

Introduction to GNU Linux

Reference Manual

Information and Communication Technology Services

Support Services

University of the Free State

Author: Albert VAN ECK

Revised: October 2019



Contents

1	Accessing Linux from a remote host	1
1.1	Console	2
1.2	Connecting from Windows	4
1.2.1	PuTTY	4
1.2.2	Xming	7
1.2.3	Transferring Files	8
1.3	Connecting from Linux or Apple Mac	11
1.3.1	SSH	11
1.3.2	Transferring Files	12
1.4	Connecting from a Web Interface	13
1.4.1	Apache Guacamole	13
1.4.2	Using Guacamole	14
2	Linux General Usage	17
2.1	Introduction to Linux	18
2.1.1	Case sensitivity and restrictions	18
2.1.2	Special Characters	18
2.2	File System and Permissions	19
2.3	Bash Shell	21
2.4	Shell Environment	21
2.4.1	Variables	21
2.4.2	Auto complete	24
2.4.3	Shortcuts	25
2.4.4	Alias	26
2.5	Getting Help	28
3	Linux Commands	29
3.1	Commands Overview	30
3.2	Introduction	30
3.2.1	Command return codes	31
3.2.2	Logical AND/OR testing	32
3.3	Using output as input	33
3.4	Command Examples	34
3.4.1	ssh	34
3.4.2	scp	35
3.4.3	rsync	35
3.4.4	cd	36
3.4.5	pwd	36
3.4.6	ls	37

3.4.7	mkdir	37
3.4.8	cp	38
3.4.9	mv	38
3.4.10	rm	38
3.4.11	cat	39
3.4.12	sort	40
3.4.13	echo	41
3.4.14	more	41
3.4.15	less	42
3.4.16	head	42
3.4.17	tail	42
3.4.18	cut	43
3.4.19	find	43
3.4.20	grep	43
3.4.21	screen	44
3.4.22	info	45
3.5	Mathematical Arithmetic	45
4	Editors	49
4.1	Introduction	50
4.2	Graphical Editors	50
4.2.1	gedit	50
4.3	Shell Editors	51
4.3.1	nano	52
4.3.2	vi	53
5	Regular Expressions	57
5.1	Overview	58
5.1.1	Character sets	58
5.1.2	Character classes	58
5.1.3	Anchors	59
5.1.4	Modifiers	59
5.1.5	Examples	59
6	Shell Scripting	61
6.1	Overview	62
6.2	Logical Testing	64
6.2.1	If statement	66
6.2.2	Case statement	67
6.3	Loops	67
6.3.1	For Loop	67
6.3.2	While Loop	68
6.4	Reading input from different sources	68

6.4.1	Reading values from shell environment variables	69
6.4.2	Working with substring parts of shell variables	70
6.4.3	Reading input from the user	71
6.4.4	Reading content from a file	72
6.4.5	Reading parameters and options from shell	73
6.5	Functions	75
6.6	Trapping signals	75
7	High Performance Computing	79
7.1	Overview	80
7.2	Creating a submit File	80
7.3	Submitting a job	82
7.4	Monitor the status of a job	83
7.5	Cancelling a job	83
7.6	Getting job output	84
7.7	Viewing Queues	85
7.8	Viewing nodes	85
	Bibliography	89
	Index	93

Preface

This text shed some light on the general use of GNU Linux. This text is part of an introductory workshop that shows computer sciences students how to manage a GNU Linux system. This text is

This text also briefly explains some HPC concepts to assist researchers with some of the terminology used in High Performance Computing.

Text Layout:

Chapter 1

An introduction to accessing Linux from remote machines.

- * Some tools used to connect from Windows machines to Linux
- * Connecting from an other Linux machine
- * Transferring files between machines over a network

Chapter 2

A brief overview of Linux and the Linux Shell Environment.

- * Linux restrictions and limitations
- * Escaping special characters
- * File System and permissions
- * The Linux shell
- * The Linux shell environment
- * Getting help

Chapter 3

Some of the commands used on a regular basis.

- * Command return codes
- * Logical testing while using commands
- * Examples of some commands

Chapter 4

Some of the Linux editors explained

- * Using an advanced editor such as vi

Chapter 5

Introduction to regular expressions

- * Character sets
- * Character classes
- * Anchors
- * Modifiers
- * Examples

Chapter 6

* The use for shell scripting

- * Testing
- * Loops
- * Functions
- * Trapping Signals

Chapter 7

Commands used to run jobs on a HPC

- * Creating a submit script
- * Submitting a job
- * Cancelling a job
- * Monitoring job status
- * Viewing Output

Some chapters include source code such as the following example:

Listing 1: Example of source code

```
1 #This is a very long comment that continues onto the next line,  
   without that line being numbered  
  
2  
3 echo "This is an echo command"  
4 This is an echo command  
5 echo -n "User input will be required here: ";read Input  
6 User input will be required here: Albert
```

In this example listing (Listing 1), each line is numbered, even empty lines
Line 1 starts with a hash (#) and represents a comment line
Line 2 is a clear line
Line 3 is a command that should be executed
Line 4 shows the output of the previous command executed on line 3
Line 5 executes a command to emulate a wizard
Line 6 displays the “wizard’s question” with the user’s response in bold

Some of the output will be omitted from this text, but parts of the output may be given where the author deemed it necessary.

The commands are referenced and can be looked up in the Index at the back of this text, under the entries:
command

Acknowledgments

A number of people were involved in the preparation of this text and the presentation of the workshops which were the initiator of the compilation of this text. Some people are thanked by name for their input, due to the vast role they played in the workshop and this text.

The author would like to thank the following people for all the work that they have done in an effort to compile this text and in the arrangements made for the workshops.

HPC Unit of the University of the Free State - Support,
ICT Services - Assistance and assistants,
Vic Coetzee - Financial aid and support,
Chris Linström - Financial aid and support,

Without these people, the workshops and this documentation would not have been possible.

-Thank you all

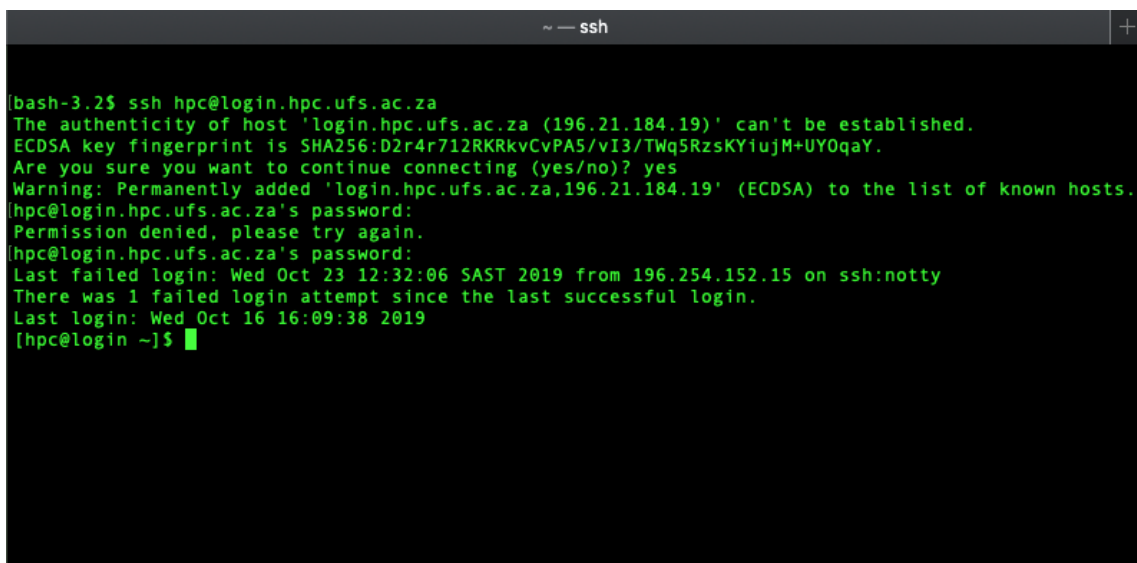
Accessing Linux from a remote host

Contents

1.1	Console	2
1.2	Connecting from Windows	4
1.2.1	PuTTY	4
1.2.2	Xming	7
1.2.3	Transferring Files	8
1.3	Connecting from Linux or Apple Mac	11
1.3.1	SSH	11
1.3.2	Transferring Files	12
1.4	Connecting from a Web Interface	13
1.4.1	Apache Guacamole	13
1.4.2	Using Guacamole	14

1.1 Console

The most used interface to a Linux system is the console or also known as a terminal. A console, or terminal, is a text based interface to a system. Figure 1.1 shows an example of a Linux console. A console is used to enter commands to be executed and to view the results on the screen. When one opens a console, a new session is started. All the commands that are executed, executes inside that same session and are killed off (by default) when that session is closed Eg. when the console is closed or the connection to the server is lost.



```
~ — ssh
[bash-3.2$ ssh hpc@login.hpc.ufs.ac.za
The authenticity of host 'login.hpc.ufs.ac.za (196.21.184.19)' can't be established.
ECDSA key fingerprint is SHA256:D2r4r712RKRkvCvPA5/vI3/TWq5RzsKYiuJM+UY0qaY.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'login.hpc.ufs.ac.za,196.21.184.19' (ECDSA) to the list of known hosts.
hpc@login.hpc.ufs.ac.za's password:
Permission denied, please try again.
hpc@login.hpc.ufs.ac.za's password:
Last failed login: Wed Oct 23 12:32:06 SAST 2019 from 196.254.152.15 on ssh:notty
There was 1 failed login attempt since the last successful login.
Last login: Wed Oct 16 16:09:38 2019
[hpc@login ~]$
```

Figure 1.1: A console connection

In Figure 1.1, the command `ssh hpc@login.hpc.ufs.ac.za` was executed. If a connection to a server is made for first time, the two machines will exchange host certificates. This will ensure that the next time a connection is made, the connection is made to the same machine as expected, or a warning will be shown to warn a user that he/she is connecting to a different machine than expected. The user have to type "yes" to import the certificate to the list of know certificates. On the sixth line, the user is prompted for his/her password. Note that the password is not displayed while typing, for security reasons. In this case, the password was typed incorrectly and therefore the user has to retype the password again. After successfully logging in; some feedback regarding when and from where the last login was made is displayed on the screen. The last line prompts the user for the next command to be executed on the remote host.

When a user is logged into a Linux system, the shell prompt may change to indicate some information regarding the path in which the user is currently working, which user is logged in and so on. Listing 1.1 dissects the same command line, breaking it up into the parts that a command line is usually made up of.

Listing 1.1: Shell Explained

```
1 [hpc@login jobs]$ ssh hpc@login.hpc.ufs.ac.za #Local Username
2 [hpc@login jobs]$ ssh hpc@login.hpc.ufs.ac.za #Local Hostname
3 [hpc@login jobs]$ ssh hpc@login.hpc.ufs.ac.za #Current Dir
4 [hpc@login jobs]$ ssh hpc@login.hpc.ufs.ac.za #Normal User
5 [hpc@login jobs]$ ssh hpc@login.hpc.ufs.ac.za #Command
6 [hpc@login jobs]$ ssh hpc@login.hpc.ufs.ac.za #Parameters
```

1.2 Connecting from Windows

1.2.1 PuTTY

When a user needs to log in to a Linux server from his/her computer (Windows), a third party application is required. The most widely used Windows application to connect to Linux is called PuTTY. “PuTTY is an SSH and Telnet client, developed originally by Simon Tatham for the Windows platform. PuTTY is open source software that is available with source code and is developed and supported by a group of volunteers.”[Tatham 2010]

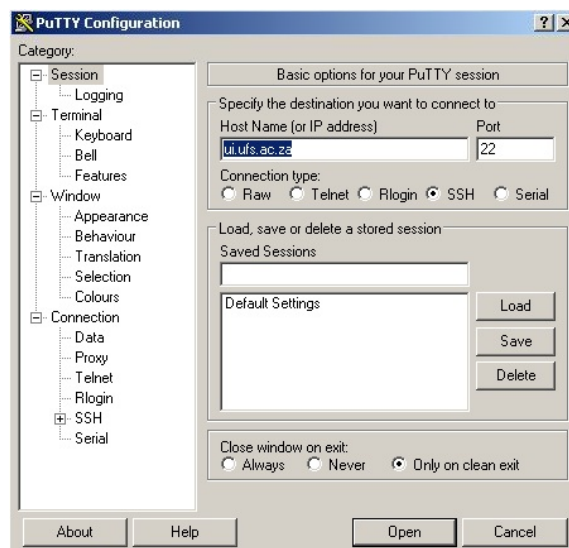


Figure 1.2: Default PuTTY login screen

When opening PuTTY, one can configure the default settings or add multiple sessions to an easy access list. Figure 1.2 shows the Default Login screen. In the Host Name box, a remote host-name can be entered to which the connection should be made. In this example the host-name “*ui.ufs.ac.za*” is used.

1.2.1.1 Configuring PuTTY

After installing PuTTY, a connection could be made to a remote host without further configuration. Although no additional configuration is required to use

PuTTY, it is recommended to configure the following two settings.

1.2.1.2 Configuring PuTTY X11 forwarding

As described in Section 1.1, a console only contains text. Sometimes a user requires more than just a text interface to a Linux system. In these cases, the user will require to export a graphical interface to the local computer's screen. Exporting graphics to an other host is called "X Forwarding".

PuTTY allows X Forwarding by enabling it under the SSH Settings in the main PuTTY interface. However, an other application called "Xming" (Section 1.2.2) is required to implement the X Forwarding but PuTTY needs to have this option enabled first to work. Figure 1.3 shows the required settings to enable X11 Forwarding.

After setting this option, if you would like to reuse the setting for any new session created by PuTTY, you have to save these settings as described in Section 1.2.1.4.

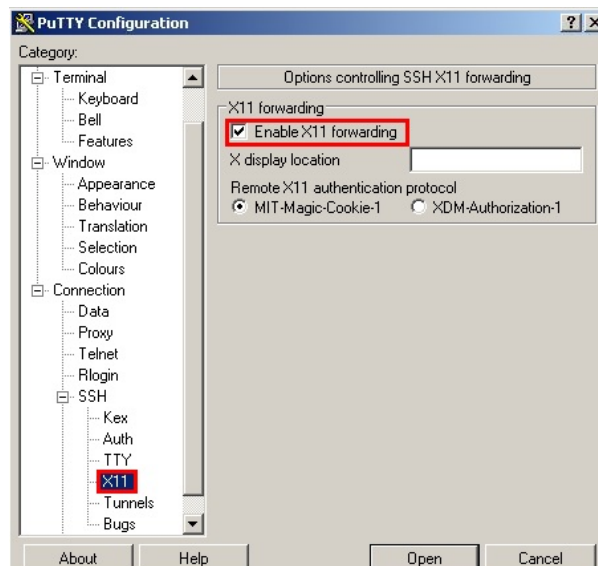


Figure 1.3: PuTTY - Enabling X11 Forwarding

1.2.1.3 Configuring PuTTY Mouse Behavior

Linux has a helpful feature for copying and pasting text using your mouse. By just highlighting text, the selected text is copied to a special clipboard (This is not the same clipboard that is used when you highlight text, right click and select copy). When the text is highlighted, you can paste it in a Linux console simply by clicking your mouse's middle button. By double clicking on a word, that word is highlighted. By triple clicking, the whole line is highlighted. Figure 1.4 shows the required setting to enable these features.

After setting this option, if you would like to reuse the setting for any new session created by PuTTY, you have to save these settings as described in Section 1.2.1.4.

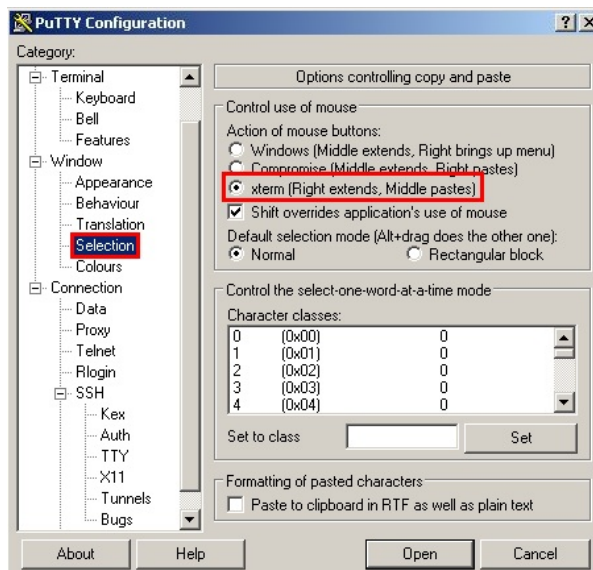


Figure 1.4: PuTTY - Enabling Mouse Copy and Paste

1.2.1.4 Saving PuTTY Settings

If any settings are changed in PuTTY, those settings are lost when PuTTY is restarted unless they are saved to a Session. If settings are changed for instance

settings described in Section 1.2.1.1, the settings can be saved to a Session by highlighting the session name and clicking on Save. There is also a text box in the Sessions section where the user can type in a session name and click on save to store a new session. In Figure 1.5, the default session is selected and Save is clicked afterwards. By highlighting “Default Settings“ and clicking ”Save”, the current settings will automatically be loaded each time PuTTY starts.

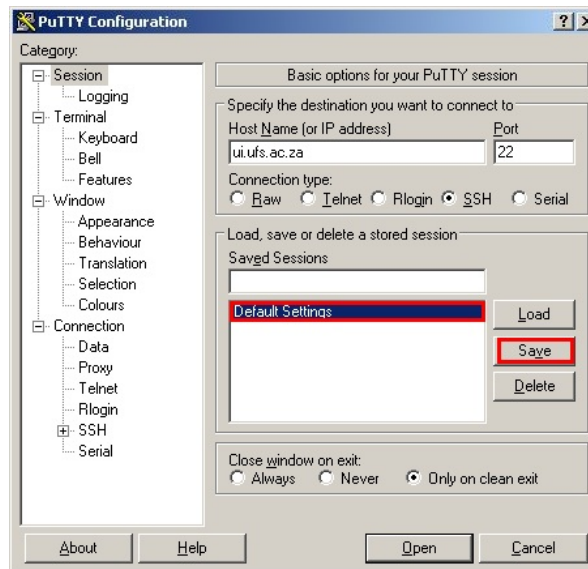


Figure 1.5: PuTTY - Saving default session profile

1.2.2 Xming

“Xming is the leading X Window Server for Microsoft XP/2008/Windows 7. It is fully featured, small and fast, simple to install and because it is standalone native Microsoft Windows, easily made portable (not needing a machine-specific installation) ”[Geeknet 2011].

Xming can be used together with PuTTY to export graphics from Linux to Windows (X11 Forwarding). In Section 1.2.1.2, the required configuration for PuTTY was described. After PuTTY is configured, Xming can be installed and

started up. Xming must be started before a PuTTY session is opened for Xming to function correctly. The installation of Xming is straight forward and the user can use all the default settings when prompted.

To test that Xming and PuTTY works correctly, a user can connect to a Linux Machine and execute one of the following commands:

xclock

xterm

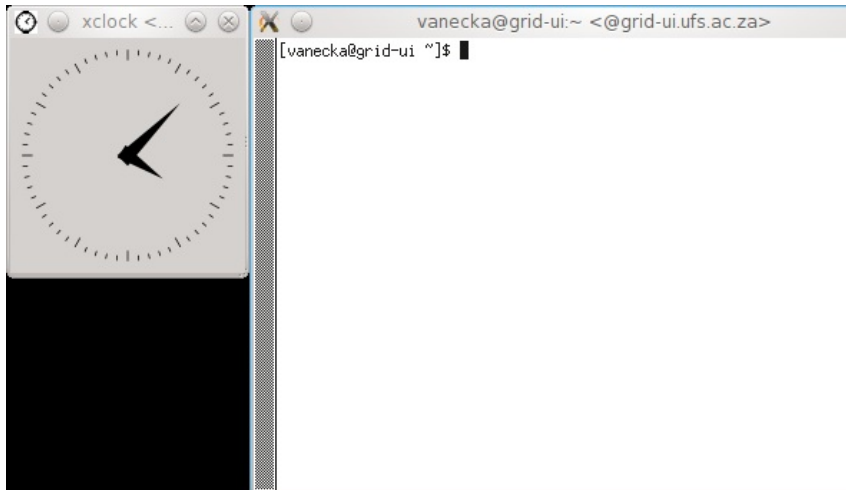


Figure 1.6: Xming - X Forwarded graphics

Figure 1.6 shows on the left hand side the “xclock” and on the right hand side the “xterm” interfaces.

1.2.3 Transferring Files

1.2.3.1 WinSCP

To transfer/copy files to and from Linux, a third party application called WinSCP could be used. Linux uses the SSH protocol to securely transfer files between systems. “WinSCP is an open source free SFTP client, SCP client, FTPS client and FTP client for Windows. Its main function is file transfer between a local and a remote computer.”[Prikryl 2011]

The installation of WinSCP is straight forward and the default settings of the installation wizard should suffice.

Figure 1.7 shows a filled out version of the login screen of WinSCP.

Figure 1.8 Shows the WinSCP file manager screen. The user's local computer is listed on the left hand side and the remote host's files and directories are listed on the right hand side.

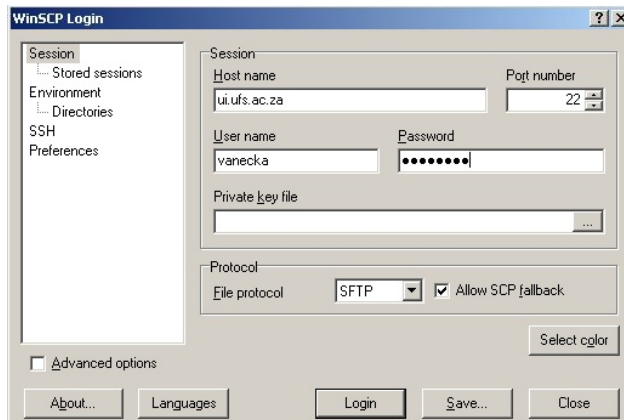


Figure 1.7: WinSCP - Login Screen

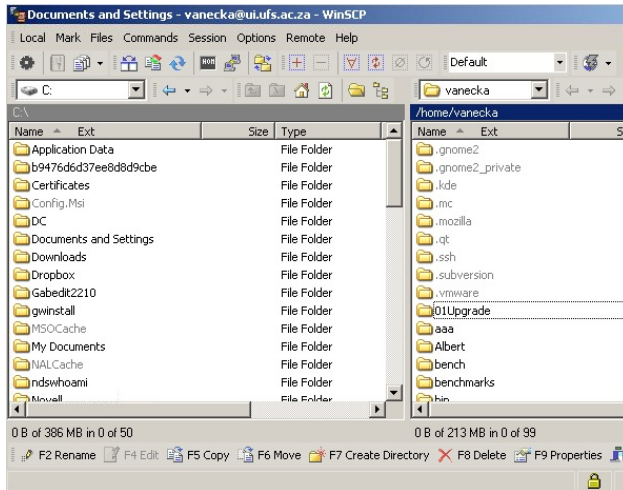


Figure 1.8: WinSCP - File Manager Screen

1.3 Connecting from Linux or Apple Mac

1.3.1 SSH

By default the majority of Linux and Apple Mac systems use a protocol called SSH (Secure SHell) to connect to each other. SSH is an encrypted connection which means that any communication that flows between two machines, are firstly encrypted before transfer and decrypted after the message is received. By encrypting all communication between servers or computers, unprivileged people can't see what exactly is contained in the message and thus securing communication. By default (if the services and firewalls are configured to do so) any two Linux machines can connect to each other over SSH, without the need of a third party application.

Listing 1.2: Shell Explained

```
1 #Log into a remote Host (testnode) and enable X Forwarding
2 ssh -XC testnode
3 xclock&
4 [1] 7147
5 exit
6 [1]+  Done                  xclock
7 Connection to testnode closed.
8
9 #Log into a remote Host (testnode) as a different username (testuser)
10 ssh testuser@testnode
11 xclock
12 Error: Can't open display:
```

Listing 1.2 shows how to connect from one Linux machine to another Linux machine using SSH. This method is the same for a Apple Mac machine to a Linux machine. Line 2 shows the **ssh** command with the **-XC** arguments. The "X" argument tells ssh to enable X Forwarding. The "C" option tells ssh to Compress the connection to save bandwidth on slow networks.

Line 3 executes the **xclock** command in the background (**&**).

Line 4 displays the Process Identifier of the **xclock** command that runs in the background.

Line 5 exits the connection to the remote host.

Line 6 shows that the background process [1] is also terminated.

Line 10 opens a new ssh connection as a different user (testuser) without X Forwarding

Line 12 displays an error message that X Forwarding is not working, which is to be expected because the SSH session (on line 10) was not opened with the "X" argument.

1.3.2 Transferring Files

Linux transfers files and directories through the SSH protocol between Linux systems. By making use of SSH, all the communication between the two machines is encrypted. The downside of this is that the encryption and decryption takes up system resources. This slows the transfer down. However, on a Local Area Network (LAN) one can easily transfer files from one machine to another using SSH at a rate of 15MB/s or more.

The command used to transfer files/directories between machines is *scp*. Listing 1.3 shows how *scp* is used to transfer files and directories between two systems.

Listing 1.3: Transferring files with scp

```
1 #Copy a file from the local machine to a remote machine:
2 scp LocalFile.dat testhost:
3 LocalFile.dat          100% 2000KB   2.0MB/s   00:00
4
5 #Copy a file from a remote host to the local machine:
6 scp testhost:RemoteFile.dat .
7 RemoteFile.dat        100%   20MB   9.8MB/s   00:02
8
9 #Copy a local directory and subdirectories to a specific path
10 #on a remote machine
11 scp -r LocalDirectory testhost:/tmp/RemoteDirectory
12 LocalFile.dat          100% 2000KB   2.0MB/s   00:00
13
14 #Copy a remote directory and subdirectories to a specific path on the
15 #local machine
16 scp -r testhost:RemoteDirectory /tmp/FromRemoteHost
17 RemoteFile.dat         100%   20MB  19.5MB/s   00:00
18
19 #Copy a specific file to a specific directory:
20 scp testhost:/etc/hosts /tmp/remote_hosts
21 hosts                  100% 7036     6.9KB/s   00:00
22
23 #Copy 2 files from the remote host, to a specific path:
24 scp testhost:/etc/{hosts,group} /tmp
25 hosts                  100% 7036     6.9KB/s   00:00
26 group                  100% 2915     2.9KB/s   00:00
```

On line 2 of listing 1.3, the file (*LocalFile.dat*) is copied to the remote server (*testhost*).

On line 2 the full path is not specified and is only defined as a colon (“:”). If the “:” is not followed by a path, as is the case of line 2, the file is copied to the home directory of the user on the remote host.

On line 6, the file (*RemoteFile.dat*) in the home directory (colon followed only by

a file name) of the remote user is copied to the current directory , which is indicated by the period (.).

Line 24 copies two files (*/etc/hosts* and */etc/group*) of the remote host to the */tmp* directory on the local host.

1.4 Connecting from a Web Interface

Sections 1.2 and 1.3 describe how to connect to a Linux system from a specific Windows, Linux or Apple Mac machine. It is also possible to connect to a Linux server using a Web Interface. This however is not standard procedure for it entails a great effort from the systems administrator to configure such a system. Fortunately, such an implementation is available to the research community via the Apache Software Foundation's Guacamole gateway.

1.4.1 Apache Guacamole

“Apache Guacamole is a clientless remote desktop gateway. It supports standard protocols like VNC, RDP, and SSH. We call it clientless because no plugins or client software are required. Thanks to HTML5, once Guacamole is installed on a server, all you need to access your desktops is a web browser.”[[Apache 2019](#)]

By making use of Guacamole, a user can gain access to the HPC using any device with a web browser installed. The inner workings of being able to use Guacamole will not be described in detail but the following services are required to make it all work: Bash Scripts, LDAP servers, Singularity, MATE Desktop, Environmental Modules, PBS Torque, VNC Server, HAProxy, Web servers, Firewall exclusions and off course Apache Guacamole itself.

Figure 1.9 shows a representation of the Guacamole user interface.

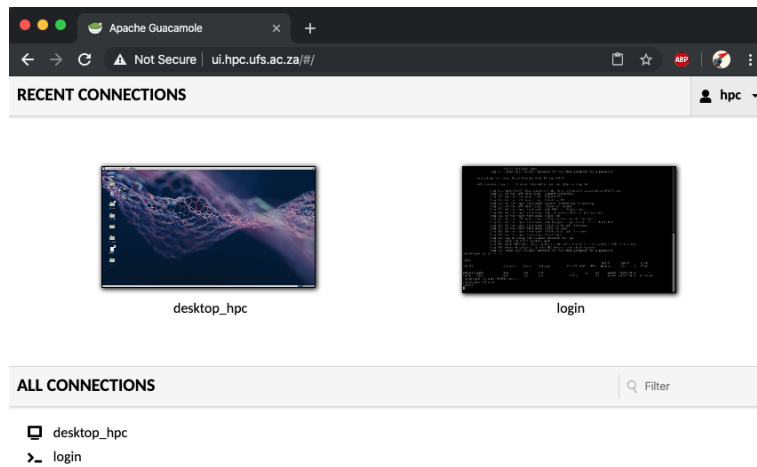


Figure 1.9: Guacamole - Graphical Web interface to the HPC

Using the Guacamole system itself should be intuitive to most researchers. However, due to the complexity of what happens in the background, the user will have to submit a VNC job to gain access to the VNC interface.

1.4.2 Using Guacamole

Guacamole works together with SSH and VNC to present a user interface. The SSH terminal provided by Guacamole can be used to execute commands on the HPC. This is useful when only a few commands have to be executed or when a user is using a system that does not have an SSH client installed on it.

To make use of a Graphical user interface, one has to have a VNC session running on the cluster first. To start a VNC session, the user has to execute a customized script called *qvnc*. The *qvnc* command will submit a VNC job to the queuing system and provide more information to connect using a VNC client. If one is using the Guacamole interface, then it is not necessary to use any of the provided configuration options but only to connect to the Guacamole web gateway.

Listing 1.4 shows the initial commands required to start a VNC session for a graphical interface.

Listing 1.4: Starting a VNC session

```
1 #Initialiaze a VNC session:
2 qvnc
3
4 #After the job is submitted, you can monitor the status thereof using
5 # the qstat command:
6 qstat
```

After executing the **qvnc** command, the connection to a VNC session should be available. The user can then use the instructions provided by the **qvnc** command or opt to open the Guacamole web gateway. At the time of writing the link to the Guacamole interface is: <https://ui.hpc.ufs.ac.za>

While the VNC session is active, any commands or processes running in the VNC session, will remain running in the background even if the Guacamole interface or the VNC client is closed. As long as the VNC job remains running, all the open applications and processes will continue running in the background. If the user logs out of the Linux system inside the VNC session, the job will be cancelled and the resources will be freed for other users to make use of.

Guidelines for using Guacamole to connect to the cluster

- A VNC session should be running when attempting to connect to the graphical interface
- The text based Guacamole interface, PuTTY or any other third party SSH terminal application can be used to submit the VNC job
- If the VNC session is closed by logging out of the Linux environment itself, the whole VNC session is terminated
- If the VNC session itself is terminated, all running processes will also be terminated
- All VNC sessions are started and running through the queuing system
- If the queued job is terminated or runs out of walltime, the VNC session and all processes are terminated

Linux General Usage

Contents

2.1	Introduction to Linux	18
2.1.1	Case sensitivity and restrictions	18
2.1.2	Special Characters	18
2.2	File System and Permissions	19
2.3	Bash Shell	21
2.4	Shell Environment	21
2.4.1	Variables	21
2.4.2	Auto complete	24
2.4.3	Shortcuts	25
2.4.4	Alias	26
2.5	Getting Help	28

2.1 Introduction to Linux

2.1.1 Case sensitivity and restrictions

It is important to know that Linux **commands, file names and directory names are case sensitive**. This is not the case with other Operating Systems such as Microsoft Windows. It is therefore possible to have a file called “MyFile.txt” and “myfile.txt” in the same directory on a Linux system. However, should a user decide to copy both files in the same directory to a Windows machine, the files will override each other.

The same case restrictions apply to commands. Most Linux commands are written in small letters (a..z). Because of the case sensitivity of commands, if a user types in “LS” instead of **ls**, the command will fail to execute.

It should be noted that a file and directory can not have the same name in the same path. For instance, if a user creates a directory called “results”, then the user will not be allowed to create a file called “results” in the same (current) directory.

A user can create a directory or a file with a space in the name but it is not recommended. However, if a user absolutely has to create a directory with a space in it; the user can enclose the directory name in quotes for instance:

mkdir "My Directory With Spaces"

or the user can create the directory by escaping the space character with a leading backslash (“\”). For instance:

mkdir My\ Directory\ With\ Spaces

If possible; it is recommended to not make use of any special characters, upper case characters or spaces in file or directory names.

2.1.2 Special Characters

Special Characters are almost any character that is not an alpha numeric character ([0-9], [a-z] or [A-Z]). These characters has special meanings and are interpreted by the shell as having a special meaning. For instance the "[" character is interpreted by the shell as a command and not as a normal character. The following table shows some special characters:

Some Special Characters		
colon (:)	comma (,)	pipe ()
slash (/)	backslash (\)	space ()
semi-colon (;)	tilde (~)	dollar (\$)
asterisk (*)	quotes (' ')	double quotes (" ")

Special characters should be escaped to not be interpreted by the shell but to rather just be displayed. One can escape most of these characters by using a back slash (\) in front of the character.

The interpretation of a quote and a double quote also differs.

Listing 2.1: Interpretation of special characters

```
1 export Amount=100
2
3 echo "The value of $Amount is $Amount."
4 The value of 100 is 100.
5
6 echo 'The value of \ $Amount is $Amount.'
7 The value of \ $Amount is $Amount.
8
9 echo "The value of \ $Amount is $Amount."
10 The value of $Amount is 100.
```

Listing 2.1 shows the difference methods the shell interprets commands and quotes.

On Line 3, the echo command interprets the \$Amount as 100 for both \$Amount entries because of the double quotes.

On Line 6, the echo command does not interpret the \ \$Amount nor the \$Amount entries because of the single quotes.

On Line 9, the echo command does not interpret the \ \$Amount because it is escaped but the \$Amount is interpreted as a variable and is replaced by the value 100 because of the escape characters and double quotes.

2.2 File System and Permissions

The Linux file system is based on the POSIX standards. Originally, the name "POSIX" referred to IEEE Standard 1003.1-1988. POSIX is a UNIX standard that was created in 1988 that stipulates access permissions and standards for UNIX based Operating Systems and File Systems.

Linux has a number of file system types. Currently Ext4 and Xfs are the norm used by Linux. Ext4 and Xfs both allow permissions to be set on files and directories. A user needs to have access to a file to be able to read or write to it. A user also needs to have executable permission on a directory to change into that directory.

The following File and Directory permissions can be set:

File/Directory Permissions:

r : File/Directory is **r**eadable
 w : File/Directory is **w**riteable
 x : File/Directory is **e**xecutable
 - : No permissions are set

A numeric representation of permissions is also used to set permissions with the **chmod** command.

Further more, each of the above permissions can be set per *owner*, *group* and *world/global*. Listing 2.2 reflects the permissions of the same directory, each line describes a specific part of the permission.

Listing 2.2: Example of a directory's permissions

```

1 #The type is a directory
2 drwxr-x--- 2 testuser hpcuser    3 Jul  7 19:56 LocalDirectory
3
4 #Owner permission (Readable,Writeable,Executable)
5 drwxr-x--- 2 testuser hpcuser    3 Jul  7 19:56 LocalDirectory
6
7 #Group permission (Readable,Executable, but not executable)
8 drwxr-x--- 2 testuser hpcuser    3 Jul  7 19:56 LocalDirectory
9
10 #World permission (none)
11 drwxr-x--- 2 testuser hpcuser    3 Jul  7 19:56 LocalDirectory
12
13 #Owner
14 drwxr-x--- 2 testuser hpcuser    3 Jul  7 19:56 LocalDirectory
15
16 #Group owner
17 drwxr-x--- 2 testuser hpcuser    3 Jul  7 19:56 LocalDirectory

```

To see the permissions of a directory or a file or directory, the command “**ls -l**” can be used. Listing 2.3 shows the long listing of directory with the files and subdirectories in the directory.

Listing 2.3: File Permissions

```

1 ls -l
2 total 22147
3 drwxr-x--- 2 testuser hpcuser      3 Jul  7 19:56 LocalDirectory
4 -rw-rw-r-- 1 testuser hpcuser 2048000 Jul  7 20:00 LocalFile.dat
5 lrwxrwxrwx 1 testuser hpcuser      13 Jul 11 2011 LocalFileLink.dat
   -> LocalFile.dat
6 drwxr-xr-x 2 testuser hpcuser      3 Jul  7 20:02 RemoteDirectory
7 -rw-r----- 1 testuser hpcuser 20480000 Jul  7 20:01 RemoteFile.dat

```


2.3 Bash Shell

Chapter 1 Section 1.1 describes the Linux console. As mentioned the console is the interface to the Linux Shell. The outline of this document is based on one type of shell used by most Linux users called Bash.

“Bash is a Unix shell written by Brian Fox for the GNU Project as a free software replacement for the Bourne shell (sh). Released in 1989, it has been distributed widely as the shell for the GNU operating system and as the default shell on Linux, Mac OS X and Darwin” [Wikipedia 2011]

Another shell type is C-Shell. The main difference between the two shells is the way that environmental variables are set and some other functionality. Section 2.4 explains the working and functionality of the shell environmental variables.

2.4 Shell Environment

2.4.1 Variables

When a new shell is opened, default settings for the session are loaded from the profile file `/etc/profile` and the bash configuration script `/etc/bashrc`. One of the more important settings loaded from this file is the path. The path is a series of directories searched when a command is called from the shell. The value of the search paths are stored in an environmental variable (PATH). The value of the PATH environmental variable can be viewed by echoing the value to the screen. Listing 2.4 shows how the **echo** is used to display the value of the PATH variable.

Listing 2.4: The PATH Environmental Variable

```
1 echo $PATH
2 ~/bin:/usr/local/bin:/bin:/usr/bin:/home/testuser/bin
```

Line 1 of Listing 2.4 executes the **echo** command with the **\$PATH** option. The value of the variable is displayed on the next line. When a command such as **ls** is executed in the shell, the path is searched to find a command (executable file) firstly in `/bin`. If the file/command does not exist, the next path is searched. In this example `/usr/local/bin` is searched and so on until the command is found. In this example the command **ls** is found in `/bin/ls`.

Listing 2.5: Setting Environmental Variables

```
1 MY_NAME=Albert
2 MY_SURNAME="van Eck"
3 export MY_NAME MY_SURNAME
4
5 echo "Hello $MY_NAME $MY_SURNAME."
```

```
6 Hello Albert van Eck.  
7 unset MY_NAME MY_SURNAME
```

Listing 2.5 shows how variables are set.

On Line 1, a variable is created and its value is set.

On Line 2, a variable is created and a multiple value (two words) is set.

On Line 3, the variables are exported. A variable needs to be exported to be used in different newly created sessions.

On Line 5, the variables are used and the output is displayed on the next line.

On Line 7, the variables' values are removed from memory and the variables itself are removed.

It should be noted that variables are also case sensitive. This means that a variable called "MY_NAME" can be set and have a different value from a variable called "My_Name". It should also be noted in Listing 2.5 that there is no space between the variable name, the equal sign (=) and the value. Also note that a variable name may not contain special characters such as spaces, but the values may. A variable name may also not start with a digit (0-9) but the name may contain digits after the first character. A variable name can start with, or contain underscore(s).

A full listing of all environmental variables can be seen by executing the **set** command. Listing 2.6 shows the **set** command and some of the more useful variables.

Listing 2.6: Set Environmental Variables

```
1 set  
2 BASH=/bin/bash  
3 ...  
4 HOME=/home/testuser  
5 HOSTNAME=grid-ui.ufs.ac.za  
6 LINES=60  
7 PWD=/home/testuser  
8 USER=testuser  
9 ...
```

Environmental variables are used to store values for further reference. Environmental variables can be used for instance to refer to a user's home directory in a shell script. For each user, the user home directory will be different and therefore the value is not predetermined. The script can reference *\$HOME* instead of the value */home/testuser*.

As mentioned, the profile (*/etc/profile*) is used to set environmental variables each time a user logs in or a new session is started. There are also other configuration files that are loaded each time a session is started. Under */etc/profile.d/* a set of

configuration files exist that sets the environment. The profile file (`/etc/profile`) and profile configuration files under `/etc/profile.d/` are only configurable by the system administrator.

There are also configuration files that the user can modify to set the user's environment. The configuration files are located in the user's home directory and is "hidden" from display by starting with a period "`.`". The files are `.bashrc` and `.bash_profile`. If a variable is exported using the `export` command, that variable can be used in all new processes that is started from the active session. When a session starts a new session or process, the session that initiated the process, is the process' parent process and the process itself is the child process of the session. To see a listing of all exported variables, the command `export -p` can be used. Listing 2.7 shows the usage of exported variables.

Listing 2.7: Exported Environmental Variables

```

1 TEST1="ABCD"
2 TEST2="1234"
3 #Export TEST2 to be available to children processes
4 export TEST2
5 #Show exported variables
6 export -p
7 ...
8 declare -x TEST2="1234"
9 ...
10 #Use the variables and show their values
11 echo "TEST1='$TEST1' TEST2='$TEST2' "
12 TEST1='ABCD' TEST2='1234'
13 #Open a new bash session (child process)
14 bash
15 #Use the variables and show their values
16 echo "TEST1='$TEST1' TEST2='$TEST2' "
17 TEST1='' TEST2='1234'
18 #TEST1 has no value in this child process

```

2.4.1.1 Referencing Variables

As mentioned, a variable is used to store a value. The variable can be NULL (have no value), text, a number or the output of a command. This section shows how a variable is declared and used in a number of ways.

Listing 2.8: Referencing Variables

```

1 #Declare some variables
2 TEXT="My lucky number"
3 NUMBER=7
4 #Set the DATE variable = the result of a command
5 DATE=$(date +%A)
6 #Display the variables on screen
7 echo "$TEXT for this beautiful $DATE is $NUMBER"

```

```

8 My lucky number for this beautiful Thursday is 7
9 #Display the variables on screen using ${...} to reference a variable
10 echo "${TEXT} for this beautiful ${DATE} is ${NUMBER}"
11 My lucky number for this beautiful Thursday is 7
12 #The results for the previous 2 commands are the same but:
13 #It is necessary to use ${} instead of $, if the text is concatenated
14 echo "$TEXT_for_${DATE}_is_${NUMBER}"
15 7
16 echo "${TEXT}_for_${DATE}_is_${NUMBER}"
17 My lucky number_for_Thursday_is_7

```

2.4.2 Auto complete

A fast method of executing commands in the shell is to make use of the built in auto completion of the shell. To use the auto completion, start typing the command and press the *TAB* button. If the command (or part of the command) is unique, the full command will be auto completed. If the command is not unique, press *TAB* again. A list of commands that starts with the part that was typed will be listed on the screen. Listing 2.9 displays the use of auto complete to complete a command.

Listing 2.9: Auto Complete

```

1 c<TAB><TAB>
2 Display all 110 possibilities? (y or n)
3 c++                chainsaw                cmake                cpu
4 c2ph               changeparam            cmp                  crash
5 ...
6
7 cl<TAB><TAB>
8 clean-binary-files clear                cluster
9 cleanlinks         cls
10
11 cle<TAB>
12 clean-binary-files cleanlinks          clear
13
14 clear<TAB>
15 clear

```

On Line 1, *c* is pressed followed by *TAB*. Nothing is displayed so *TAB* is pressed again. The user is warned that there is 110 different possibilities. After pressing *y* all the possible commands are shown.

On Line 7, the user adds an *l* and *TAB* is pressed again, nothing is displayed so *TAB* is pressed again. This time only five possibilities are shown.

On Line 11, the user adds an *e* and *TAB* is pressed again. This time an *a* is automatically added to the command because the remaining available possibilities all start with *clea*.

On Line 14, a *r* is added and *TAB* is pressed. This time the command is complete and a space is added at the end of the command. This is an indication that the full

command is typed.

Auto completion is not exclusive to commands. A user can also use the same method to automatically complete a path, a file name, an alias or even a variable name.

2.4.3 Shortcuts

The Linux shell provides some useful shortcuts. Some users prefer not to make use of shortcuts thinking that it will take too long to learn or that it is not worth learning the shortcuts in the first place. This section describes a few very useful shortcuts that will save a user a lot of effort and just enhance the overall use of the system. Section 2.4.2 described the use of auto completion and the advantage of using auto completion is apparent.

While using the console, the output on the screen may become cluttered or distracting. An easy method of clearing the console screen is to use the *Ctrl+l* (Control and 'el') key combination. A user may press the *Ctrl+l* combination even while typing a command to have the screen cleared. If a user presses the key combination while typing a command, the screen will be cleared and the input of the command will continue on the first line of the console where it was before the key combination was pressed.

If a user starts typing a command and wants to cancel the command without executing the command and without deleting the whole command, the user can press the *Ctrl+c*. The shell will just ignore whatever was typed and go to the next input line.

An other useful shortcut key combination is *Ctrl+d*. This shortcut will attempt to exit the current application or session. This is useful because some applications require a user to type in "exit", some applications require a user to press "q" and other require a user to type "quit" to exit the application. This functionality can be tested by typing in the **cat** command without any options. If the user types something and press enter, the user input is just returned to the screen and the user is prompted for the next command. The best way to exit the "program" is to press *Ctrl+d*.

The shell holds a list of executed commands in a history file. The history file is saved to a hidden file (*.bash_history*) in the user's home directory. The history file works more or less like the auto completion feature described in the previous section by completing the command as the user is typing in the command. The difference is that the history types in the full command and all the options

that the user used with the command where as the auto complete feature only auto completes the active part of the command or path. To use the the history, the user can press *Ctrl+r*. The user will then enter a prompt that asks the user to type in the command. As the user types in the command, the last command that the user typed which reflects the command that the user is typing, is displayed. The user may continue to type in the command if the command displayed is not the one the user wants to use. When the required command is displayed on the screen, the user can press *enter* to use the command or press *Ctrl+c* to cancel the command. To view a list of the commands held in the history, the user can type the **history** command. A list of previously used commands are displayed.

All the commands saved in the history file can be accessed by browsing through the history list using the up and down arrows. When the required command is displayed on the screen, the user can simply press enter to re-execute the command. Optionally the user can also press the left or right arrows to edit the command first.

An other way to re-execute a command previously used is by prepending it with an exclamation mark (!). When using this feature, the user won't be able to search through the list of commands in the history. The last used command that starts with the part of the command typed, will be executed. For instance if the user typed the **wget** command earlier and types **!wg** and presses enter, the last command that started with "wg" (**wget http://.....**) will be executed.

A user may also want to scroll through previous output screens. To scroll up to previous screens *Shit+Page Up* can be used. To scroll down, *Shift+Page Down* can be used. As soon as the user starts typing again, the screen will automatically jump back to the last output screen.

Sometimes a user starts an application and needs to send the application to the background. The best method to start an application in the background is to add an ampersand (&) at the end of the line. However, if the user already started the application and want to send it to the background, the user can make use of the *Ctrl+z* key combination. When *Ctrl+z* is pressed, an identifier in square brackets are displayed followed by a plus sign (to show the last active process), the word "Stopped" and the command. To continue the process in the background, the user can type in the background command: **bg**. When the user wants to return the process to the foreground, the user can use the foreground command: **fg**.

2.4.4 Alias

Linux has a method to predefine options or a set of commands that are regularly used called an **alias**. An alias creates a command that can be used to execute a

set of instructions or commands with predefined options. For instance, on most Red Hat Linux systems a predefined alias *ll* exists. The alias *ll* calls the command `/bin/ls` with “`-lh --color=tty`” as the parameters. This allows a user to just type *ll* instead of typing in `/bin/ls -lh --color=tty`.

Aliases can also be used to override the default behaviour of a command. For instance Listing 2.10 defines an alias called *grep*. If the alias exists, that alias will be called each time the user types in *grep*.

At the end of Section 2.4.1 on page 21, it is mentioned that two configuration files can be used to set environmental variables. These files can also be used to store aliases in. Listing 2.10 shows how an alias can be created, viewed, used and finally destroyed.

Listing 2.10: Using an alias

```

1 #Define the alias 'grep':
2 alias grep='/bin/grep --color'
3
4 #Show a list of all defined aliases:
5 alias
6 ...
7 alias cls='/usr/bin/clear;pwd'
8 alias cp='/bin/cp -ivr'
9 alias grep='/bin/grep --color'
10 alias l='ls -al'
11 alias ll='/bin/ls -lh --color=tty'
12 alias ls='ls --color=tty'
13 alias nodes='/usr/bin/pbsnodes |/bin/grep -iv status|/bin/grep node0 -
    A 1'
14 ...
15
16 #Which 'grep' command will be used?
17 which grep
18 alias grep='/bin/grep --color'
19     /bin/grep
20
21 #Use the 'grep' alias
22 grep "testuser" /etc/passwd
23 testuser:x:27313:10000::/home/testuser:/bin/bash
24
25 unalias grep
26 #Which 'grep' command will be used?
27 which grep
28 /bin/grep
29
30 grep "testuser" /etc/passwd
31 testuser:x:27313:10000::/home/testuser:/bin/bash

```

Listing 2.10 on page 27, shows the following:

- On Line 2, a new alias for **grep** is created.
- On Line 5, the command **alias** will display a list of defined aliases.
- On Line 7, the alias **cls** is defined. The alias will execute two commands (**clear** and **pwd**).
- On Line 9, the 'newly' created alias **grep** is shown.
- On Line 17, the command **which** is called to display which **grep** command will be executed.
- On Line 22, the **grep** alias is used with additional parameters.
- On Line 23, returns the result of the **grep** command with the searched string printed in red because the **grep** alias used the “*--color*“ parameters.
- On Line 25, the **grep** alias is destroyed.
- On Line 27, the **which** command is called again, this time no alias is displayed, only the command itself.
- On Line 30, the **grep** command is called again but this time **grep** does not have predefined parameters that the alias provided.
- On Line 31, the same results are returned as on Line 23 but without colour formatting.

2.5 Getting Help

In Linux to see more information regarding the usage of a certain command, one can make use of the **man** command. The **man** command opens the manual pages of the specific command, if the manual pages are available. For instance to get more help on how to use the **ls** command, the user can type **man ls**.

Another method of getting help is to type the command and add the *--help* parameter to the command. For example to get a brief listing of the options that the command **ls** uses, the user can execute **ls --help**. Listing 2.11 shows the *--help* option for the **ls** command.

Listing 2.11: Brief command line help

```

1 ls --help
2 Usage: ls [OPTION]... [FILE]...
3 List information about the FILES (the current directory by default).
4 Sort entries alphabetically if none of -cftuvSUX nor --sort.
5
6 Mandatory arguments to long options are mandatory for short options
   too.
7  -a, --all                do not ignore entries starting with .
8  -A, --almost-all        do not list implied . and ..
9      --author              with -l, print the author of each file
10  -b, --escape             print octal escapes for nongraphic
   characters
11 ...

```


Linux Commands

Contents

3.1	Commands Overview	30
3.2	Introduction	30
3.2.1	Command return codes	31
3.2.2	Logical AND/OR testing	32
3.3	Using output as input	33
3.4	Command Examples	34
3.4.1	ssh	34
3.4.2	scp	35
3.4.3	rsync	35
3.4.4	cd	36
3.4.5	pwd	36
3.4.6	ls	37
3.4.7	mkdir	37
3.4.8	cp	38
3.4.9	mv	38
3.4.10	rm	38
3.4.11	cat	39
3.4.12	sort	40
3.4.13	echo	41
3.4.14	more	41
3.4.15	less	42
3.4.16	head	42
3.4.17	tail	42
3.4.18	cut	43
3.4.19	find	43
3.4.20	grep	43
3.4.21	screen	44
3.4.22	info	45
3.5	Mathematical Arithmetic	45

3.1 Commands Overview

This chapter describes some of the commands that are used on a regular basis. In Chapter 2, the use of the console was described with examples of some commands such as *alias* and *ls*. This chapter will describe some of the commands and how they are used. Some commands such as *rsync*, and *screen* may need to be installed by the systems administrator to be able to use them. The other commands should be installed on most installations.

3.2 Introduction

A command can be executed by typing the command followed by options and parameters and pressing enter.

See Listing 3.1

Listing 3.1: Basic command execution

```
1 ls -l /var/log
```

In Listing 3.1 *ls* is the command. “-l” is the options and */var/log* is a parameter. It is possible to switch parameters and options for some commands but it should be attempted to keep to a specific convention.

Most commands can have the options combined in one string such as in Listing 3.2.

Listing 3.2: Using multiple options for a command

```
1 ls -l -a -r -n -t /var/  
2 # ...is the same as  
3 ls -larnt /var/  
4 # One can also switch the order of most options, if permitted.  
5 ls -altrn /var/
```

Also note that the options are Case Sensitive. The shell interpreter does not mind the number of spaces between options but a space should be used to separate options and parameters.

Listing 3.3: Using spaces in the shell

```
1 ls          -al          /var  
2 # ...is the same as...  
3 ls -al /var
```

To perform a number of commands in one line, a semi-colon “;” can be used to separate the commands.

Listing 3.4: Multiple commands on one line

```
1 clear; cd /var/log; pwd; cd ~; pwd
2
3 /var/log
4 /home/testuser
```

In Listing 3.4 Line 1, the command will clear the screen.

Change the directory to /var/log.

Display the directory has changed to /var/log.

Change to the user's home directory and

lastly display that the current working directory is in fact the user's home directory.

This is useful especially in cases where a long period of time will elapse before the next command will be executed. For instance when compiling a kernel, Listing 3.5 shows some steps taken when compiling a kernel.

Listing 3.5: Commands used to compile a kernel (taking a long time)

```
1 make
2 make modules
3 make modules_install
4 make install
```

If the commands in Listing 3.5 is executed, 2 to 30 minutes may pass between each step. If the user does not want to wait for the whole duration to execute the next command, the commands may composed into a single line as displayed in Listing 3.6.

Listing 3.6: Commands used to compile a kernel (on one line)

```
1 make;make modules;make modules_install;make install
```

3.2.1 Command return codes

Every command that is executed in a shell, should return an error/exit code. By default an exit code of zero '0' will be returned if a command completed successfully and another value will be returned when a command failed. To see the code returned, one can look at the value of the variable "\$?".

Listing 3.7: Displaying command return codes

```
1 tail -n 1 /etc/passwd
2 vanecka:x:500:500::/home/vanecka:/bin/bash
3 echo $?
4 0
5 tail -n 1 /var/log/messages
6 tail: cannot open '/var/log/messages' for reading: Permission denied
7 echo $?
8 1
```

3.2.2 Logical AND/OR testing

Section 3.2.1 mentions that each command executed, returns a “return code”. It can be helpful to base the execution of the next command upon the success of the previous command. For instance, if the command was successful, execute a specific command. An other approach could be to execute a command if the previous command was unsuccessful. To execute a command if the previous command was successful, the AND operator (&&) can be used. However, if the command failed or was not successful, the OR operator (||) can be used.

Listing 3.8: Logical OR example

```
1 tail -n 1 /var/log/messages || echo "Could not be executed"
2 # The above statement reads: get the last line of /var/log/messages OR
   write error to the screen
```

Listing 3.9: Logical AND example

```
1 cd /tmp && echo "Command successful"
2 # The above command will change to the /tmp directory and if it was
   successful, write a comment to the screen.
```

To take full advantage of this logical testing, Listing 3.6 on page 31 should be rewritten to only execute the next command if the previous command was successful. Listing 3.10 shows the better alternative to compile the kernel.

Listing 3.10: Compiling the kernel using logical tests

```
1 # Rewriting the following, testing for error codes: make;make modules
   ;make modules_install;make install
2 make && make modules && make modules_install && make install
3 # The Above command will execute each command after the previous
   command completed successfully.
```

3.2.2.1 Command Syntax

When looking at the help files of a certain command, you will come across the following usage statement:

```
command_statement [option0] {option1|option2} ...
```

In this example above the following are constrains upon the command:

- * Anything typed without brackets are mandatory when calling the command. (command_statement)
- * Text within square brackets are optional. ([option0])
- * Options between curly “{}” brackets separated by a pipe “|” means that the user has to choose one or more of the options. ({option1|option2})
- * Text between angle brackets “< >” indicates that you must replace the text with

the needed value or name.

* The ellipses “...” means “and so forth”

Also note that if multiple options exist, the user are not limited to just one option but he/she can combine them together.

Listing 3.11: Command syntax

```
1 ls -a    #Shows a listing of all files and directories, including
             hidden ones
2 ls -l    #Shows a listing in a long format - showing more information
             about files/directories.
3 ls -h    #Shows file size in human readable form, but only works
             together with the -l option
4 ls -alh  #Shows all the above listing information.
```

3.3 Using output as input

The shell has a very powerful feature that allows a user to use one command's output as the input of the next command. To use the output of the previous command, the two (or more) commands are split using the pipe character (|).

For instance Listing 3.12 shows how a user can perform a listing of a directory using **ls**, and search for a specific string ("host") using the **grep**.

Listing 3.12: Using one command's output as input

```
1 ls /etc/ |grep host
2 ghostscript
3 host.conf
4 hosts
5 hosts.allow
6 hosts.deny
```

3.4 Command Examples

This section shows a few commands used on a regular basis. More options are available than listed here.

For more options on a command execute *man* followed by the command.

Eg *man ls*

3.4.1 ssh

The *ssh* command is used to connect to a remote host over a Secure Shell. The communication between the two nodes are encrypted with SSL certificates. This means that a third party can't intercept communication to retrieve passwords or data. The *ssh* command can also be used to execute commands on a remote host. Chapter 1 Section 1.3.1 on page 11 also shows some examples.

When using SSH, the user will be prompted to provide a password for the remote machine. This is the standard operation when connecting to a remote machine. It is also possible to create SSH keys that can be used to authenticate a user, without having the user type in a password. This method is very secure since anyone that wants to use this method, has to first have the private key of the user to be able to connect to the remote server.

Listing 3.13: Example: ssh

```
1 # Connect to a remote host as the same user that is currently logged
   in
2 ssh remotehost.name.domain
3 # Connect to a remote host as a the root user
4 ssh root@remotehost
5 # Execute a command (df -h) on a remote host, as the root user
6 ssh -n root@remotehost "df -h"
7 # Pipe the content of a file to an SSH command
8 cat /etc/hostname | ssh remotehost " cat - >> /tmp/remote_hostname"
```

Most of the above commands are straight forward and self explanatory when looking at the preceding comments. However line 8 is a bit more complex. In this line, the content of `/etc/hostname` is captured by the `cat` command. The content is then piped to the `ssh` command. The `ssh` command itself, captures the content that was sent by the first `cat` command and redirect that content to the `cat -` command on the remote host. That "input" is then redirected (`>>`) to a file called `/tmp/remote_hostname` on the remote host. A much simpler method to perform this would have been to use the `scp` command as explained Section 3.4.2, to *secure copy* the file from the local host to a different name on the destination host. However the above method was used to show how standard output from one system can be sent to standard input on a remote machine and eventually redirected to a file on the remote machine. The `cat` command is explained in Section 3.4.11 with an example (line 4 in listing 3.24) of how it is used in a similar way as shown

in listing 3.13 line 8.

3.4.2 scp

The Secure Copy command (**scp**) is used to copy files from one machine to another. The data transfer is also SSL encrypted. The Secure Copy command can be used to copy files and directories. Chapter 1 Section 1.3.2 on page 12 also shows some examples.

Listing 3.14: Example: scp

```
1 # Copy LocalFile.txt to a remote host
2 scp LocalFile.txt /home/current_user/
3 # Copy /home/current_user/LocalFile.txt to a remote host, to the home
  directory of root, as root
4 scp /home/current_user/LocalFile.txt root@remote_host:/root
5 # Copy /home/current_user/RemoteFile.txt from a remote host to
  LocalFile.txt in the current working directory
6 scp remote_host:/home/current_user/RemoteFile.txt LocalFile.txt
7 # Copy the whole home directory of root (-r = recursively), from a
  remote host to a backup folder
8 scp -r root@remote_host:/root /backup
```

3.4.3 rsync

The **rsync** command is used for a similar purpose and in a similar way as the **scp** command. The **rsync** command also utilises the SSH protocol to transfer data from one machine to and from another. The major difference between **rsync** and **scp**, is that **rsync** can determine which files should be copied over and which files can be skipped. The **rsync** command uses a few constraints to determine whether the source and destination files are the same or whether the files should override each other. The **rsync** command can also copy important file permissions such as SELinux context parameters, whereas **scp** can not transfer these special permissions.

Listing 3.15: Example: rsync

```
1 #Copy the whole /etc directory from the remote host to the /backup/etc
  directory:
2 rsync root@remote_host:/etc /backup
3 #Copy the subdirectories and files inside /etc from the remote host to
  the /backup directory:
4 #Note, the only difference from the previous command is the trailing
  / after the /etc
5 rsync root@remote_host:/etc/ /backup
6 #Only show which files would be copied without actually copying the
  files
7 rsync -dry-run root@remote_host:/etc/ /backup
8 #It is also useful to see the progress as the files are transferred:
9 rsync -progress root@remote_host:/etc /backup
```

```

10 #If you have possibly modified some files and don't want to override
    it with older versions, use the -u option:
11 rsync -progress -auv root@remote_host:/work /working_files
12 #Finally, lets copy all the files and directories, keeping SELinux
    permissions
13 rsync -progress -av -HAX root@remote_host:/etc /backup

```

In Listing 3.15 line 2, the remote `/etc` directory and its subdirectories are copied to `/backup/etc`.

On line 5, the files and directories inside the remote `/etc` are copied to `/backup` but they are not contained in the `/backup/etc` directory, instead they are copied loosely into the `/backup` directory.

On line 7, no files are copied, instead only a list of the files that would have been copied is displayed.

Line 11 shows how to copy files but ignore files where the destination file has been modified/updated (`-u`) and is newer than the source files.

Line 13 have a few more options to preserve the SELinux context etc. The `-H` option preserves **H**ardlinks. The `-A` option preserves the **A**CLs and permissions. The `-X` option preserves the e**X**tended attributes such as the SELinux context.

3.4.4 cd

The `cd` command is used to Change Directory to another location.

Listing 3.16: Example: cd

```

1 #Change the working directory to /etc/init.d
2 cd /etc/init.d
3 # To change to the last working directory:
4 cd - # (That's cd minus)
5 # To change to your home directory:
6 cd
7 # ...or...
8 cd ~ # (That's cd Tilde)
9 # ...or...
10 cd $HOME
11 # To go one directory up
12 cd ..
13 # To go three directories up
14 cd ../../..

```

3.4.5 pwd

The `pwd` command Prints the current Working Directory. No useful options exists for this command.

Listing 3.17: Example: pwd


```

1 #Change the working directory to /etc/init.d
2 cd /etc/init.d
3 pwd
4 /etc/init.d
5 cd ..
6 pwd
7 /etc
8 cd $HOME
9 pwd
10 /home/testuser

```

3.4.6 ls

The **ls** command is used to list the contents of a directory. A wide range of options exist for this command but some of the more useful ones are listed in Listing 3.18.

Listing 3.18: Example: ls

```

1 # Show a full listing of the /home/user directory:
2 ls -l /home/testuser
3 drwxr-x---+ 2 testuser hpcuser          3 Jul  7 19:56 LocalDirectory
4 -rw-rw-r---+ 1 testuser hpcuser  2048000 Jul  7 20:00 LocalFile.dat
5 ...
6 # Show a long listing in human readable file size format:
7 ls -lh
8 drwxr-x---+ 2 testuser hpcuser    3 Jul  7 19:56 LocalDirectory
9 -rw-rw-r---+ 1 testuser hpcuser 2.0M Jul  7 20:00 LocalFile.dat
10 # Show all files, including hidden ones:
11 ls -a
12 .                .bash_profile  LocalFile.dat      RemoteFile.dat
13 ..               .bashrc       LocalFileLink.dat  .ssh
14 ...
15 # Show results in a reversed order:
16 ls -r
17 RemoteFile.dat  RemoteDirectory  LocalFileLink.dat  LocalFile.dat
18 # Show sub folders Recursively:
19 ls -R
20 ./LocalDirectory:
21 LocalFile.dat

```

3.4.7 mkdir

The **mkdir** command is used to Make a new Directory.

Listing 3.19: Example: mkdir

```

1 # Create a new directory named DataFiles in /home/testuser/
2 mkdir /home/testuser/DataFiles
3 # Create a new sub directory with its parent directories as needed (
  create parents if it doesn't exist):

```

```
4 mkdir -p /home/testuser/new_parent/new_subfolder/new_directory
```

3.4.8 cp

The **cp** command is used to copy files from one location to another on the same machine. If a file already exists the system may ask the user if the file should be overwritten. This is not the default behaviour on most systems, but it may be configured to automatically override files by prompting the user with the **-i** flag.

Listing 3.20: Example: cp

```
1 # Copy /etc/passwd to file1 - override file1 with the contents of /etc
  /passwd:
2 cp /etc/passwd file1
3 # Copy file1 to file2 preserving the file permissions
4 cp -p file1 file2
5 # Copy /var/log directory to the current directory recursively,
  including sub-directories and files (Note the full stop)
6 cp -r /var/log .
7 # Copy files recursively and show what is happening verbosely:
8 cp -rv /var/log .
```

3.4.9 mv

The **mv** command is used to move (cut and paste) a file/folder from one location to another. It can also be used to rename a file/folder.

Listing 3.21: Example: mv

```
1 # Move a folder from a mounted volume to a user's home directory:
2 mv /mnt/usb/folder ~/
3 # Move only the contents of a folder to the current location:
4 mv /mnt/usb/folder/*.
5 # Move files, while showing verbosely what files are moved to the
  screen:
6 mv -v /mnt/usb/folder .
7 # Rename a file or directory to another name
8 mv old_directory new_name
```

3.4.10 rm

The **rm** command is used to remove files and can also remove directories. Use caution when removing files from the system. A system may be configured using the **-i** option to interactively ask if a file should be removed.

Listing 3.22: Example: rm

```
1 # Remove a file called unwanted from the current location:
```

```
2 rm unwanted
3 # Remove files starting with old, forcing removal without prompting
  for confirmation:
4 rm -f old*
5 # Remove a folder recursively and forcing no confirmation:
6 rm -rf foldername
```

3.4.11 cat

The Concatenate command is used to display the contents of a file to the screen. When the **cat** command is executed against a file, the whole file's content will scroll over the screen. The full advantages of the **cat** command can only be gained by using it together with some other commands. One can for instance append the contents (or part thereof) of one file to the end of another file by redirecting the output to another file. Some of the examples in Listing 3.23 may look strange but going through this text, these commands will become more easier to interpret and can save a lot of time.

Listing 3.23: Example: cat

```
1 # Show the contents of /etc/passwd:
2 cat /etc/passwd
3 # Copy the contents of /etc/passwd to a new file called MyUsers:
4 cat /etc/passwd > MyUsers
5 # Append the contents of /etc/group to the end of file MyUsers:
6 cat /etc/group >> MyUsers
7 # Display the contents of the /etc/passwd file, but only show the
  lines containing the word "bash"
8 cat /etc/passwd | grep bash
```

The above listing shows some of the basic uses of the **cat** command and how to redirect the output to other commands such as **grep**. This is however only part of the use for the **cat** command. The **cat** command can also be used to create what we refer to as a *Here Document*. A here document is a file that is generated by redirecting either user input or predefined content into a file. The following listing shows how the **cat** command can be used to redirect content to new files:

Listing 3.24: Example: using cat to create files

```
1 #Read input from the standard input (aka the keyboard) and save to a
  file:
2 echo "Type a list of your friends' names."
3 echo "Press Cntrl+c when done typing, or type END"
4 cat - >> friends.txt <<END
5 chandler
6 joey
7 rachel
8 monica
9 phoebe
```

```
10 ross
11 END
12 #Create a file called other.txt with a few names in it;
13 cat > other.txt <<EOF
14 jake
15 pete
16 mike
17 andrew
18 John
19 EOF
20 #Add two more names to the file, without overriding the existing
    content:
21 cat >> other.txt <<EOF
22 ellen
23 louis
24 EOF
```

In listing 3.24 on line 4, the **cat** command is called with a minus/dash (-) that tells **cat** to read user input from the keyboard. The input is then redirected (>>) to a file called friends.txt up to the point when the user types the word **END** on its own line or until the user presses control+c. Lines 5 to 11, was user input in this case.

In lines 13 through 19, a "here document" is created using the cat command. If the file did not exist, it would have been created. If the file existed, the content would have been overridden with the new content.

3.4.12 sort

The Sort command is used to sort output. The results of the **sort** command can be saved directly into a new file, or it can sort the output from another command and redirect the output to a file. The result of sort will be displayed to the standard output (screen).

Listing 3.25: Example: sort

```
1 # Display the contents of Test1.txt and sort the output
2 cat Test1.txt | sort
3 # Display the contents of Test1.txt and sort the output in reverse
    order
4 cat Test1.txt | sort -r
5 # Display the contents of Test1.txt, sort the output and write the
    results in Sorted.txt
6 cat Test1.txt | sort > Sorted.txt
7 # Sort the file Test2.txt and write the results to Results.txt
8 sort Test2.txt > Results.txt
9 # Sort the file Test2.txt and write the results to Results.txt
10 sort Test2.txt -o Results.txt
11 # Display the contents of Names.txt and display the unique names in
    the file
```

```
12 sort Names.txt -u
```

3.4.13 echo

The `echo` command is normally used to display text to the screen. Sometimes users will also use the `echo` command and redirect the output to a file. There are a few options or parameters that can be passed to the `echo` command but the more used ones are the `-n` and `-e` options. The `-n` option is used to not print the trailing newline character. This is useful when for instance a user is prompted to input a value in a script and have the user type the value directly next to the prompt instead of having the user type the value in the next line. It is also useful if you want to print some text using `echo` and then print more text using a new `echo` command that is printed next to the previous output. Another useful option is the `-e` option. This will tell the interpreter to interpret for instance `"\n"` as a new line character or `"\t"` as tab instead of just printing "backslash n or t".

Listing 3.26: Example: echo

```
1 echo "Hello world"
2 # Prints Hello world to the screen.
3 echo "Hello there. I am $USER."
4 # A system environmental variable ($USER) is included in the statement
5 :
6 # This variable name will be replaced in the output with its value.
7 # Thus what you will see on the screen is the following :
8 [john_doe@localhost ~]$ echo "Hello there. I am $USER."
9 Hello there. I am john_doe.
10
11 #Add a line "Mike was here" at the end of a file:
12 echo "Mike was here" >> /home/mike/myfile.log
```

3.4.14 more

The `more` command is used to browse through the contents of a file, from top to bottom. This scrolling is only done in one direction from the top of the file and the user can not return to the previous screen. This command is useful if one wants to view the content of a large text file and read through it. To move around with the `more` command, one can press "return" on the keyboard to move down one line. To move down a full page, one can press the "space bar". To exit this view press "q/Q". To start an editor at the current line, press "v". An even more useful use for this command is to scroll the output of the screen. If for instance a listing of the contents of `/etc` is displayed, one will find that the output scrolls over the screen too fast to read. By redirecting the output of the `ls` command to the `more` command (by using pipe), one will be able to go through the listing more conveniently.

Listing 3.27: Example: more

```
1 ls /etc | more
```

3.4.15 less

The **less** command has exactly the same use as the **more** command but more functionality. The **less** command also displays the content of a file, but one can browse up and down in the file. The **less** command is also based on the user's editor's usage. A user's default editor may be set to **vi**. Thus, some of the features of **vi** also exists in **less**. Unlike **more** and **vi**, **less** does not open the whole file at once. Allowing the user to open a very big file with the least amount of overhead. It is a good idea to use **less** to open a big log file or a big output file rather than using **vi** or **more**. **Less**, like **more**, can also be used to scroll the output of the screen with ease. For instance, one can do the same directory listing as in Listing 3.27, but one will be able to scroll up and down by using **less**.

Listing 3.28: Example: less

```
1 # This example displays the contents of the /etc directory and allows
   browsing through the output in both directions.
2 ls /etc | less
```

3.4.16 head

The **head** command is used to view the first few lines of a file. A specific number of lines from the top of a file can be specified.

Listing 3.29: Example: head

```
1 # Show the first ten lines of the /etc/passwd file
2 head /etc/passwd
3 # Show the first thirty lines of the /etc/passwd file.
4 head -n 30 /etc/passwd
```

3.4.17 tail

The **tail** command is the opposite of the **head** command but it has one additional feature. The **tail** command will display the last few lines of a file to the screen. One added feature that the **head** command lacks is the option to follow the content of a file. This means that while a file grows, the output is scrolled on the screen. So one will be able to see the last few lines of for instance a log file as events are logged to the file. This is very useful to monitor an output file on screen, while it is being created.

Listing 3.30: Example: tail

```

1 # Display the last twenty lines of the /etc/passwd file.
2 tail -n 20 /etc/passwd
3 # Display the last ten lines of the output.file file and displays the
   lines as they are being added to the file.
4 tail -f output.file
5 # Show the amount of free space for the mounted file systems, skipping
   the first header lines.
6 df -h | tail -n +2

```

3.4.18 cut

The **cut** command is used to extract certain parts of a text field. One could use cut to split the output on a fixed length or on a specific separator.

Listing 3.31: Example: cut

```

1 # Show characters 1 to 10 of each line from /etc/passwd
2 cut -c1-10 /etc/passwd
3 # Show only column 1 and 6 of /etc/passwd, using ":" as delimiter
4 cut -d: -f1,6 /etc/passwd
5 # Show who is logged in to the system, show characters 1-8 and 18 to
   the end
6 who | cut -c1-8,18-

```

3.4.19 find

The **find** command is used to search for files and folders. A user can search for files by Eg. File name, file size, owner or permissions.

Listing 3.32: Example: find

```

1 # Search for a file named myfile.txt from the root file system
   throughout the whole system
2 find / -name myfile.txt -type f
3 # Finds all directories starting with the name "test" in the current
   directory
4 find . -name test* -type d
5 # Find all files in /tmp that are less than one megabyte big
6 find /tmp -size -1M -type f
7 # Find all the files larger than one gigabyte and delete it
8 find /tmp -size +1G -type f -exec rm -f '{}' \;
9 # Find all files/directories that belongs to the user john.
10 find / -user john

```

3.4.20 grep

The **grep** command is used to search for a certain string inside another string. The **grep** command can be used to find a part of text inside files too. This command

makes use of special characters (Page 18 Section 2.1.2) as well as regular expressions (described in Chapter 5).

Listing 3.33: Example: grep

```
1 # Look through all the files in /home/sarah and print this phrase and
   the filenames of the files containing this exact text phrase.
2 grep "The man with the red hat" /home/sarah/*
3 # The following command will display all the lines of the file called
   logfile.log that contains the word error.
4 # The -i option indicates that the search is performed case-
   insensitive.
5 # This means that results that could be found, may contain something
   like ERROR, errors, Error and Terrorist.
6 grep -i "error" logfile.log
7 # Perform a listing of /var/log, show all results that contain a 's'
   and highlight the result where the 's' was seen.
8 ls /var/log |grep --color s
9 # Perform a listing of /var/log and show all the results that DOES NOT
   contain a 's'
10 ls /var/log |grep -v s
11 # Display the contents of /etc/passwd, look for lines that begin with
   "r"
12 cat /etc/passwd |grep "^r"
13 # Display the contents of /etc/passwd, look for lines that end with "
   nologin"
14 cat /etc/passwd |grep "nologin$"
```

3.4.21 screen

Some users may execute commands or applications that runs for a very long time. If the user starts the application and closes the terminal before the application is done, the application will be killed and execution will stop. One way to overcome this problem is to run the applications in a **screen** session. The **screen** command opens a special session that is not closed when the parent process is closed.

Listing 3.34: Example: screen

```
1 #See if there is a screen session already opened
2 screen -ls
3 No Sockets found in /var/run/screen/S-testuser.
4 screen
5 #The screen is cleared
6 #See if there is a screen session already opened
7 screen -ls
8 There is a screen on:
9      4135.pts-5.grid-ui      (Attached)
10 1 Socket in /var/run/screen/S-testuser.
11 #Now the user has an active session and start the application
12 test_application
13 001 - Thu Jul 14 14:15:18 SAST 2011 - Still running
```



```
14 002 - Thu Jul 14 14:15:19 SAST 2011 - Still running
15 003 - Thu Jul 14 14:15:20 SAST 2011 - Still running
16 #When the application starts, the user can close the session
17 #the application will continue in the background
18
19 #Now the user can re-connect to the screen session:
20 screen -ls
21 There is a screen on:
22          4135.pts-5.grid-ui          (Detached)
23 screen -x 4135
24 075 - Thu Jul 14 14:16:33 SAST 2011 - Still running
25 076 - Thu Jul 14 14:16:34 SAST 2011 - Still running
26 077 - Thu Jul 14 14:16:35 SAST 2011 - Still running
27 tail -n1 /tmp/${USER}_sleeptest
28 100 - Thu Jul 14 14:17:08 SAST 2011 - Done
```

3.4.22 info

More information regarding commands can be found using the *manual* pages or by typing in the *info* command. The *info* command can be used by typing in the command info followed by the command that you need more information on. The info pages also have sub sections. If a line starts with an asterisk (*), you can press enter on that line to follow the sub section. To return to the upper section, press *backspace*.

Listing 3.35: Example: info

```
1 info bash
```

3.5 Mathematical Arithmetic

There is a number of methods to perform some mathematical equations. Two more used ones are the *let* command and making use of double parentheses (()). The format for both these commands are similar. We are going to focus more on the double parentheses. The sequence in which precedence is given to the execution of a mathematical equation is determined by the order in the following table. The following table shows the operators in order of decreasing precedence in equations.

Mathematical Operands	
var++, var-- -	Post-increment and post-decrement. Interpret the value of integer variable and then add or subtract one
++var, --var	Pre-increment and pre-decrement. Add or subtract one and then interpret the value
+expr, -expr	Unary plus or unary minus. Unary plus returns the value as if it was multiplied by one. Unary minus returns the value as if it was multiplied by negative one
!	Logical negation, Logical negation returns true if the operand was false and returns false if the operand was true
**	Exponentiation
*, /, %	Multiplication, Division, Remainder (modulo)
+, -	Addition, Subtraction
<=, >=, <, >	Comparison: Less than or equal to, Greater than or equal to, Less than, Greater than
==, !=	Equality, Inequality
&&	Logical AND
	Logical OR
expr1 ? expr2 : expr3	If expr1 is true, return expr2. If expr1 is false, return expr3.
=, *=, /=, %=, +=, -=, <<=, »=, &=, ^=, =	Assignment. Assign the value of the expression that follows the operator, to the variable that precedes it. If an operator prefixes the equals sign, that operation is performed prior to assignment. For instance, let "var += 5" is equivalent to let "var = var + 5". The assignment operation itself evaluates to the value assigned

When the following commands are executed, the "math" part is not executed as expected:

Listing 3.36: Expecting Interpreter to return math equation's result

```
1 myvar="5 + 5"
2 echo $myvar
```

In listing 3.36, one may expect the interpreter to return 10 in the last echo command; instead the value "5 + 5" is returned to the screen. To get the correct/expected result, one can first **declare** the variable as an integer and then set its value, as indicated by listing 6.19.

Listing 3.37: Defining a variable's type as integer and executing an equation

```
1 declare -i myvar
2 myvar="5 + 5"
```

```
3 echo $myvar
```

Listing 3.37 will return the expected value (10) to the screen due to the fact that the variable's type was first set to integer. Another method to have the result of an arithmetic equation be assigned to a variable, is as follows:

Listing 3.38: Setting a variable's value equal to the result of an equation

```
1 #Note the space after and before the parentheses
2 #The space before and after = is optional inside the parentheses
3 (( myvar = 6 + 6 ))
4 echo $myvar
```

Listing 3.38 shows a method that uses double parentheses to perform an equation and save the value thereof to a variable. It is also possible to perform an equation, without saving the value to a variable. Listing 3.39 shows how an equation can be executed and the result be shown, without saving it to a variable:

Listing 3.39: Basic equation, only showing result without setting a variable's value

```
1 echo " 7 x 7 = $(( 7 * 7 )) is equal to 7 squared: $(( 7 *\ 2)) "
```

Listing 3.39, performs two equations and returns the results to the screen. In contrast, the following example (3.40) shows how two variables are set and the result of the whole equation is displayed to the screen.

Listing 3.40: Setting two variables and displaying the result of the equation

```
1 echo "$(( myvar = 7 + $(( vartwo = 4+4 )) ))"
2 #15 is displayed as a result
3 echo "myvar=$myvar and vartwo=$vartwo"
4 #Displays: "myvar=15 and vartwo=8" on the screen
```


Editors

Contents

4.1	Introduction	50
4.2	Graphical Editors	50
4.2.1	gedit	50
4.3	Shell Editors	51
4.3.1	nano	52
4.3.2	vi	53

4.1 Introduction

Most of the files on a Linux system is text files. A text file is a file that can be opened, read and modified by a user in a easy to understand way. For instance a XML file can be seen as a text file, for it is human readable without binary characters. Most HPC applications make use of a basic text file as a input file and perhaps a few database files that should not be modified with a normal text editor.

Because users normally access a Linux system from a remote host and especially using a shell to execute commands, this chapter will focus more on the use of text editors from the shell and not graphical editors. One graphical editor is shown in this chapter for users that struggle to get to grips with an advanced editor such as *vi*. There is no shame in preferring a certain editor above an other, it is up to the user to decide which editor he/she prefers.

That said, this chapter will focus more on the *vi* editor than on the others because *vi* has a lot of different “shortcuts” to perform a number of functions that may save the user a lot of time.

4.2 Graphical Editors

Linux has a number of graphical text editors that can be used to edit files. The author decided to include only one of them because the functionality of the chosen editor sufficed the requirements for the workshop. However, there is a lot of other and better editors available for free use.

To make use of the graphical editor from a Windows machine, the user first have to connect to the Linux machine with X Forwarding enabled. Chapter 1 described this method in Section 1.2.1.1 on page 4 and Section 1.2.2 on page 7. In short; Xming must be started

A PuTTY session with X11 Forwarding must be opened.

4.2.1 gedit

The graphical interface editor chosen for this document is called *gedit*. Figure 4.1 shows the PuTTY screen in the background with the command that was called (*gedit /etc/profile*) to open the *gedit* interface. The *gedit* interface is shown in front of the PuTTY screen.

It could be useful to open two *PuTTY* sessions, one to open the editor and another to execute other commands etc.

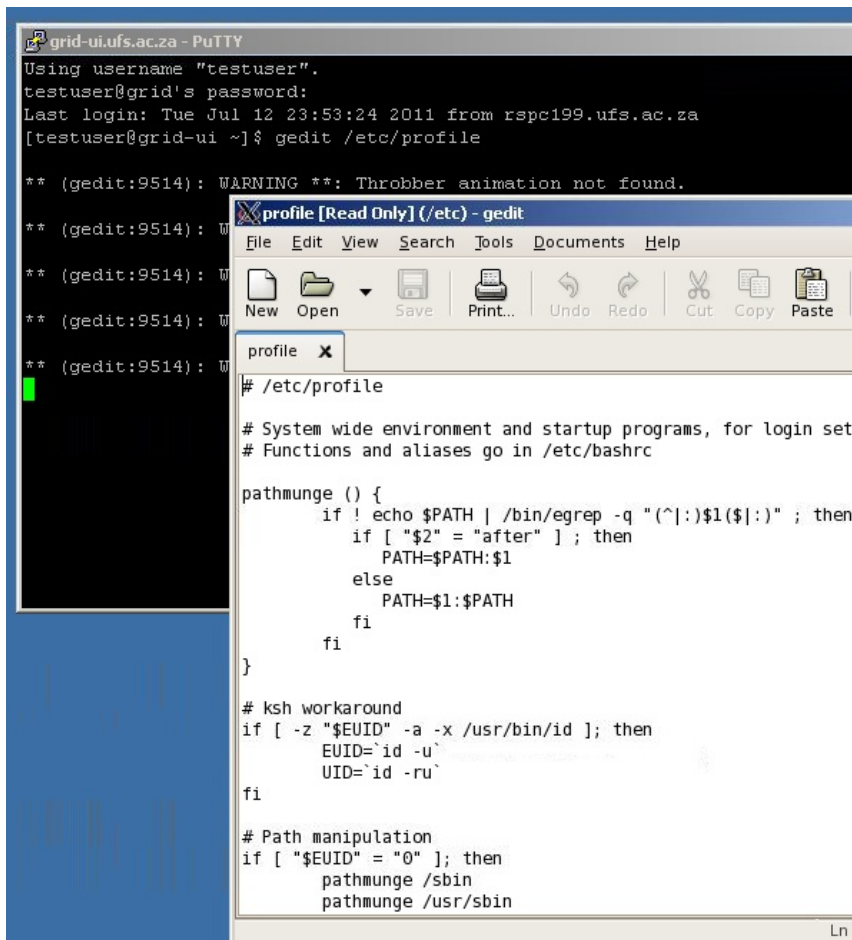


Figure 4.1: gedit - Graphical Text File Editor

4.3 Shell Editors

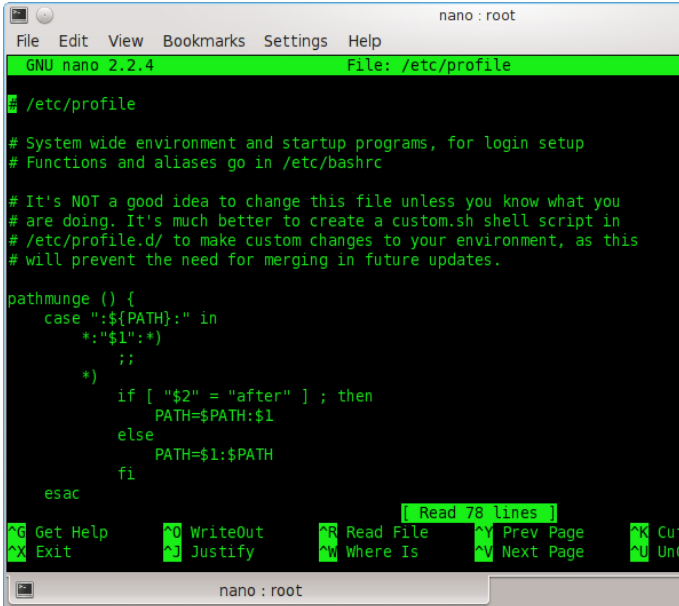
Linux has a vast number of editors that can be used from the shell. The most popular shell editors are: **vi** and **nano**. These two editors should be installed by default but it is possible that they are missing from the system. If the editor that a

user prefers to use is not installed on the system, the systems administrator should install it on the system.

4.3.1 nano

Nano is ANOther editor, an enhanced free Pico clone developed by Chris Allegretta et al.

Nano is a basic text editor that is used to modify human readable (text) files. Unlike **vi**, **nano** usually has to be manually installed onto the system. When opening a file with nano, a list of commands is provided at the bottom of the **nano** editor screen. For instance the command option to exit **nano** is *Ctrl+x*. Figure 4.2 shows the editor interface of **nano**.



```
nano : root
File Edit View Bookmarks Settings Help
GNU nano 2.2.4 File: /etc/profile

/etc/profile

# System wide environment and startup programs, for login setup
# Functions and aliases go in /etc/bashrc

# It's NOT a good idea to change this file unless you know what you
# are doing. It's much better to create a custom.sh shell script in
# /etc/profile.d/ to make custom changes to your environment, as this
# will prevent the need for merging in future updates.

pathmunge () {
  case "${PATH}:" in
    *:"$1":*)
      ;;
    *)
      if [ "$2" = "after" ] ; then
        PATH=$PATH:$1
      else
        PATH=$1:$PATH
      fi
  esac
}

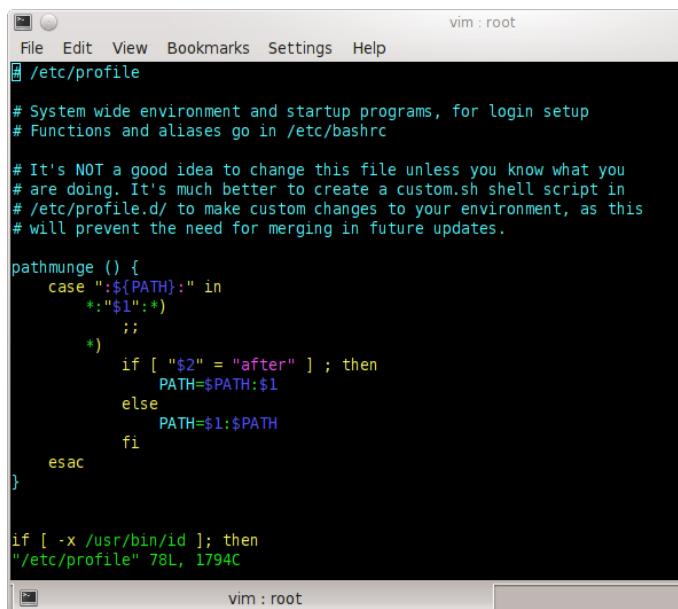
Read 78 lines
^G Get Help      ^O WriteOut     ^R Read File    ^Y Prev Page    ^K Cut
^X Exit          ^J Justify      ^W Where Is     ^V Next Page    ^U UnC
nano : root
```

Figure 4.2: nano - Text File Editor

4.3.2 vi

vi - a programmers text editor. The original code for **vi** was written by Bill Joy in 1976.

As described earlier in this chapter, **vi** is a powerful and advanced text editor. **vi** has six basic modes while using it, but this text focuses on only two. The first mode, and the mode that a file is opened in when opening **vi**, is the *normal mode*. The normal mode is used to execute **vi** specific commands. The second mode is the *insert mode*. The insert mode, as the name implies, is used to insert or edit the file. While a user is in the insert mode, the user can browse up or down using the arrow keys or the Page Up and Page Down options. Figure 4.3 shows the same file opened by the two other editors. The first thing that the user may notice is that **vi** has syntax highlighting enabled. Syntax highlighting is very useful while writing scripts. Other editors also support syntax highlighting if the file is detected as a specific type by the editor.

The image shows a terminal window with the vi text editor open. The title bar reads 'vim : root'. The menu bar includes 'File Edit View Bookmarks Settings Help'. The current file is '/etc/profile'. The content of the file is displayed with syntax highlighting: comments are in green, function names in blue, and code in white. The code includes a 'pathmunge' function and an 'if' statement at the bottom. The status bar at the bottom reads 'vim : root'.

```
vim : root
File Edit View Bookmarks Settings Help
# /etc/profile

# System wide environment and startup programs, for login setup
# Functions and aliases go in /etc/bashrc

# It's NOT a good idea to change this file unless you know what you
# are doing. It's much better to create a custom.sh shell script in
# /etc/profile.d/ to make custom changes to your environment, as this
# will prevent the need for merging in future updates.

pathmunge () {
    case "${PATH}:" in
        *:"$1":*)
            ;;
        *)
            if [ "$2" = "after" ] ; then
                PATH=$PATH:$1
            else
                PATH=$1:$PATH
            fi
        esac
    }

if [ -x /usr/bin/id ]; then
"/etc/profile" 78L, 1794C

vim : root
```

Figure 4.3: vi - Text File Editor

4.3.2.1 Using vi

Most of the time a user will want to be in either normal mode to execute some commands or in insert mode to edit a file. If the user is in insert mode, the word "INSERT" will be displayed on the bottom left hand corner. To go into insert mode while in normal mode, press the letter *i*. Optionally press *insert*. If the user presses *insert* for a second time, the word "INSERT" in the left hand corner will change to "REPLACE" and any text typed, will override the previous text.

The return to normal mode, press *Escape*. The following table shows some useful commands that can be executed while in normal mode.

vi Commands	
:w	Write/Save the modifications to the disk
:q	Quit/Exit vi
:w!	Force a write to disk
:q!	Force to quit, without saving changes
:x	Save changes and quit
:x!	Force Save changes and quit
i	Insert Text (IM)
a	Insert Text, at the beginning of the line (IM)
A	Insert Text, at the end of the line (IM)
r	Replace/Override Text (IM)
o	Add a line below the current line (IM)
O	Add a line above the current line (IM)
yy	Yank a line (copy)
y4y	Yank 4 lines (copy)
p	Paste a yanked or deleted line
dd	Delete the current line
d7d	Delete 7 lines from current position
D	Delete text from current position, up to the end of the line
C	Delete text from current position, up to the end of the line (IM)
.(period)	Repeat the last command
Alt+u	Undo the previous command
Ctrl+g	Show file status on "status bar"
G	Go to end of the file
:43	Go to line 43
/	Search for text in the file
n	Search for next instance of search result
N	Search for previous instance of search result
:e!	Undo all changes to file, and reopen last saved version
* (IM) = ... and enter Insert mode	

4.3.2.2 Searching for Strings

vi can search for a specific string (or regular expression) in the document. To search for a string in the document:

Press *escape* to enter normal mode

Press forward slash (/)

Type in the search string

Press *enter*

All the occurrences of the string will be highlighted and the cursor will move to one of the occurrences

To search for the next occurrence in the document press *n*

To search for the previous occurrence in the document press *N* (Shift+n)

To clear the search string, search for something that will not be in the document
Eg. "asfrwqe"

4.3.2.3 Search and replacing strings

vi can also search and replace text in the open file. To search and replace all instances of the word "done" to "achieved", the following can be executed:

Press *escape* to enter normal mode

Press colon (:)

A colon is displayed at the bottom of the screen

Type the following:

%s/done/achieved/g

Press *enter*

The "%s/done/achieved/g" command executed above, tells **vi** to **S**earch for the word **done** and to **r**ep**l**ace the word with **achieved**, **G**lobally (throughout the entire document).

If the user chooses to be prompted to replace the string for each occurrence, the user can change the last character in the command from a "g" to a "c":

%s/done/achieved/c

The above command, tells **vi** to search for the string and to replace it with the provided string but the user should **C**onfirm each action first.

Regular Expressions

Contents

5.1	Overview	58
5.1.1	Character sets	58
5.1.2	Character classes	58
5.1.3	Anchors	59
5.1.4	Modifiers	59
5.1.5	Examples	59

5.1 Overview

Regular expressions are used to search for a string in text that conforms to a specific pattern. For instance if a user is looking for the value *123-5678-9ABC*, the user can use the command **grep "123-5678-9ABC"**. What if the user only knows that the pattern is *###-####-#XXX*? Now the user needs to search for a specific pattern and not a specific value. The user can make use of a regular expression to search for the pattern.

Regular expressions can be used in the shell or with some applications. In this chapter the command **grep** will be used in combination with **cat** to show the use of regular expressions.

A Regular Expression contains one or more of the following:	
Character set or class	Characters retaining their literal meaning
Anchors	The position in the line of text that the RE is to match. Eg ^, and \$
Modifiers	Expand or narrow (modify) the range of text the RE is to match. Eg asterisk, brackets, and backslash

5.1.1 Character sets

Some Character Sets:	
[abc]	Matches any occurrence of the letter "a", "b" or "c".
[s-w]	Matches any character between "s" and "w" (s,t,u,v,w)
[A-Z0-9]	Matches any character A to Z (capital letter) or 0 to 9
[A-Z0-9]*	Matches multiple characters A to Z (capital letter) or 0 to 9
[^f-i]	Matches a character other than f to i
\<the\>	Matches the word "the" but not "them", "there" or "other"
	Logical OR
&	Logical AND

5.1.2 Character classes

Some Character Classes:		
Class	Description	Example
<code>[:alnum:]</code>	Alphabetic and numeric values	a-z A-Z 0-9
<code>[:alpha:]</code>	Alphabetic values	a-z A-Z
<code>[:blank:]</code>	Space or tab character	
<code>[:cntrl:]</code>	Control characters	
<code>[:digit:]</code>	Digits (Numerals)	0-9
<code>[:print:]</code>	Printable characters including space	a-zA-Z0-9 ! @ #
<code>[:lower:]</code>	Lower case characters	a-z
<code>[:upper:]</code>	Upper case characters	A-Z

5.1.3 Anchors

Anchors:	
<code>^</code>	Indicates the beginning of a line
<code>\$</code>	Indicates the end of a line

5.1.4 Modifiers

Modifiers:	
<code>*</code>	Zero or more occurrences
<code>?</code>	Zero or one occurrences
<code>+</code>	One or more occurrences
<code>.</code>	One character

5.1.5 Examples

In these examples, a file was generated using the *fortune* command. The file was saved as */etc/file*.

Listing 5.1: Examples: Regular expressions

```

1
2 #Copy the file /etc/file to the current directory:
3 cp /etc/file .
4 #Search for the word "found"
5 grep "found" file
6 #Search for the word "the"
7 grep "the" file
8 #The previous search found wrong entries too such as:
9 # their,they,these,others etc.
10 #Seach only for the word "the"
11 grep "\<the\>" file

```

```
12 #Search for the word "The" and "the"
13 grep "<[Tt]he>" file
14
15 #Search for double characters "oo", or "ss"
16 grep "[o]\{2\}|[s]\{2\}" file
17 #Search for numbers in the file
18 grep "[0-9]" file
19 #Search for words that are 3 characters long
20 grep "<[A-Za-z]\{3}>" file
21 #Search for words that are 9 or more characters long
22 grep "<[A-Za-z]\{9,\}>" file
23 #Search for a line that starts with an "A"
24 grep "^A" file
25 #Search for a line that ends with an "e"
26 grep "e$" file
27 #Search of a string XXXXXXXX-##
28 grep "[[:alnum:]]*-[0-9]\{2\}" file
```


Shell Scripting

Contents

6.1	Overview	62
6.2	Logical Testing	64
6.2.1	If statement	66
6.2.2	Case statement	67
6.3	Loops	67
6.3.1	For Loop	67
6.3.2	While Loop	68
6.4	Reading input from different sources	68
6.4.1	Reading values from shell environment variables	69
6.4.2	Working with substring parts of shell variables	70
6.4.3	Reading input from the user	71
6.4.4	Reading content from a file	72
6.4.5	Reading parameters and options from shell	73
6.5	Functions	75
6.6	Trapping signals	75

6.1 Overview

A shell script is a collection of commands contained in one single file. Shell scripts can be used for various functions. Some scripts are just used to execute a few commands and display something back to the user. Others are used to modify files, extract text from files and write output to new files.

An other function of a shell script is to set the environmental for an application. Essentially a shell script is the same Linux commands that can be executed in a shell, just stored in a file. It is conventional to save the file with a “.sh” extension. However, it is not necessary to save the file with a .sh extension.

As mentioned in Chapter 2 (Section 2.4 page 21), the PATH environmental variable controls which directories will be searched when a command is executed. One such directory usually includes the bin directory in the users’ home directory (\$HOME/bin). Therefore if a script is created that will be executed from multiple paths, the script can be copied to the user’s home directory.

The first line in the shell script usually starts with something like `#!/bin/bash`. The `#!/...` part is known as the **shebang**. The shebang, in this instance, tells the kernel that a bash interpreter is used when the script is executed. It is important to tell the kernel which interpreter to use by specifying the shebang, or else the wrong interpreter can be called and the script may fail to execute correctly.

Lines in the script starting with (or text after) a hash (#) are comments and are not executed by the shell (except for the shebang as mentioned above). It is good measure to add comments and indentations in shell scripts to make it easier to read.

After creating a shell script, the script needs to be made executable and optionally copied into certain directory that is in the user’s PATH. To make a shell script executable, the user can use the **chmod** command. The following command will make the file called `script.sh` executable: **chmod +x script.sh**

After the script has been made executable, there are a few methods of executing the script. If the script is located in a directory that is in the PATH, the script can be executed by simply typing in the script’s name. The script can be executed by using the **source ScriptName** command. The source command executes the commands that are in the script, in the current environment. This means that the script can set/modify/export variables and have those changes apply to the current environment. A similar method that has the same effect as source, is to execute

the script by typing in dot (.) followed by the path of the script. For instance `./home/user/script.sh`

Another option is to execute the script using the (bash) or sh command followed by the script name/full path. By executing the script in this way, a new shell session is opened and the exported variables are copied to that process. After that process (the script) finishes, the current environment remains unchanged.

Listing 6.1 shows a basic shell script that only prints out a few of the special environmental variables specific to a shell script.

Listing 6.1: Basic Shell Script (script.sh)

```
1 #!/bin/bash
2 echo "This script was called as: $0"
3 echo "The number of parameters sent to this script was: $#"
```

```
4 echo "This script was called with the parameters: $@"
5 echo "The first parameter is : $1"
6 echo "The second parameter is : $2"
7 echo "The process id of this script is: $$"
8 echo "This script was called from: $(pwd)"
9 echo "The name of this script without the path is: $(basename $0)"
10 echo "The path of this script is: $(dirname $0)"
11 exit 0
```

Listing 6.2: Executing the Basic Shell Script

```
1 ./script.sh One Two
2 This script was called as: ./script.sh
3 The number of parameters sent to this script was: 2
4 This script was called with the parameters: One Two
5 The first parameter is : One
6 The second parameter is : Two
7 The process id of this script is: 21568
8 This script was called from: /tmp
9 The name of this script without the path is: script.sh
10 The path of this script is: .
```

6.2 Logical Testing

Logical testing is the testing of a value and determining (depending on the value) what the next step should be. In this section, the if-, test- and case-statements will be discussed.

There are different tests that can be performed and different methods that should be used to test different types of values. For instance the operand used to test a number differs from the operand used to test a string.

For more information execute *info test*.

6.2.0.1 Basic Testing

A basic test can be performed in the shell using square brackets “[]”. To illustrate, Listing 6.3 shows a test for a directory. If the directory exists, a message is displayed. If the directory does not exist, it is created and a message is displayed. Note, the command in Listing 6.3 can be written in one line by removing the two backslashes (“\”) and concatenating the lines.

Listing 6.3: Basic Logical Test

```
1 [ -d /tmp/$USER/ ] \
   && echo "The directory already exists" \
   || echo "Creating the directory"; mkdir -p /tmp/$USER };
```

6.2.0.2 Testing Numbers

Operant	
-eq	Equal
-ne	Not Equal
-lt	Less than
-le	Less than or equal
-gt	Greater than
-ge	Greater than or equal

Example:

```
[ $(date +%k) -lt 12 ] && echo "Good morning" || echo "Good afternoon"
```

6.2.0.3 Testing Text

Operant	
==	Equal
=	Equal
!=	Not Equal
-z	Zero length text (Empty)
-n	Not Zero length text

6.2.0.4 Testing Files and directories

Operant	
-e	Exists
-r	Readable
-w	Writeable
-s	Non Zero sized file (Not Empty)
-d	Type is directory
-f	Type is file
-L	Type is a symbolic link

6.2.1 If statement

The if-statement is used to test a certain condition that may have only one of two outcomes (yes/no). For instance Listing 6.4 shows two variables that determines an action. In this example, if the user has the day off (DAY_OFF=yes) and if it is not raining (RAINING=no) then the user can go play golf (CAN_PLAY_GOLF=yes).

Listing 6.4: Basic Logical Test

```
1 #!/bin/bash
2 RAINING=no
3 DAY_OFF=yes
4
5 if [ "$RAINING" = "no" -a "$DAY_OFF" = "yes" ]; then
6     CAN_PLAY_GOLF=yes
7 else
8     CAN_PLAY_GOLF=no
9 fi
10 echo "Can you play golf today: $CAN_PLAY_GOLF"
```

On Line 5, notice the space after the “[” and before the “]”

On Line 5, the “-a” option is interpreted as AND

Line 6 will only be executed if both conditions are satisfied

Line 8 will be executed if one or both conditions aren’t satisfied

6.2.2 Case statement

A **case** statement is used when a variable can have multiple values, instead of writing an if-statement for each possibility. For instance assume a user has a variable called **SPORT** that is tested against multiple predefined values such as soccer, rugby, golf, tennis or squash. For each of the predefined values, a different action should be performed. Listing 6.5 shows an example of a **case** statement.

Listing 6.5: Basic Logical Test

```
1 #!/bin/bash
2 SPORT=tennis
3 case "$SPORT" in
4     "golf")
5         echo "You better clean your golf clubs first"
6         ;;
7     "soccer" )
8         echo "It's too late for Bafana, maybe not for you"
9         ;;
10    "rugby" )
11        echo "Better bulk up first"
12        ;;
13    "tennis" | "squash")
14        echo "First look for your racket"
15        ;;
16    *)
17        echo "You did not specify what you want to do"
18        echo "I guess you're just going to watch TV then"
19 esac
```

6.3 Loops

Loops are used to execute a set of commands for a certain number of iterations. Two types of loops will be discussed in this section.

6.3.1 For Loop

A **for** loop is used to execute a subset of commands for each iteration in a list of iterations. The number of iterations can be determined from the number of items in a list or from a sequence. Listing 6.6 shows a **for** loop for a sequence ranging from 5 to 400 with a iteration of 3.

Listing 6.6: For Loop

```
1 #!/bin/bash
2 ITERATION=3
3 START=5
4 END=400
5
```

```
6 for i in $(seq $START $ITERATION $END); do
7     (( iCount += 1 ))
8     echo "Iteration Number $iCount for the value $i"
9 done
```

Listing 6.7: For Loop Output

```
1 Iteration Number 1 for the value 5
2 Iteration Number 2 for the value 8
3 Iteration Number 3 for the value 11
4 Iteration Number 4 for the value 14
5 ...
6 Iteration Number 132 for the value 398
```

6.3.2 While Loop

A **while** loop is used to execute a subset of commands for each iteration for an unknown number of iterations or until the logical test becomes false. One should take extra precaution not to run into an instance where an endless loop occurs. The following script shows two while loops. The first one counts from zero to 29 and displays the values on the screen. The second creates an endless loop. If an extra terminal is opened while the endless loop runs, one will note (using **top** or **htop**) that the while process uses the entire CPU time on the thread/core that it is running.

Listing 6.8: While Loop

```
1 #!/bin/bash
2 i=0
3 MAX=30
4 while [ $i -lt $MAX ]; do
5     echo "Value: $i"
6     (( i++))
7 done
8 echo "The following is an endless loop." echo "Press Cntrl+c to cancel
9     " while[1-!t2];do
10    # Perform no action but use : as a place-holder
11    :
12 done
```

6.4 Reading input from different sources

The following sections show how input can be read by shell scripts from different sources such as environment variables, user input and files.

6.4.1 Reading values from shell environment variables

If a variable is defined and exported using either the *declare* or the *export* commands, that variable is copied and available inside the shell script. To access the value, one can simply access the variable as you would in the bash terminal. For instance, the variable `$USER` is usually exported and available in a shell script. Listing 6.9 shows how an exported variable can be accessed:

Listing 6.9: Accessing a variable inside a script

```
1 #!/bin/bash
2 #Set the value of MyName equal to the value that $USER holds
3 MyName=$USER
4 echo "My user name is $MyName"
```

6.4.1.1 Reading/Testing values from possibly empty or not defined shell variables

If a shell-script tries to access the value of a variable that does not exist or is not exported, no error is given. Instead a NULL value is returned/assigned to a variable. This is normal behaviour for a bash script and the script continues without problems. Sometimes, it is required to set a default value when a variable has not been set before. The following script shows how this can be accomplished. The first method is to test if the variable has a value and then set it, using an *if*-statement. This method will work but there is a better way. The better solution would be to make use of a `:-` environment variable reference.

Listing 6.10: Accessing an empty/unset variable and setting a default value

```
1 #!/bin/bash
2
3 #If $Name is defined, set MyName=$Name or else MyName=Albert...
4 if [ -s "$Name" ]; then
5     MyName="$Name"
6 else
7     MyName="Albert van Eck"
8 fi
9
10 #The above statements can be rewritten as:
11 MyName=${Name:-Albert van Eck}
12
13 echo "My name is: $MyName"
```

If a variable is not set, it is sometimes necessary to terminate the script without continuing. The example below shows yet again a bulky method and then the more elegant `?:` method. The following script tests to see if a variable is set and if not; the script will return an error and exit to the shell.

Listing 6.11: Terminate script if variable is not set

```
1 #!/bin/bash
2
3 #If the length of $Name is zero; exit the script:
4 if [ -z "$Name" ]; then
5     echo "The value of Name is not set, we will not continue"
6     exit 1
7 else
8     MyName="$Name"
9 fi
10
11 #The better/shorter method:
12 MyName=${Name:? "The value of Name is not set, we will not continue"}
13
14 #If there was an error above, the next will not be executed
15 echo $MyName
```

The next example shows how to test if a variable holds a value. If the variable is set, then set another variable (Message in this case) to a specific value. If the variable is not defined at all, set the value to an empty value.

Listing 6.12: Define a variable, only if another variable has been set

```
1 #!/bin/bash
2
3 #The long if-statement method:
4 if [ -z "$Name" ]; then
5     Message="Welcome to the system"
6 else
7     Message=
8 fi
9
10 #The shorter version:
11 Message=${Name:+ "Welcome to the system"}
12
13 echo "$Message"
```

6.4.2 Working with substring parts of shell variables

The following script shows how to use some text manipulation methods when working with environment variables.

Listing 6.13: Working with sub-strings in variables

```
1 #!/bin/bash
2 Fruit="grapefruit apples oranges peaches kiwi grapes"
3 echo "The length of the string \${Fruit} = '${#Fruit}'"
4 echo "Substring of chars 0 to 5 is '${Fruit:0:5}'"
5 echo "Substring starting at 34, for 4 chars long is: '${Fruit:34:4}'"
6 echo "Substring of characters 34 to the end is '${Fruit:34}'"
```

```

7 echo
8 #Now we cast the string into a new array:
9 FruitArray=( $Fruit )
10 echo "The number of items in the array is: '${#FruitArray[@]}'"
11 echo "The first item in the array is: '${FruitArray[0]}'"
12 echo
13 ScriptName=$(basename $0)
14 ScriptNameOnly=${ScriptName%.*}
15 ScriptExtension=${ScriptName##$ScriptNameOnly}
16 echo "The full script name is: $ScriptName"
17 echo "The name without the extension is: $ScriptNameOnly"
18 echo "The script's extension is: $ScriptExtension"

```

The above script will produce the following, when executed:

Listing 6.14: Output of listing 6.13

```

1 ./example.sh
2 The length of the string $Fruit = '45'
3 Substring of chars 0 to 5 is 'grape'
4 Substring starting at 34, for 4 chars long is: 'kiwi'
5 Substring of characters 34 to the end is 'kiwi grapes'
6
7 The number of items in the array is: '6'
8 The first item in the array is: 'grapefruit'
9
10 The full script name is: example.sh
11 The name without the extension is: example
12 The script's extension is: .sh

```

6.4.3 Reading input from the user

A script/shell can request input from the user. To read input from the user and save the input to a variable, the **read** command can be used. Listing 6.15 reads the input that the user provides. Note that the **-s** option is used to hide the input from the screen output. Thus, the user will type in some text and won't see the output on the screen. This is useful when a user is prompted for a password.

Listing 6.15: Reading input from the user

```

1 USERNAME=
2 PASSWORD=
3 echo -n "Username : "
4 read USERNAME
5 echo -n "Password : "
6 read -s PASSWORD
7
8 PASSWORDHASH=$(echo $PASSWORD|sed "s|[:print:]|x|g" )
9 echo
10 echo "Username: $USERNAME"
11 echo "Password: $PASSWORDHASH"

```

It is important to remember that the user input is case sensitive. That implies that anything a user types, has to be converted into either upper-case or lower-case before testing the user input. For instance testing a user's input in a case statement; you will have to test the value "dog" against "dog", "Dog", "DOg", "DOG", "DoG" etc. To make testing easier; one can for instance declare the variable as lower-case by executing the (declare -l VariableName) command or one can convert the output to upper/lower case after the fact by piping the result to the transform (`tr [A-Z][a-z]`) command. The following example shows how user input can be tested with an if-statement or a **case**-statement ignoring the case-sensitivity:

Listing 6.16: Reading and then testing user input

```
1 #!/bin/bash
2 declare -l hasPet
3 echo "Please answer the following question yes or no:"
4 echo -n "Do you have a pet:"
5 read hasPet
6 echo
7 #Testing for y and yes... no need to test for Yes/YES/yeS etc.
8 if [ "$hasPet" == "y" -o "$hasPet" == "yes" ]; then
9     echo -n "What is your pet's name: "
10    read PetName
11    case "$(echo $PetName | tr 'A-Z' 'a-z' )" in
12        "fluffy" | "snoopy" | "roger" | "spike" )
13        echo "I also have a pet called $PetName."
14        ;;
15    *)
16        echo "$PetName is such a nice name!"
17        ;;
18    esac
19 else
20    echo "That is too bad."
21 fi
```

6.4.4 Reading content from a file

There are multiple methods to read a file from disk and process it in a shell script. In most instances, a file is either read word for word or line by line. Words are separated by white spaces (space, tab, new line). When a file is read word for word, all the new line characters are changed to spaces. This means that each line will be broken up into words and each word is returned to the script. Listing 6.17 shows how a file is processed word for word:

Listing 6.17: Reading a file word for word using a for loop

```
1 #!/bin/bash
2 for word in $(cat names.txt); do
3     echo "$word"
4 done
```

In the next example, a file is read and the content is processed, line by line. This example is especially useful if a user wants to read a file line for line and perform some tests or other actions on the whole line.

Listing 6.18: Reading a file line for line using a while loop

```

1 #!/bin/bash
2 FILE=/tmp/${USER}_sleeptest
3 [ -f $FILE ] || { echo "The file does not exist."; exit 1; }
4 while read LINE; do
5     echo $LINE |tr "a-z" "A-Z"
6 done < $FILE
7 exit 0

```

6.4.5 Reading parameters and options from shell

A shell script can access the parameters sent to the script by simply referring to the number of the parameter sent to the script. The first parameter sent to the script is accessible inside the script by the variable \$1, the second option is accessible as \$2 and so on, up to \$9. After the ninth option; the variables must be accessed by enclosing the number inside curly brackets: such as \$10.

Listing 6.19: Accessing parameters sent to a script

```

1 #!/bin/bash
2 echo "This script's name is: $0"
3 echo "There were $# parameters parsed to this script"
4 echo "Parameter 1 was: '$1'"
5 echo "All the parameters are: '$@'"

```

When a command is executed in the shell, the user is also able to specify different parameters. For example, the command **tail -n 2 /etc/hosts** has the parameter "-n" with the value "2". If a user wants to write a shell script that reads parameters and values from the shell, the values must be read by the script using the **getopts** command.

Listing 6.20: Reading input from the user

```

1 #!/bin/bash
2 USAGESTRING="usage: $(basename $0) -t TIME -n NODES -m MEM"
3
4 while getopts "t:m:n:h" OPT; do
5     case $OPT in
6         't' )
7             TIME=$OPTARG;;
8         'n' )
9             NSCM=$OPTARG;;
10        'm' )
11            MEM=$OPTARG;;
12        '?' | 'h' )

```

```
13     echo $USAGESTRING
14     exit 1;;
15     esac
16 done
17
18 echo "You chose to execute this command using the following:"
19 echo "Nodes   : $NSCM"
20 echo "Memory  : $MEM"
21 echo "Time    : $TIME"
```

6.5 Functions

If the same set of commands will be executed inside a shell script, it is better to put the set of commands in a **function**. A **function** is only available in the bash shell and not in the c-shell. A function can read a number of parameters and can return values. A function can also return a code to indicate that the function completed successfully or not.

Listing 6.21: A basic Function to do divisions

```
1 #!/bin/bash
2 function div ()
3 {
4     NUMBER=$1
5     DIVIDER=$2
6
7     if [ $DIVIDER -eq 0 ]; then
8         echo "ERROR"
9         return 1
10    fi
11
12    echo "$(( NUMBER / DIVIDER ))"
13    return 0
14 }
15
16 echo "12 / 3 = $(div 12 3)"
17 echo "12 / 0 = $(div 12 0)"
```

6.6 Trapping signals

When an application is closed, a signal is sent to the process. Different signals are sent to applications to initiate different actions. This is helpful especially if a action should be performed just before the application is terminated. For instance, an application can be closed before the application ran in full. Some applications may create files that needs to be removed when the application is closed, or needs to move data from one location before the application terminates. An example could be if an application uses data in a temporary directory and the user terminates the application, the application should first try and move the temporary files to the location where it should reside.

The following table shows some of the more common signals that an application will receive.

A full list can be seen in the man pages: **man 7 signal**

Standard signals		
Name	Value	Description
SIGHUP	1	Death of controlling process
SIGINT	2	Keyboard interruption
SIGKILL	9	Kill signal - Can't be trapped
SIGTERM	15	Termination

Listing 6.22: Trapping signals

```
1 #!/bin/bash
2 PID_FILE=/tmp/${USER}_traptest
3 echo $$ > $PID_FILE
4 declare -i i=1
5
6 function ShowInterupt ()
7 {
8     echo
9     echo "Caught signal $1"
10    date
11    kill -SIGTERM $$
12 }
13
14 function CloseApplication()
15 {
16     [ -e $PID_FILE ] && { echo "Removing PID File $PID_FILE"; rm -
17         f $PID_FILE; }
18     echo "The application will terminate in 2 sec"
19     sleep 2
20     echo "Premature Termination"
21     exit 1
22 }
23 trap "ShowInterupt SIGINT" SIGINT
24 trap "CloseApplication" SIGTERM
25
26 echo "The PID is $$"
27 echo "This application will count until iteration 5 or Ctrl-C is
28     pressed."
29 echo "Each iteration is 10 secs long"
30 while [ $i -lt 5 ]; do
31     echo "Iteration $i"
32     sleep 10
33     (( i++ ))
34 done
35 echo "Iteration $i"
36 echo "Normal Termination"
37 exit 0
```

In Listing 6.22, the method to trap signals is illustrated.

On Line 14, the function “CloseApplication” is declared.

On Line 23, a trap will capture the SIGINT (Cntrl+c) event and execute the CloseApplication function.

When this script is executed and *Cntrl+c* is pressed, the return code from the command **echo \$?** will return 1.

If the script is left to count upto iteration 5, the return code will be 0.

High Performance Computing

Contents

7.1	Overview	80
7.2	Creating a submit File	80
7.3	Submitting a job	82
7.4	Monitor the status of a job	83
7.5	Cancelling a job	83
7.6	Getting job output	84
7.7	Viewing Queues	85
7.8	Viewing nodes	85

7.1 Overview

This chapter describes the different commands and use of a generic HPC. This chapter is based on a batch system called Torque. Torque is a queuing system that is based on PBS (Portable Batch System). In a HPC, two software components are very important. The first component is called the queuing system. In the case of Torque, this system is basically PBS. It is important for the user to know which queuing system is used because the user needs to create descriptive instructions for the specific queuing system.

The second software component is the scheduler. The scheduler is responsible to start the job on a node and to stop the job when the job is complete. Torque makes use of the scheduler called Maui. Maui is based on the Moab scheduler.

More information regarding Torque (PBS/Maui) can be found on the Clusterresources website:

<http://www.clusterresources.com/>

The following work flow is the generic work flow that a user will use to submit a job on a HPC:

01. Log into a HPC Linux server
02. Authenticate to the server
03. Create a sub directory for input files
04. Create a submit file
05. Create/Upload input file(s)
06. Create/Upload an execution script
07. Upload Applications (If not already installed on the HPC)
08. Submit the Job
09. Check job status of the job
10. Get output when done
11. Download output to own computer
12. View/Analyze Output

7.2 Creating a submit File

In the Overview on page 80, 12 steps are mentioned that a HPC user normally follows to submit a job etc. Steps 01 to 03 are straight forward and discussed in previous chapters. This section shows a submit file and describes the different aspects thereof.

When a user wants to execute a job on a HPC, the user needs to describe the resources that the job requires to execute. These resources are listed in a file referred to as a *submit script*. The name submit script implies that the file is a shell script that is used to submit a job.

The submit script is used to perform the following actions:

- * Define which resources are required
- * Define and prepare input/output files
- * Set the environment for the job
- * Execute the job
- * Copies results back
- * Clean up temporary files (Optional)

Listing 7.1: A simple submit script (TestJob.pbs)

```

1 #!/bin/bash
2 #PBS -N Hostname_Test
3 #PBS -l nodes=1:ppn=1:prod
4 #PBS -l walltime=00:10:00
5 #PBS -M vanecka
6 #PBS -S /bin/bash
7 #PBS -m abe
8 #PBS -o general.out
9 #PBS -e general.err
10
11 #####
12 #           Set environment           #
13 #####
14 rshcmd="/usr/bin/ssh -n"
15 export np=1
16 export SCRATCH=/scratch/$USER/general
17 #####
18
19 #Create the scratch paths on all executing nodes
20 MACHINES=`cat nodelist`
21 for machine in $MACHINES; do
22   $rshcmd $machine "mkdir -p $SCRATCH"
23 done
24
25 #####
26 #           Physical job that will run           #
27 #####
28 echo "Execution Started: $(date)"
29 hostname -f
30 sleep 20
31 cd /root
32 echo "Execution Ended: $(date)"
33 #####
34
35 # Cleanup the Scratch
36 for machine in $MACHINES; do
37   $rshcmd $machine "/bin/rm -rf $SCRATCH"
38 done
39
40 exit 0

```

In Listing 7.1, the following settings and commands are set:

Line 2, define the name of the job

Line 3, requests 1 node with 1 CPU core available and that has the production setting set

Line 4, requests that job will run for a maximum of 10 minutes

Line 5, send email to user “vanecka”

Line 6, set the shell to bash

Line 7, send email when job aborts, begins or exits

Line 8, save the output to a file

Line 9, save the errors to a file

The example in Listing 7.1 basically only executes the command **hostname -f**, sleeps for 20 seconds and then tries (and fails) to change to the /root directory.

7.3 Submitting a job

Section 7.2 showed how a submit script can be created. If a submit script is created, the next step is to submit the job.

Listing 7.2: Submitting a job

```
1 qsub TestJob.pbs  
2 37230.man.int.hpc.ufs.ac.za
```

7.4 Monitor the status of a job

When a job is submitted, the status of the job can be monitored. In Listing 7.2, a job identifier was returned. This job identifier can be used to monitor the status of the job. One command that can be used to check the status of a job is the **qstat** command. The **qstat** command can be used with different options. In Listing 7.3, a short description of the status of the job is requested.

Listing 7.3: Job status - summary

```
1 qstat -a 37230.man.int.hpc.ufs.ac.za
2
3 Job ID      Username Queue Jobname   SessID NDS   TSK Time   S Time
4 -----
5 37230.man. testuser short  Hostname_T 26207   1    1 00:10 R   --
```

Listing 7.4: All jobs' statistics

```
1 qstat -a
2
3 Job ID      Username Queue Jobname   SessID NDS   TSK Time   S Time
4 -----
5 23860.man. vansoele long   aic_****   1785    1  48 170:0 R 106:1
6 25271.man. barnarde veryl  FeS_****   17422   1  48 1000: R 345:4
7 25766.man. engelbre veryl  JIr1****   30881   1  48 1000: R 319:0
8 27051.man. freitagr long   Mntf****   21133   1   8 200:0 R 24:20
9 27053.man. freitagr long   Mntf****   12376   1   8 200:0 R 11:33
10 27054.man. freitagr long   Cotf****   23226   1   8 200:0 R 08:14
11 27055.man. freitagr long   Cotf****   7646    1   8 200:0 R 04:18
12 27056.man. freitagr long   Cotf****   --      1   8 200:0 Q  --
```

If the job is not running, the command **showstart** can be used. The **showstart** command shows an estimated time when the job can start execution. This estimation is calculated by checking all the running jobs and checking when the required resources will become available.

Listing 7.5: Job estimated start time

```
1 showstart 27056
2 job 27056 requires 8 procs for 8:08:00:00
3 Earliest start in           00:00:00 on Wed Jul 13 05:33:06
4 Earliest completion in    8:08:00:00 on Thu Jul 21 13:33:06
5 Best Partition: DEFAULT
```

7.5 Cancelling a job

A HPC job can be canceled by using the **qdel** command. A user can not delete another user's jobs. A user can also delete a sequence of jobs.

Listing 7.6: Deleting jobs

```
1 # For instance, say a user wants to delete jobs 1234 1236 and 1288
2 cmdqdel 1234 1236 1288
3 # Now say the user wants to delete jobs 1234 to 1300:
4 qdel {1234..1300}
```

7.6 Getting job output

When a job finished, the user can check the output of the job. By default most HPCs will share the home directories between nodes. By sharing the home directories between nodes, it is not necessary to retrieve job output after the job is done. The HPC will take care of the retrieval task itself. However, the user would like to check the output. By default the output will be saved in the directory where the job executed. Listing 7.7 shows the output of the job that was submitted in Listing 7.2 and Listing 7.8 shows the error that was generated by the job. The error was expected because the job tried to enter the /root directory and a normal user is not able to **cd** to /root.

Listing 7.7: Expected Output (general.out)

```
1 Execution Started: Wed Jul 13 04:54:49 SAST 2011
2 node0317.int.hpc.ufs.ac.za
3 Execution Ended: Wed Jul 13 04:55:09 SAST 2011
```

Listing 7.8: Expected Error (general.err)

```
1 /var/spool/pbs/mom_priv/jobs/37230.man.int.hpc.ufs.ac.za.SC: line 31:
   cd: /root: Permission denied
```


7.7 Viewing Queues

A HPC may have different queues for different jobs, resources, users or walltime settings. A user needs to view the settings of the different queues to know to which queue to submit a job. The `qstat` command is used to view the resources and specifications of a queue. Listing 7.9 shows a list of queues with the statistics of running jobs etc. for each queue. This information could be helpful to see which queues are over saturated etc.

Listing 7.9: Queue statistics

```

1 qstat -Q
2 Queue      Max   Tot   Ena   Str  Que Run Hld Wat Trn Ext T
3 -----
4 short      0  288  yes  yes   0 288  2  0  0  0  E
5 express    0    0  yes  yes   0  0  0  0  0  0  E
6 long       0   53  yes  yes  42 11  0  0  0  0  E
7 gilda      0    0  yes  yes   0  0  0  0  0  0  E
8 medium     0    0  yes  yes   0  0  0  0  0  0  E
9 smp8       0   17  yes  yes  11  6  0  0  0  0  E
10 sagrid    0   10  yes  yes   0 10  0  0  0  0  E
11 parallel  0   10  yes  yes   0 10  0  0  0  0  E
12 verylong  0    2  yes  yes   0  2  0  0  0  0  E
13 add       0    0  yes  yes   0  0  0  0  0  0  R

```

Listing 7.10: Queue Walltime Limits

```

1 qstat -q
2
3 Queue      Memory CPU Time Walltime Node  Run Que Lm  State
4 -----
5 short      --      --      24:00:00  1 384  0 --  E R
6 express    --      --      02:00:00  1  0  0 --  E R
7 long       --      --      300:00:0  1 11 31 --  E R
8 gilda      --      48:00:00 72:00:00  --  0  0 --  E R
9 medium     --      --      150:00:0  1  0  0 --  E R
10 smp8       --      --      1000:00:  --  0  0 --  E R
11 sagrid    --      --      72:00:00  --  0  0 --  E R
12 parallel  --      --      --        --  4  0 --  E R
13 verylong  --      --      1000:00:  1  1  0 --  E R
14 add       --      --      --        --  0  0 --  E R
15
16                               400  31

```

7.8 Viewing nodes

It is useful to check the properties of a node to make sure that the requested resources will be satisfied by a node. To check the status of all nodes (or just one) the command (`pbsnodes`) is used.

Listing 7.11: Viewing a node's properties

```

1 pbsnodes
2 ...
3 node0317.int.hpc.ufs.ac.za
4     state = job-exclusive,busy
5     np = 48
6     properties = prod,grid,g03,gaussian,gate,yasara,espresso,vasp,
7                 gamess_us,crystal,beam,lcgpro,gromacs,blast,express,
8                 amd
9     ntype = cluster
10    jobs = 0/36644.man.int.hpc.ufs.ac.za, 1/36622.man.int.hpc.ufs.ac.
11          za,
12          2/36622.man.int.hpc.ufs.ac.za, 3/36645.man.int.hpc.ufs.ac.
13          za,
14          ....
15          46/36949.man.int.hpc.ufs.ac.za, 47/36659.man.int.hpc.ufs.ac.
16          .za
17    status = rectime=1310401687,varattr=,jobs=36644.man.int.hpc.ufs.
18          ac.za
19          36645.man.int.hpc.ufs.ac.za 36646.man.int.hpc.ufs.ac.za
20          36647.man.int.hpc.ufs.ac.za 36648.man.int.hpc.ufs.ac.za
21          uname=Linux node0317.hpc.ufs.ac.za 2.6.18-194.26.1.el5
22          #1 SMP Tue Nov 9 12:46:16 EST 2010 x86_64,opsys=linux
23    mom_service_port = 15002
24    mom_manager_port = 15003
25    gpus = 0

```

An other script, written by the author, gives a bird's eye view of the resources available on a Torque HPC. The script **nodes-free** can be executed and the results are shown in Listing 7.12.

Listing 7.12: Available cores per node

```

1 nodes-free
2
3 Node                #Cores Free/#Cores          Properties
4
5 node0008             5/ 8
6 node0301             16/48
7 node0302             32/48
8 node0305             22/48
9 node0306             29/48
10 node0308             24/48
11 node0312             8/48
12 node0317            8/48
13
14 Total Nodes Free:   8
15 Total Cores Free:  144

```

Figure 7.1, shows an overall performance graphs collected from a website for the whole HPC.

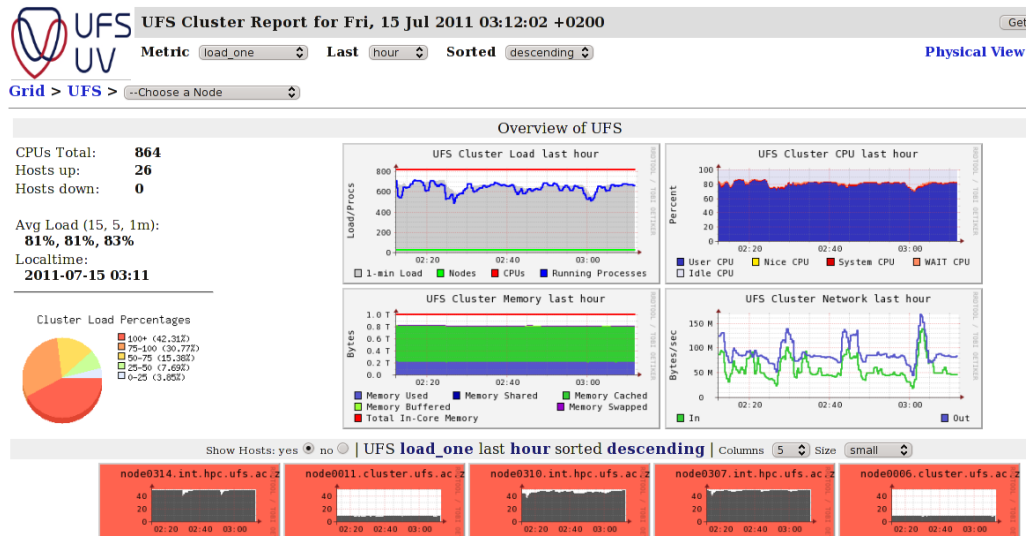


Figure 7.1: HPC Performance Graphs

Bibliography

- [Apache 2019] Apache. *Apache Guacamole*. <https://guacamole.apache.org>, 2019. 13
- [Geeknet 2011] Geeknet. *Xming X Server for Windows*. <http://sourceforge.net/projects/xming>, 2011. 7
- [Prikryl 2011] Martin Prikryl. *WinSCP :: Introducing WinSCP*. <http://winscp.net/eng/docs/introduction>, 2011. 8
- [Tatham 2010] Simon Tatham. *PuTTY - A free SSH and telnet client for Windows*. <http://www.putty.org>, 2010. 4
- [Wikipedia 2011] Wikipedia. *Bash (Unix Shell)*. [http://en.wikipedia.org/wiki/Bash_\(Unix_shell\)](http://en.wikipedia.org/wiki/Bash_(Unix_shell)), 2011. 21

List of Figures

1.1	A console connection	2
1.2	Default PuTTY login screen	4
1.3	PuTTY - Enabling X11 Forwarding	5
1.4	PuTTY - Enabling Mouse Copy and Paste	6
1.5	PuTTY - Saving default session profile	7
1.6	Xming - X Forwarded graphics	8
1.7	WinSCP - Login Screen	9
1.8	WinSCP - File Manager Screen	10
1.9	Guacamole - Graphical Web interface to the HPC	14
4.1	gedit - Graphical Text File Editor	51
4.2	nano - Text File Editor	52
4.3	vi - Text File Editor	53
7.1	HPC Performance Graphs	87

Index

- Bash
 - Commands, *see* command, 30–47
- Case Sensitivity, 18
- command
 - (, 45–47
 - alias, 26, 26, 28, 30
 - Arithmetic, 45–47
 - auto complete, 24
 - basename, 64, 71, 73
 - bg, 26
 - case, 67, 72, 73
 - cat, 39, 40, 58, 59, 72
 - cd, 31, 32, 36, 37, 84
 - chmod, 20, 62
 - clear, 28, 31
 - cp, 38, 59
 - cut, 43
 - date, 76, 81
 - declare, 46, 72, 76
 - df, 34
 - dirname, 64
 - echo, 21, 31, 32, 41, 64, 71, 74, 76
 - examples, 34–47
 - exit, 64, 73, 76, 81
 - export, 21, 23, 81
 - fg, 26
 - find, 43
 - for, 67, 68, 72, 81
 - function, 75, 76
 - gedit, 50
 - getops, 73
 - grep, 27, 28, 33, 43, 58, 59
 - head, 42, 42
 - history, 26
 - hostname, 81
 - HPC
 - nodes-free, 86
 - pbsnodes, 85
 - qdel, 83
 - qstat, 83, 85
 - qsub, 82
 - showstart, 83
 - info, 45, 64
 - kill, 76
 - less, 42
 - let, 45
 - ll, 27
 - logical AND (&&), 32
 - logical OR (||), 32
 - ls, 18, 20, 21, 27, 28, 30, 33, 34, 37, 37, 41
 - man, 28, 34
 - Mathematics, 45–47
 - mkdir, 18, 37, 81
 - more, 41, 42
 - mv, 38
 - nano, 52
 - pwd, 28, 31, 36, 64
 - qvnc, 14
 - read, 71–73
 - return codes, 31
 - reverse search, 25
 - rm, 38, 43
 - rsync, 35, 35, 36
 - scp, 12, 35
 - screen, 44, 45
 - sed, 71
 - seq, 68
 - set, 22
 - sleep, 76, 81
 - sort, 40
 - source, 62
 - special characters, 18

- ssh, 2, 3, 11, 34, 34, 81
- syntax, 32–33
- tail, 31, 32, 42, 45, 73
- trap, 76
- vi, 42, 53, 55
- wget, 26
- which, 27
- while, 73
- who, 43
- xclock, 8, 11
- xterm, 8
- Commands, *see* command
- Console, 2, 2–3, 21
- Copying Directories, *see* Transferring Files
- Copying Files, *see* Transferring Files
- CPU, *see* Central Processing Unit, 82
- Editors, 50–55
 - gedit, 50, 50–51
 - Graphical, 50–51
 - Introduction, 50
 - nano, 52, 51–52
 - Shell, 51–55
 - vi, 42, 51, 53, 53–55
- Escaping characters, 19
- File
 - permission, 19, 20
 - Reading line for line, 68
- File System, 19–21
- Graphical Interface, 13
 - from a web browser, *see* Guacamole
- Guacamole, 13, 13
- Help, 28–29
- Here Document, 39, 40
- HPC, *abv.* High Performance Computing1, 50, 80, 83–86
 - Cancelling a job, 83–84
 - Getting job output, 84
 - Monitor job status, 83
 - queuing system, 80
 - scheduler, 80
 - Submit a job, 82
 - Submit File, 80–82
 - Viewing nodes, 85–86
 - Viewing queues, 85
- Introduction, 18–19
- Login, *see* SSH11
- PBS, *abv.* Portable Batch System1, 80
- Permissions, 19
- Pipe, 33
- Piping, 33
- Portable Batch System, 80
- PuTTY, 6, 50
 - Copy and Paste, 6
 - X11 Forwarding, 50
- Regular Expressions, 58–60
- Remote
 - Access, 2–13
 - Graphical Interface, 15
 - Guacamole, 15
- Remote Host
 - Copying files, *see* Transferring Files
 - Login, *see* SSH
- Secure Shell, *see* SSH
- Session, 2
- shebang, 62
- Shell, 21
 - Alias, 26, 26–28
 - Auto Complete, 24–25
 - bash, 21
 - Commands, *see* command, 30–47
 - Environment, 21–28
 - Environmental Variables, 21–24
 - General Usage, 21–29
 - Logical Testing, 64, 64–67
 - case statement, 67
 - if statement, 66
 - Reading input, 68, 68–74
 - Scripting, 62–77, 80

- Shortcuts, *see* shortcut, 25–26
- shortcut
 - auto complete, 24
 - background, 26
 - backgrounding a process, 26
 - cancel command, 25
 - cancel input, 25
 - clear screen, 25
 - exit, 25
 - history, 26
 - re-execute, 26
 - re-execute last, 26
 - reverse search, 25
 - scrolling through commands, 26
 - scrolling through output screens, 26
- SSH, 11, 11
 - as a different user, 11
 - from Windows, *see* Windows Tools
 - with X Forwarding, 11
- terminal, *see* Console
- Transferring Files, 8–10
 - between Linux machines, 12–13
 - from Windows, 8
 - Linux to Linux, 12–13
 - Linux to Windows, 8
 - scp, 12–13
 - to Windows, 8
 - Windows to Linux, 8
 - WinSCP, 8
- Variables, *see* Environmental Variables
- VNC
 - abv. Virtual Network Computing,
see Guacamole
- Windows Tools
 - PuTTY, 4, 4–8, 15, 50
 - Copy and Paste, 6
 - Saving session, 6–7
 - X11 Forwarding, 5
 - WinSCP, 8, 9
 - Xming, 7, 7–8, 50
- WinSCP, *see* Windows Tools
- X Forwarding, 11
 - to Windows, *see* Xming
 - through SSH, 11
- Xming, *see* Windows Tools

