out. You can set a person default editor by setting

EDITOR

environment variable to the editor you want

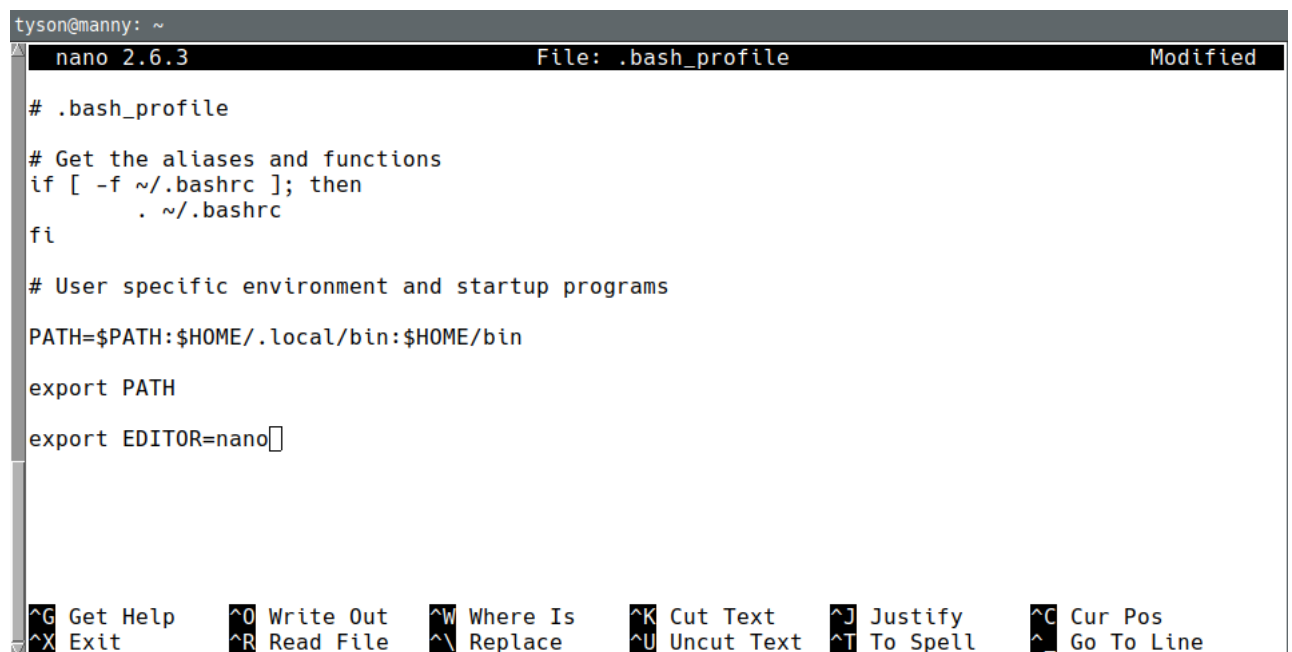
```
export EDITOR=nano
```

There are actually many such settings that programs inherit from your [command line](#) session, and even more that are just specific to *bash* and not inherited by other programs. They are documented in the various [commands](#) manual pages.

The [commands](#) in *~/.bash_profile* (~ mean */home/tyson*) are always run at each login, so it is a good place put such setting that you always want set.

We will do this now to demonstrate editing a file.

```
[tyson@gra-login3 ~]$ nano .bash_profile
```



```
tyson@manny: ~
nano 2.6.3 File: .bash_profile Modified

# .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

export PATH

export EDITOR=nano

^G Get Help      ^O Write Out    ^W Where Is     ^K Cut Text     ^J Justify      ^C Cur Pos
^X Exit          ^R Read File    ^\ Replace      ^U Uncut Text   ^T To Spell     ^_ Go To Line
```

After editing *~/.bash_profile* (runs every login) or *~/.bashrc* (runs every time

bash

runs), always test you can still login with a new

ssh

session in a new terminal. These are startup files, and some errors can leave you unable to login, which will be impossible to fix unless you still have an active sessions running to fix things.

Last modified: Friday, 24 May 2024, 12:03 PM

Exercises

Completion requirements

Doing the following [exercises](#) help ensure you understand the course material. The results of some of the later ones are needed to answer the course completion quizzes.

An example of something we may want to do with these files is determine which pilot each file belongs to in order to give each member a copy of their file. In this exercise we are just going to use our new [commands](#) to look at the files and do a few operations by hand to get an idea of the sort of things we will be automating.

1. Using

```
head
```

to look at a few (say 3-5) of the igc files (refer to the igc file specification given earlier) and answer the following questions

- a. who the pilot is, and
- b. what the year was.

2. For each of these files first few files, use the

```
mkdir
```

and

```
cp
```

[commands](#) to

- a. make a directory for that pilot (if required),
- b. make a directory in the pilot's directory for that year (if required), and
- c. copy the file into the pilot/year directory.

3. What happens if you run

```
cat
```

,

```
head
```

, and

tail

without any filename argument? In trying this, be aware that

CTRL+c

can be used to abort most command and

CTRL+d

signals the end of keyboard input.

Last modified: Wednesday, 22 May 2024, 6:14 PM

Regular expressions and globbing

Completion requirements

[Regular expressions and globbing](#)

The

grep

(global regular expression search) command searches through files for [regular expressions](#). Regular expressions are a sequence of characters that define a search pattern. Variants of them are supported by a wide range of applications, including google sheets, and are well worth learning. A very simple usage would be to use

grep

to extract all the line that starts with

HFPLT

from an igc file

```
[tyson@gra-login3 ~]$ grep ^HFPLT flights/0144f5b1.igc
```

```
HFPLTPILOTINCHARGE:Lena
```

As can be seen in the above example, a regular expression is simply a sequence of regular characters that match themselves plus some special characters like

^

that match things like the start of the line. The most basic matches supported by almost all [regular expressions](#) are (these are covered in our quick reference guide and the

grep

manual page too)

- `^`
 - match start of line
- `$`
 - match end of line
- *character* - match the indicated character
- `.`
 - match any character
- `[`
 - `...`
 - `]`
 - match any character in the list or range (`^` inverts)
- `(`
 - `...`
 - `)`
 - group
- `...`
 - `|`
 - `...` - match either or
- `?`
 - match previous item zero or one times
- `*`

- match previous item zero or more times

- +

- match previous item one or more times

- {

...

}

- match previous item a range of times

Regular expression implementations frequently differ in what special characters have to be escaped (preceded with a

\

) to have the above special meaning or not. For example,

grep

supports both basic and extended regular expressions where the former requires several of the special characters to be escaped to have their special meaning and the latter does not.

A more complex example would

```
[tyson@gra-login3 ~]$ grep '^B.*A.....5..' flights/0144f5b1.igc
```

```
B1837024309903N08053926WA009230050102706
```

```
B1837074309873N08053817WA009200051002706
```

...

which gives all GPS trace lines with a GPS altitude recording of 500-599 meters.

In this example we have enclosed the regular expression in single quotes. This is because *bash* also has sequences that it treats special, and this includes

*

which, without the single quotes, indicates a glob pattern for pathname expansion. Glob patterns allow us to specify pathnames with the following wildcards

- *

- match any number of characters (include none)

- ?

- match any single character

• [

...

]

- match any character in this list

When

bash

sees unquoted versions of these in a [command line](#), it replaces the pattern with all pathnames that match the pattern. This allows us to easily run [commands](#) like extract the pilot line from all the igc files

```
[tyson@gra-login3 ~]$ grep ^HFPLT flights/*.igc
0144f5b1.igc:HFPLTPILOTINCHARGE:Lena
04616075.igc:HFPLTPILOTINCHARGE:Bill
```

...

It is important to realize is that it is

bash

that literally replaces the pattern with all the matching pathnames. The command that is being run never sees the patterns. It just gets the list of files, exactly as if we had typed in all the filenames ourselves.

There are actually many such expansions that can occur, and sometimes it is useful to know what the command really being run is. The

set -x

command tells

bash

to print out each command it runs, and

set +x

tells bash to stop printing them out. We can use this to see exactly how our pattern gets expanded into the command that is run

```
[tyson@gra-login3 ~]$ set -x
[tyson@gra-login3 ~]$ grep ^HFPLT flights/*.igc
+ grep --color=auto '^HFPLT' flights/0144f5b1.igc flights/04616075.igc ...
0144f5b1.igc:HFPLTPILOTINCHARGE:Lena
```

```
04616075.igc:HFPLTPILOTINCHARGE:Bill
```

```
...
```

```
[tyson@gra-login3 ~]$ set +x
```

This also shows something I hadn't intended to talk about. The

```
grep
```

command is automatically being provided an

```
--color=auto
```

option. This is done with the

```
bash
```

aliases feature which can provide default arguments and short forms for various [commands](#). It is also commonly used to make

```
-i
```

(verify) a default for many [commands](#) like

```
rm
```

. You can view the current set of alias with the

```
alias
```

command and read more about them in the

```
bash
```

manual page.

Last modified: Wednesday, 22 May 2024, 6:15 PM

Redirection and pipes

Completion requirements

Now that we have a simple way of extracting all the pilot and date fields from all the igc files, we need to use a redirection to save it in a file so we can process it further.

Redirections are specified at the end of the command line with the

```
<
```

and

```
>
```


characters. A simple mnemonic is that they are like arrows re-directing the input and output, receptively from and to a file. An example redirection to save out list of pilots to *pilots* would be

```
[tyson@gra-login3 ~]$ grep ^HFPLT flights/*.igc > pilots-extracted
```

Running this command produces no output as all the output has been redirected from the screen to the *pilots* file, as we can easily verify by loading *pilots* in our editor or printing it out using

```
cat
```

```
[tyson@gra-login3 ~]$ cat pilots-extracted
flights/0144f5b1.igc:HFPLTPILOTINCHARGE:Lena
flights/04616075.igc:HFPLTPILOTINCHARGE:Bill
flights/054b9ff8.igc:HFPLTPILOTINCHARGE:Lena
...
```

We only want the pilot names though, not the filename and the

```
HFPLTPILOTINCHARGE:
```

field identifier. To remove these we can use the

```
cut
```

command which lets us cut out bits from a line. The pilot names always start at the 41st character in this output, so one way to do this would be to specify

```
-c 41-
```

, which means give me from character 41 to the end of the line. A simpler way though is to note the line is broken in parts with the

```
:
```

character. With this observation, we can use

```
-d :
```

to break it up into three pieces as

```
-f 3
```

to select the third

```
[tyson@gra-login3 ~]$ cut -d : -f 3 pilots-extracted
```

```
Lena
```

```
Bill
```

```
Lena
```

```
...
```

```
[tyson@gra-login3 ~]$ cut -d : -f 3 pilots-extracted > pilots-names
```

The only issue we have now is we have each pilot listed for each flight they took. We would like to have each pilot listed only once. As it happens, there is

`uniq`

command that removes duplicate lines. On closer reading of the manual page though, it becomes apparent that `uniq` only removes duplicate lines if they are adjacent, and states that you have to run your file through the

`sort`

command first. Doing this we get

```
[tyson@gra-login3 ~]$ sort pilots-names > pilots-sorted
```

```
[tyson@gra-login3 ~]$ uniq pilots-sorted
```

Aasia

Bill

Fred

...

Creating a bunch of intermediate files simply to feed the output from one command into the input of the next command is tedious though, so

`bash`

provides a

|

(pipe) syntax to do this for us. With this syntax we can eliminate all the temporary files in

```
[tyson@gra-login3 ~]$ grep ^HFPLT flights/*.igc > pilots-extracted
```

```
[tyson@gra-login3 ~]$ cut -d : -f 3 pilots-extracted > pilots-names
```

```
[tyson@gra-login3 ~]$ sort pilots-names > pilots-sorted
```

```
[tyson@gra-login3 ~]$ uniq pilots-sorted
```

and collapse it down to just

```
[tyson@gra-login3 ~]$ grep ^HFPLT flights/*.igc | cut -d : -f 3 | sort | uniq
```

Aasia

Bill

Fred

...

As a small point of clarification, this pipe command is not creating temporary files and providing them to the [commands](#). Rather it redirecting the (screen) output of each program into the (keyboard) input of the next one. This works as all the above [commands](#) (and most others) get their input from the keyboard if a filename is not specified (this came up in an earlier exercise). The above is then technically equivalent to

```
[tyson@gra-login3 ~]$ grep ^HFPLT flights/*.igc > pilots-extracted
[tyson@gra-login3 ~]$ cut -d : -f 3 < pilots-extracted > pilots-names
[tyson@gra-login3 ~]$ sort < pilots-names > pilots-sorted
[tyson@gra-login3 ~]$ uniq < pilots-sorted
Last modified: Wednesday, 22 May 2024, 6:17 PM
```

Exercises

Completion requirements

Doing the following [exercises](#) help ensure you understand the course material. The results of some of the later ones are needed to answer the course completion quizzes.

In these [exercises](#) we are going to create a series of pipelines for extracting key bits of information from our igc files. In the next section we will be converting these pipeline [commands](#) into shell scripts.

The key to building a successful pipeline is to build it up slowly and test each addition. Instead of typing in an entire pipeline, running it, and then not knowing where it went wrong, test each stage and get it correct before adding the next. This is very easy to do as the up arrow brings up the previous [commands](#) run for further editing and re-running. For example

```
[tyson@gra-login3 ~]$ grep ^HFPLT flights/*.igc | head
[tyson@gra-login3 ~]$ grep ^HFPLT flights/*.igc | cut -d : -f 3 | head
[tyson@gra-login3 ~]$ grep ^HFPLT flights/*.igc | cut -d : -f 3 | sort | head
[tyson@gra-login3 ~]$ grep ^HFPLT flights/*.igc | cut -d : -f 3 | sort | uniq
First an output redirection question though.
```

1. What is the difference between redirecting output with

```
>
```

and

```
>>
```

(hint, try running the same redirected command twice and see what happens to the output file in each case).

2. Create pipelines to extract
 - a. the flight date from a single igc file,

- b. all flight years from all igc files,
 - c. all GPS time records from a single igc file, and
 - d. all GPS altitude records from a single igc file.
- 3. Extend the GPS time record extraction pipeline
 - a. to give the starting (first) time, and
 - b. to give the finishing (last) time.
- 4. Extend the GPS altitude record extraction pipeline to give the highest altitude.

Last modified: Wednesday, 22 May 2024, 6:18 PM

Automation

Introduction

Completion requirements

The [command line](#) keeps track of prior [commands](#) that have been run. These include both interactively selecting prior [commands](#) to re-run

- *up* and *down* arrow keys to bring back up prior [commands](#)
- *CTRL+r* to interactively search through prior [commands](#)

and the

history

command to display the last [commands](#). This lets us process our history of [commands](#) with our [commands](#).

As an example, we created a variety of pipelines for extracting information from the igc files, such as

```
[tyson@gra-login2 ~]$ grep '^B' flights/0144f5b1.igc | cut -c 31-35 | sort | tail -n 1
```

which gives the highest GPS altitude recorded in the given igc file. After such as session we can use the

history

command along with output redirection to save the [commands](#) we came up with to a file for future reference of sharing with a colleague

```
[tyson@gra-login2 ~]$ history 20 > flight-<a class="autolink" title="Commands" href="https://training.computeontario.ca/courses/mod/page/view.php?id=1611">commands</a>
where the
```

20

specifies that we want the last twenty [commands](#) run. From this it is trivial to open our file up in a text editor like

nano

and clean it up a bit to get a nice reference

```
[tyson@gra-login2 ~]$ nano flight-<a class="autolink" title="Commands" href="https://training.computeontario.ca/courses/mod/page/view.php?id=1611">commands</a>
```

...

```
[tyson@gra-login2 ~]$ cat flight-<a class="autolink" title="Commands" href="https://training.computeontario.ca/courses/mod/page/view.php?id=1611">commands</a>
```

Reference **of** useful pipes **for** working with igc files extracted from history

Date

```
grep '^HFDTE' flights/0144f5b1.igc | cut -c 6-
```

Pilot

```
grep '^HFPLT' flights/0144f5b1.igc | cut -d : -f 2
```

Plane

```
grep '^HFGIDGLIDERID' flights/0144f5b1.igc | cut -d : -f 2
```

Start time

```
grep '^B' flights/0144f5b1.igc | cut -c 2-7 | head -n 1
```

End time

```
grep '^B' flights/0144f5b1.igc | cut -c 2-7 | tail -n 1
```

Highest GPS altitude

```
grep '^B' flights/0144f5b1.igc | cut -c 31-35 | sort | tail -n 1
```

All pilots

```
grep ^HFPLT flights/*.igc | cut -d : -f 3 | sort | uniq
```

Once we have our [commands](#) in a file, it is pretty natural to wonder if we can get bash to just run our [commands](#) from the file instead of us having to type them back in each time.

Last modified: Wednesday, 22 May 2024, 6:19 PM

Scripting

Completion requirements

This is precisely what a shell script is: a file with a list of [commands](#) in it that we get our shell (bash) to run. Our *flights-command* is almost a shell script as we have written it above. The only issue is that bash doesn't know what to make of the comments as they aren't proper [commands](#). We can fix this by prefixing them with

```
#
```

to mark them as comments

```
[tyson@gra-login2 ~]$ nano flight-<a class="autolink" title="Commands"
href="https://training.computeontario.ca/courses/mod/page/view.php?id=1611">commands</a>
...
[tyson@gra-login2 ~]$ cat flight-<a class="autolink" title="Commands"
href="https://training.computeontario.ca/courses/mod/page/view.php?id=1611">commands</a>
# Reference of useful pipes for working with igc files extracted from history
# Date
grep '^HFDTE' flights/0144f5b1.igc | cut -c 6-
```

Now we can tell bash to run our [commands](#) directly for us from our file

```
[tyson@gra-login2 ~]$ source flight-<a class="autolink" title="Commands"
href="https://training.computeontario.ca/courses/mod/page/view.php?id=1611">commands</a>
There are actually several ways this last step can be done
```

- `. <filename>`

or

```
source <filename>
```

- run [commands](#) in current session

- `(source <filename>)`

- run [commands](#) in a sub shell (current directory and such will be restored)

- `bash <filename>`

- start a new shell, run the [commands](#), and exit back to current shell

Earlier we had mentioned that Linux doesn't use a

```
.exe
```

extension to identify executable files. Instead executable files have the `executable mode` set on them. We can set this with the command

```
chmod +x <file>
```

and we can see it as the

```
x
```

when we run look at the

```
ls -l
```

long listing. Because our program is a script, we also have to tell Linux what program to use to run it by adding a special

```
#!/<interpreter>
```

comment to start of it

```
[tyson@gra-login2 ~]$ ls -l flights-<a class="autolink" title="Commands"
href="https://training.computeontario.ca/courses/mod/page/view.php?id=1611">commands</a>
-rw-r----- 1 tyson tyson 630 May 11 22:32 flights-<a class="autolink" title="Commands"
href="https://training.computeontario.ca/courses/mod/page/view.php?id=1611">commands</a>
[tyson@gra-login2 ~]$ chmod +x flight-<a class="autolink" title="Commands"
href="https://training.computeontario.ca/courses/mod/page/view.php?id=1611">commands</a>
-rwxr-x--- 1 tyson tyson 630 May 11 22:32 flights-<a class="autolink" title="Commands"
href="https://training.computeontario.ca/courses/mod/page/view.php?id=1611">commands</a>
[tyson@gra-login2 ~]$ nano flight-<a class="autolink" title="Commands"
href="https://training.computeontario.ca/courses/mod/page/view.php?id=1611">commands</a>
...
[tyson@gra-login2 ~]$ cat flight-<a class="autolink" title="Commands"
href="https://training.computeontario.ca/courses/mod/page/view.php?id=1611">commands</a>
```

```
#!/bin/bash
```

```
# Reference of useful pipes for working with igc files extracted from history
```

```
...
```

With all this in place (the executable mode set and the special interpret comment as the first line) we can now directly run our file as if it was just another command

```
[tyson@gra-login2 ~]$ flight-<a class="autolink" title="Commands"
href="https://training.computeontario.ca/courses/mod/page/view.php?id=1611">commands</a>
-bash: flight-<a class="autolink" title="Commands"
href="https://training.computeontario.ca/courses/mod/page/view.php?id=1611">commands</a>: command not
found
[tyson@gra-login2 ~]$ ./flight-<a class="autolink" title="Commands"
href="https://training.computeontario.ca/courses/mod/page/view.php?id=1611">commands</a>
```

```
030816
```

```
Lena
```

```
...
```

The first run attempt failed because the current directory is not somewhere bash look for a command unless we explicitly tell it to as we did in the second command. environment variable in a

```
:
```

delimited format. We can see the setting of this variable by either using variable substitution with the

```
echo
```

command or using the

```
declare
```

command to print it

```
[tyson@gra-login2 ~]$ echo $PATH
```

```
/opt/software/slurm/current/bin:../home/tyson/bin
```

```
[tyson@gra-login2 ~]$ declare -p PATH
```

```
declare -x PATH="/opt/software/slurm/current/bin:../home/tyson/bin"
```

The

```
declare
```

version is interesting as it actually prints the

```
declare
```

command we would have to run to set it to its current value. This shows us additional information such as the

```
-x
```

which means that it is to be also made available (exported) to [commands](#) that bash runs as well.

You might be tempted to add

```
.
```

(the current directory) to this list. This will work, but don't do it. If someone puts a

```
ls
```

command in a directory you go into and run

```
ls
```

in, it will then run their

```
ls
```

command and not the system one you are expecting. Their

ls

command could do anything, including deleting all your files or giving them access to your account in the background. The last element of

PATH

is a

bin

directory under your home directory. Create this directory instead and put your scripts there

```
[tyson@gra-login2 ~]$ mkdir bin
```

```
[tyson@gra-login2 ~]$ mv flight-<a class="autolink" title="Commands"
```

```
href="https://training.computeontario.ca/courses/mod/page/view.php?id=1611">commands</a> bin
```

```
[tyson@gra-login2 ~]$ flight-<a class="autolink" title="Commands"
```

```
href="https://training.computeontario.ca/courses/mod/page/view.php?id=1611">commands</a>
```

030816

Lena

GBJY

...

Our command isn't as useful as the other [commands](#) though as we can tell them what files to operate on. Our command just ignores everything we tell it and always does the same operations on the same files

```
[tyson@gra-login2 ~]$ flight-<a class="autolink" title="Commands"
```

```
href="https://training.computeontario.ca/courses/mod/page/view.php?id=1611">commands</a> --you-are-just-going-to-ignore-this--
```

030816

Lena

GBJY

...

To make our command more useful, we can use variables to change it from a specific command to run to a template command to run. We do this by replacing the fixed filenames with special symbols (variables) that get replaced with the arguments provided on the [command line](#)

- `$<n>`

- the nth argument provided on the [command line](#)

- `$@`

- all the arguments provided on the [command line](#) separated by spaces

• \$#

- the number of arguments provided on the [command line](#)

With this we can make a copy of our example [commands](#) file and edit it into a command that takes an igc filename and prints the date of the flight

```
[tyson@gra-login2 ~]$ cd bin/flights-<a class="autolink" title="Commands" href="https://training.computeontario.ca/courses/mod/page/view.php?id=1611">commands</a> bin/igc-date
[tyson@gra-login2 ~]$ nano bin/igc-date
```

...

```
[tyson@gra-login2 ~]$ cat bin/igc-date
```

```
#!/bin/bash
```

```
# Run the date extraction pipeline using the first argument as the source filename
```

```
grep '^HFDTE' $1 | cut -c 6-
```

```
[tyson@gra-login2 ~]$ igc-date ../flights/fffdcaad.igc
```

```
250616
```

All we have done is put a name to pipeline template. This isn't trivial though. Our minds can only deal with so much information at any one point. Switching from thinking about a complex pipeline to a simple, appropriate-named command, frees up the brain power required to successfully integrate that command into its some other complex operation. Repeating this process lets us build up from small blocks to mansions.

Last modified: Wednesday, 22 May 2024, 6:20 PM

Exercises

Completion requirements

Doing the following [exercises](#) help ensure you understand the course material. The results of some of the later ones are needed to answer the course completion quizzes.

In these [exercises](#), you will see the

```
;
```

character. In bash the

```
;
```

is equivalent to a *newline* (pressing *enter* on your keyboard). This lets us write multiline [commands](#) on a single line. You will see when you scroll back through your history (the *up* key), that bash will replace your *newlines* with

```
;
```

S.

1. In the live session, we converted our example date extension pipeline into an new

igc-date

command

- igc-date <filename>

- date field from the igc-file

Do this for the other pipelines to create the following [commands](#)

a.

igc-pilot <filename>

- pilot field from the igc-file

b.

igc-plane <filename>

- plane call sign from the igc-file

c.

igc-start <filename>

- starting (first) time for the igc-file

d.

igc-**end** <filename>

- ending (last) time for the igc-file

e.

igc-maxalt <filename>

- maximum altitude recorded in the igc-file

2. The

```
[[ <test> ]]
```

command lets us perform a variety of test (see

```
help [[
```

and

```
help test
```

). Combined with the

```
if <command>; then <command>; else <command>; fi
```

command (see

```
help if
```

), this lets us write a further improved

```
igc-date
```

command that provide feedback to the user if it was invoked incorrectly.

```
[tyson@gra-login2 ~]$ cat bin/igc-date
```

```
#!/bin/bash
```

```
if [[ $# -eq 1 ]]; then
```

```
grep '^HFDTE' $1 | cut -c 6-
```

```
else
```

```
echo "Proper usage is igc-date [igc file]"
```

```
fi
```

Give this a try and update the other [commands](#) to also do this.

3. Add an

```
elif [[ $# -eq 0 ]]; then <command>
```

branch to make the command also support reading the igc file directly from the keyboard when not given any filenames as most other [commands](#) do (remember

```
CTRL+c
```

aborts and

```
CTRL+d
```

signals the end of the input when testing this out).

Last modified: Wednesday, 22 May 2024, 6:20 PM

Submitting jobs

Completion requirements

Earlier we discussed how the graham supercomputer is actually a large number very beefy standard computers, and that you ran programs on these computers by telling the system

- what [commands](#) you want to run, and
- what resources those [commands](#) require to run.

Now that we know how to create scripts, we know how to do this first of these. The second is a simply a matter of looking through the command options for

SBATCH

(slurm batch) command. The alliance [documentation wiki](#) has pretty extensive coverage of most circumstances and what options should be specified. At a minimum we need

- `--time [[dd-]hh:]<mm>`
- amount of time required
- `--mem-per-cpu [megabytes]`
- amount of memory required
- `--account <account>`
- sponsor account to record usage against
- `--output <file>`
- file to record output in

The script will be killed if it exceeds the resources specified, so we want to give ourselves a bit of room when specifying our limits. We don't want to be excessive though, as our script will not run until the system has secured all the specified resources for us, so the more we specify, the longer we wait before running.

For most people there is only one account to submit the script under (their sponsor's default account).

Running without

```
--account
```

option will print a list of possible accounts. For most users there is only one option (their sponsor's default account), for these guest accounts we use the special

```
def-training-wa
```

account which is configured to allow us to start small training jobs without much delay. A sample submission might then be

```
[tyson@gra-login2 ~]$ sbatch --time 5 --mem-per-cpu 500 --account def-training-wa --output example.log igc-date flights/0144f5b1.igc
```

Submitted batch job 31352623

To avoid having to specify all these options every time, the sbatch command also lets us put them in the comments at the top of our script file after the

```
#!/bin/bash
```

line but before any [commands](#) by prefixing them with

```
SBATCH
```

```
.
```

```
#SBATCH --mem-per-cpu 500
```

```
#SBATCH --account def-training-wa
```

If an option is specified on both the [command line](#) and the script file, the [command line](#) will take precedence. In this sense, putting options in our script files gives us a powerful way to specify our defaults.

The returned number is the job identifier. Make sure to provide this to us if you ever request support for an issue regarding your job so we can look it up. It can also be used to cancel a job with the

```
scancel
```

([slurm cancel](#)) command. The

```
squeue
```

([slurm queue](#)) command shows you the status of queued jobs

```
[tyson@gra-login2 ~]$ squeue -u tyson
```

JOBID	USER	ACCOUNT	NAME	ST	TIME_LEFT	NODES	CPUS	GRES	MIN_MEM	NODELIST	(REASON)
31352623	tyson	def-training	igc-date	R	4:50	1	1	(null)	500M	gra1064	(None)

The

```
-u <username>
```

option limits the output to just the jobs queued for the specified user. From this we see that our job is running (

```
ST
```

```
is
```

```
R
```

) on the node (computer)

```
gra1064
```

. Once a job has completed running, it is removed from the queue and no longer shows up in the output of of

```
squeue
```

. Information about it can still be retrieved using the

```
sacct
```

(slurm accounting) command

```
[tyson@gra-login2 ~]$ sacct -S 2020-05-01 -u tyson
```

```
...
```

where the

```
-S <date>
```

option specifies how far back in the job records to report on (the default is just the current day). We can also look in the specified output file to get any messages that may have been printed by the job (the extracted flight date in our case)

```
[tyson@gra-login2 ~]$ cat example.log
```

```
030816
```

The supercomputers have a lot of standard software already installed on them. It is not possible to enable all these software packages at the same time though, as many provide the same [commands](#), so you need to use the

```
module
```

command to tell the system what software you want enabled. The

```
module avail
```

command lists what software is available. For example

```
[tyson@gra-login2 ~]$ module avail python
```

```
...
ipython-kernel/2.7 ipython-kernel/3.6 ipython-kernel/3.8 (D) python/3.6.10 (t,3.6) python/3.7.9 (t)
ipython-kernel/2.7 ipython-kernel/3.7 python/2.7.18 (t,2.7) python/3.7.7 (t,3.7) python/3.8.2 (t,D:3.8)
```

...
and the module

module load

command enables the choosen software

```
[tyson@gra-login2 ~]$ python --version
Python 3.7.7
[tyson@gra-login2 ~]$ module load python/3.8
[tyson@gra-login2 ~]$ python --version
Python 3.8.2
The
```

module avail

command only shows software that is compatible with the current core packages that loaded. To see all available software you need to use the

module spider

command. It will also tell you what other packages you need to load in order to make your desired package available. For example

```
[tyson@gra-login2 ~]$ module avail qgis
...
No module(s) or extension(s) found!
Use "module spider" to find all possible modules and extensions.
[tyson@gra-login2 ~]$ module spider qgis
```

```
...
Versions:
qgis/2.18.24
qgis/3.10.6
```

...
For detailed information about a specific "qgis" package (including how to load the modules) use the module's full name.

```
...
[tyson@gra-login2 ~]$ module spider qgis/3.10.6
```

...
You will need to load all module(s) on any one of the lines below before the "qgis/3.10.6" module is available to load.

```
StdEnv/2020 gcc/9.3.0
```

...

Last modified: Wednesday, 22 May 2024, 6:21 PM

Loops

Completion requirements

At the very start, we demoed how easy it was to add the date to the name of a large number of files with the [command line](#) instead of a graphical user interface. We did this using a for loop. For [loops](#) let us create a template command (such as one to renaming a file to include the date), and then apply it to a large number of cases.

Lets consider the case of getting all our pilot names. We have our

igc-pilot

command that gives us the pilot field from a single igc file. If we wanted to retrieve all our pilot, we would have to run this command once for each file. That is

```
[tyson@gra-login2 ~]$ igc-pilot flights/0144f5b1.igc
```

```
[tyson@gra-login2 ~]$ igc-pilot flights/04616075.igc
```

```
[tyson@gra-login2 ~]$ igc-pilot flights/054b9ff8.igc
```

...

Comparing these first few cases, it is pretty clear that the only thing changing between each of these [commands](#) is the name of the igc file. That is, we have a common command template that we are running

igc-pilot \$file
where

\$file

is placeholder for the filename that changes with each command. Our initial [commands](#) could then equally well be written as

```
[tyson@gra-login2 ~]$ file=flights/0144f5b1.igc; igc-pilot $file
```

```
[tyson@gra-login2 ~]$ file=flights/04616075.igc; igc-pilot $file
```

```
[tyson@gra-login2 ~]$ file=flights/054b9ff8.igc; igc-pilot $file
```

...

where we are simply setting the value

file

each time and then doing our template that runs

igc-file

on our file.

A for loop is nothing more than special syntax for doing that only requires us to have to specify the template once, which makes sense as the template is the same every time. In bash the syntax looks like this

```
for file in flights/0144f5b1.igc flights/04616075.igc flights/054b9ff8.igc ...; do
    igc-pilot $file
done
```

By bringing the list of file we run our template for into one place, we have also now made it possible for us to specify our file list using a *glob pattern*. That is, we can say

```
[tyson@gra-login2 ~]$ for file in flights/*.igc; do
    igc-pilot $file
done
Lena
Bill
Lena
...
A
```

for

statement is also a command, and can be used as any other [commands](#). For example, its output can be piped through

sort

and

uniq

in order to obtain a compact list of our pilots

```
[tyson@gra-login2 ~]$ for file in flights/*.igc; do igc-pilot $file; done | sort | uniq
```

Aasia

Bill

Fred

Lena

Mary

Mo

When we say

`$(<command>)`

in bash, it gets replaced with the output

`<command>`

. We can use this to write a powerful for loop to (finally!) rename all our files to something more useful than their current names

```
[tyson@gra-login2 ~]$ cd flights
[tyson@gra-login2 ~]$ for file in *.igc; do
    pilot=$(igc-pilot $file)
    date=$(igc-date $file)
    mv $file $date-$pilot-$file
done
```

```
[tyson@gra-login2 ~]$ ls
```

```
000000-Lena-551a9c25.igc 010815-Aasia-7cded70f.igc 010815-Mary-4bf15e07.igc ...
```

Having the pilot name and date in each of the files allows us to easily do things like select all the files for a specific pilot or year using *glob patterns* (e.g.,

```
*-Lena-*
```

will select all of Lena's flights).

Last modified: Wednesday, 22 May 2024, 6:21 PM

Exercises

Completion requirements

Doing the following [exercises](#) help ensure you understand the course material. The results of some of the later ones are needed to answer the course completion quizzes.

In this exercise we are going to create a triple loop to print out the highest altitude obtained by each pilot for each year by filling in the missing command in the following double loop

```
for year in 15 16 17; do
    for pilot in Aasia Bill Fred Lena Mary Mo; do
        maxalt=$(<command to retrieve greatest altitude for $pilot in $year>)
        echo "$year - $pilot: $maxalt"
    done
done
```

Past experience has shown this is a very tough problem for people to solve outright. When you get stuck like this, the key is to switch from trying to find an outright solution, and instead focus on trying to break the problem down into a series of smaller problems that you can produce outright solutions to. Often the steps become obvious if you do a few cases by hand. You just need to take note of what you did and get the computer to do the same.

For this problem, we can break it down into the following sub-problems (which are the exercise)

1. Extract the highest altitude for a single flight (we made a command for this).
2. Come up with a *glob pattern* to select all the flights in a given year for a given pilot.
3. Use both of these to extract the highest altitudes for a given pilot in a given year (put the highest altitude command in a *for loop* over the files that match the *glob pattern*).
4. Extract the highest of the highest flight altitudes (pipe the output of the for loop into a pipeline that extracts the largest value).
5. Insert the command you built up in 1-4 into the loop above and run that to see which pilot achieved the highest altitude each year.

Last modified: Wednesday, 22 May 2024, 6:22 PM

UNIX Reference

Key features

Completion requirements

Unix Architecture page on Wikipedia

Files are stored on disk in a hierarchical [file system](#), with a single top location throughout the system (root, or "/"), with both files and directories, subdirectories, sub-subdirectories, and so on below it.

With few exceptions, [devices](#) and some types of communications between processes are managed and visible as files or pseudo-files within the [file system](#) hierarchy. This is known as *everything's a file*.

Doug McIlroy (inventor of Unix pipes)

This is the Unix philosophy: Write programs that do one thing and do it well. Write programs to work together. Write programs to handle text streams, because that is a universal interface

Last modified: Wednesday, 22 May 2024, 6:23 PM

File system

Completion requirements

Key differences from Windows

- there are mount points instead of

A:

,

C:

, etc.,

- directories and files are case sensitive, and
- the separation character is

/

instead of

\

What would appear as a separate media hierarchy in Windows (e.g.,

A:\MyDir\MyCode.c

) simply appears under a separate directory (known as a mount point) in Unix (e.g.,

/media/disk/MyDir/MyCode.c

).

Root (

/

)

- /boot

- boot loader files

- /etc

- configuration files

- /dev

- device files

- /bin
 - user programs required for booting
- /sbin
 - system programs required for booting
- /lib{,32,64}
 - libraries required for booting
- /usr
 - programs, libraries, and such not required for booting
- /root
 - superuser directory
- /home
 - users directories (shared by all nodes)
- /tmp
 - temporary files
- /var
 - variable data (spool files, log files, etc.)
- /opt
 - add on package directory
- /media
 - mount point for removable media
- /proc
 - process information pseudo-file system
- /sys

- system information pseudo-file system

User (

/usr

)

The

/usr

directory is split off from the

/

directory mostly because disk space used to be precious.

- /usr/bin
 - user programs not required for booting
- /usr/sbin
 - system programs not required for booting
- /usr/lib_{32,64}
 - libraries not required for booting
- /usr/games
 - game programs
- /usr/share
 - architecture independent data
- /usr/man
 - on-line manuals
- /usr/src
 - source code
- /usr/include
 - header files

User Local (

/usr/local

)

The

/usr/local

directory is a place to locally install programs without messing up

/usr

.

- /usr/bin
 - user programs not required for booting
- /usr/sbin
 - system programs not required for booting
- /usr/lib{[32](#),[64](#)}
 - libraries not required for booting
- /usr/games
 - game programs
- /usr/share
 - architecture independent data
- /usr/man
 - on-line manuals
- /usr/src
 - source code
- /usr/include
 - header files

Alliance supercomputers

- /project

- group data files (shared by all nodes and all group members)

- /scratch

- user temporary data files (local to each cluster)

Last modified: Wednesday, 22 May 2024, 6:24 PM

Devices

Completion requirements

Some of the special

/dev

files are

- /dev/**null**

- discards all data written and provides no data

- /dev/zero

- provides a constant stream of *NULL* characters

- /dev/random

- provides a stream of random characters

- /dev/urandom

- provides a constant stream of pseudo-random characters

Last modified: Wednesday, 22 May 2024, 6:24 PM

Commands

Completion requirements

Programs are run by specifying the command followed by the arguments separated by spaces.

program

[

argument...

]

By convention, arguments are switches followed by strings (e.g., regexps, paths, file names, etc.). Switches are usually single dashes followed by letter for each switch or a double dash followed by a descriptive string (e.g.,

```
rm -fr mydir
```

or

```
rm --force --recurse mydir
```

). Most commands also understand

- -

- as a file name means read or write to the terminal

- --

- the end of switches and the start of the strings (in case the string needs to start with

-

or

--

).

Help

Traditionally man pages (a single help page) have been the de facto documentation source, however, some software suites have been switching to info pages (a collection of hyperlinked pages). Help for the shell built in commands is available by the built in

```
help
```

.

- man

command - on-line reference manuals

- apropos

`[-a]`

keyword ... - search on-line reference manuals (same as

`man -k`

)

- info

item - info documents

Directories

The current directory is

.

and the parent directory is

..

.

- pwd

- current directory

- cd

directory - change directory

- mkdir

directory - make directory

- rmdir

directory - remove directory

Files

Files beginning with

.

are considered hidden and not normally shown.

- ls

[-a]

[-l]

destination - list files

- cp

[-a|-p]

[-r]

[-s]

source ... destination - copy files

- ln

[-s]

target name - link to file

- mv

source ... destination - move files

- rm

[-r]

[-f]

destination ... - remove files

Permissions

Standard permissions are

r

ead,

w

rite, and e

x

ecute for

u

ser,

g

roup, and

o

ther. They are frequently abbreviated as three octal numbers (0=000, 1=001, 2=010, 3=011, 4=100, 5=101, 6=110, 7=111) corresponding to user read, write, and execute; group read, write, and execute; other read, write, and execute.

For directories,

r

ead allows the contents to be listed,

w

rite allows files to be added or removed, and e

x

ecute allows the directory to be traversed.

- `chmod`

`[u|g|o|a]`

...

`[+|-|=][r|w|x|X]`

...

`[-R]`

destination ... - change mode (user/group/other permissions)

- `chown`

`[-R]`

user destination ... - change owner

- `chgrp`

`[-R]`

group destination ... - change group

- `setfacl`

`[-m|-x]`

`[-R]`

[[u|g|o|m]...:

user

: [r|w|x|X]...

destination* ... - set file access control list *(individual users)

- getfacl

destination ... - get file access control list (individual users)

View Files

The

space

key will advance a page and the

q

key will quit in

more

and

less

. In addition, the arrow keys will move in the appropriate direction in

less

.

- more

file - view one page at a time

- less

file - view forward and backwards

- cat

[

file ...

]

- concatenate files in sequence

- head

[-n *lines*]

[

file ...

]

- first part of files

- tail

[-n *lines*] [-f]

[

file ...

]

- last part of files

- paste

[-d

delimiter

]

[

file ...

]

- concatenate files in parallel

- cut

[

-d

delimiter

]

[

-f

range

]

[

file ...

]

- extract columns

- sort

[-g]

[-f]

[-u]

[

file ...

]

- sort lines

Comparison

Digests are numbers computed from the content of files such that it is extremely difficult to come up with two different files with the same number.

- diff

[-w]

[-i]

[-u

number

[-y]

file1 file2 - compare files line by line

- sdiff

[-W]

file1 file2 - compare files side by side (similar to

diff -y

)

- md5sum

[

file ...

]

- compute MD5 digest

- sha256sum

[

file ...

]

- compute SHA256 digest

Searching

- egrep

[-i]

[-v]

regex

[

file ...

]

- find lines matching regex in files (same as

grep -E

)

- `fgrep`

`[-i]`

`[-v]`

strings

`[`

file ...

`]`

- find lines matching strings in files (same as

`grep -F`

)

- `find`

directory ... predicates - find files satisfying predicates in directories

Process

Each process (a running programs) is identified by a unique number.

- `ps`

`[-A]-U`

user

`]`

`[-H]`

[-f]

- process list

- kill

[-s

signal

]

process ... - signal process

- nohup

command - disconnect command

- nice

command - low priority command

Remote

- ssh

[

user

@]

host

[

command

]

- login to remote system

- scp

[[

user

@]

host

:]

source ...

[[

user

@]

host

:]

destination - copy remote files

- unix2dos

file ... - convert to DOS line breaks

- dos2unix

file ... - convert to Unix line breaks

Other

- sleep

seconds - waits given number of seconds

- echo

[-n]

[-e]

strings - prints strings

- test

tests - perform various string (e.g., equality) of file (e.g., existence) tests

Last modified: Wednesday, 22 May 2024, 6:25 PM

Editors

Completion requirements

The two most popular Unix editors are

vi

and

emacs

. Both are extremely powerful and very complex. A simpler editor is

nano

.

- vi

[

file ...

]

- common Unix editor

- emacs

```
[-nw]
```

```
[
```

file ...

```
]
```

- common Unix editor

- nano

```
[
```

file ...

```
]
```

- simple Unix editor

Vi

Vi distinguishes between command and insert mode. Command mode allows you to move around and enter [commands](#). Insert mode allows you to edit text.

- :h

- help

- :w[!] [

file

```
]
```

- write file (exclamation forces it)

- :e

file - edit file

- :q[!]

- quit Vi (exclamation forces it)

- :n[!]

- next file (exclamation forces it)

- [a|A]

- append after cursor or at end of line

- [i|I]

- insert (capital for beginning of line)

- [v|V]

- select to cursor or to end of line

- [c[w|c]|C]

- change selection/word/line or to end of line

- [d[w|d]|D]

- delete selection/word/line or to end of line

- [y[w|y]|Y]

- copy selection/word/line or to end of line

- [p|P]

- paste before or after cursor/line

- J

- join lines

- [u|U]

- undo (capital for current line)

- ESC

- revert to command mode

Emacs

Emacs is a more traditional single mode editor. Partially typed entries can be completed by pressing *TAB* (twice to list).

- CTRL+h

- help (

b

list keys and

k

describes keys)

- CTRL+g

- abort current operation

- CTRL+[1|2|3]

- single window or split vertical/horizontal window

- CTRL+x

CTRL+s

- save current buffer

- CTRL+x

CTRL+b

- switch current buffer

- CTRL+x

CTRL+k

- quit current buffer

- CTRL+x

CTRL+c

- quit Emacs

- CTRL+SPACE

- mark start of region

- CTRL+w

- copy from start of region to cursor

- CTRL+y

- past copied region

- CTRL+k

- delete to end of line or line if start of line

- CTRL+s

- search for text

- CTRL+_

- undo

- ALT+x

- enter command (*TAB* twice to list)

Last modified: Wednesday, 22 May 2024, 6:26 PM

Command line

Completion requirements

The shell is a command line interpreter that lets users run programs. It provides ways to start programs and to manipulate/setup the context in which they run. The main parts of this are

- arguments,
- environment,
- standard input (stdin),
- standard output (stdout),
- standard error (stderr), and
- return value

A standard command looks like so

command

[<

stdinfile

]

[>[>]

stdoutfile

]

[2>[>]

stderrfile

]

[&]

Arguments

Options passed to the program to tweak its behaviour. Traditionally switches (e.g.,

-xzf

or

--extract

--gzip

--file

) followed by strings (e.g., regexp, paths, file names, etc.). Partially typed file names and directories can be completed by pressing *TAB* (twice to list).

- ...

{

...

}

... (brace expansion) - if not quoted, expands once for each comma separated list or once for each number in

..

separated range

- ~

... (tilde expansion) - if not quoted, expands to home directory of user following the tilde or the current user if no user specified

- \${...}

(parameter and variable expansion) - if not single quoted, expands to environment variable specified or the corresponding parameter if number specified (

{

and

}

are not always required)

- \$(...)

(command substitution) - if not single quoted, expands to output for command (

`

...

`

is an alternative syntax)

- `$((...))`

(arithmetic substitution) - if not single quoted, expands to evaluated result of the expression

- ... (word splitting) - if not quoted, splits into separate arguments anywhere an IFS character (by default space, tab, and newline) occurs
- ...

`[*]?[`

...

`]]`

... (path name expansion) - if not quoted, is considered a pattern and replaced with matching file names (

`*`

matches any string,

`?`

matches any character, and

`[`

...

`]`

matches all the enclosed characters)

Quoting

Special characters can be escaped with

`\`

to remove their special meaning. Single and double quoting strings affect escaping as well as which expansions and substitutions are preformed.

- `'`

...

`'`

- no expansion or substitutions is preformed

- `"`

...

`"`

- only escaping, parameter and variable expansion, command substitutions, and arithmetic substitutions occur

Environment

A set of key value pairs (e.g.,

`USER=root`

) that programs can look up and use. Each program gets a fresh copy (i.e., changing it will not change the original) of all environment variables marked for export.

- *key*

`=`

value - make a local environment variable

- `export`

key

`[=`

value

`]`

- mark an environment variable for export

- unset

key - delete an environment variable

Two important environment variables are

- PATH

- list of

:

separated directories to look for programs in

- LD_LIBRARY_PATH

- list of

:

separated directories to look for libraries in (ahead of the system defaults specified in

[/etc/ld.so.conf](#)

)

Input and Output

Programs are run with a standard place to read input from, a standard place to write output to, and a standard place to write error messages to. By default these are all the terminal window in which the program is run. This can be changed via

- <

file - read standard input from file

- [>|>>]

file - write standard output to file (overwriting or appending)

- [2>|2>>]

file - write standard error to file (overwriting or appending)

- [&>|&>>]

file - write standard output and error to file (overwriting or appending)

Status

Programs return an integer exit status. The status of the most recent executed foreground command is available as

\$?

.

- 0 - program completed successfully
- 1...127 - program specific error code
- 128...255 - program terminated by signal 127+*signal*

Job Control

Programs run in the foreground by default. Background jobs will be suspended if they require input. Existing jobs will be sent

SIGHUP

when the shell exits.

- jobs

- list jobs

- fg

id - switch job to foreground

- bg

id ... - switch jobs to background

- disown

id ... - release jobs from job control

Foreground jobs usually respond to the following key combinations

- *CTRL+Z* - suspend program
- *CTRL+C* - abort program
- *CTRL+D* - end of input

Multiple [Commands](#)

[Commands](#) can be combined in several ways.

- ...

```
;
```

... - run first command and then second (same as pressing *ENTER*)

- ...

```
&
```

... - run first command in background at the same time as second

- ...

```
|
```

... - run first command in background with its output going to the second as input

- ...

```
&&
```

... - run first command and then second only if first returns success

- ...

```
||
```

... - run first command and then second only if first returns failure

[Commands](#) can be combined in several ways.

- {

```
...
```

```
}
```

- group command in current shell -- has to end with

```
;
```

or newline

- (

...

)

- group command in sub shell -- does not have to end with

;

or newline

Last modified: Wednesday, 22 May 2024, 6:26 PM

Scripting

Completion requirements

Executable text files that start with

#!/

command (#!/bin/bash for shell scripts) are run as *command file*.

Parameters

- \$#

- number of parameters

- \$0

- name of shell or shell script

- \$

number - positional parameter

- \$*

- all positional parameters (in double quotes expands as one argument)

- \$@

- all positional parameters (in double quotes expands as separate arguments)

The following functions manipulate parameters

- shift

```
[  
  
  number  
  
]
```

- drop specified number of parameters (one if unspecified)

- set

parameter ... - set parameters to given parameters

Programming

- if

command ...

; then

command ...

; [elif

command ...

; then

command ...

;]

...

[else

command ...

;] fi

- conditionally run [commands](#) depending on success

if

and

elif

[commands](#)

- **for**

key

in

value ...

; do

command ...

; done

- for each value, set key to value and run [commands](#)

- **while**

command ...

; do

command ...

; done

- repeatedly run [commands](#) until

while

[commands](#) fail

- **case**

value

in [

pattern

```
[
```

pattern

```
]
```

...

```
)
```

command ...

```
::]
```

...

```
esac
```

- run [commands](#) where first pattern matches (same as path name expansion)

- continue

```
[
```

number

```
]
```

- next iteration of enclosed loop (last if not specified)

- break

```
[
```

number

```
]
```

- exit enclosed loop (last if not specified)

- **function**

name

{

command

;}

... - create a command that runs the [commands](#) with passed parameters

- **return**

[

number

]

- return from function with given exit status (last command if not specified)

- **exit**

[

number

]

- quit shell with given exit status (last command if not specified)

Last modified: Wednesday, 22 May 2024, 6:27 PM

Regular expressions

Completion requirements

Regular expressions are strings where several of the non-alphanumeric characters have special meaning. They provide a concise and flexible means for string searching and replacing and are used by several Unix programs.

Anchoring

- `^`
- match start of line
- `$`
- match end of line

Characters

- *character* - the indicated character

- `.`
- any character

- `[`
...
`]`
- any character in the list or range (
`^`
inverts)

Combining

- `(`
...
`)`
- group
- `...`
`|`
... - match either or

Repetition

- `?`

- match zero or one times

- `*`

- match zero or more times

- `+`

- match one or more times

- `{`

...

`}`

- match a range of times

Replacement

- `\`

digit - substitute text matched by corresponding group

Last modified: Wednesday, 22 May 2024, 6:27 PM