Operating Systems

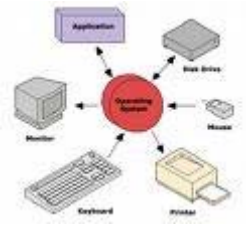
Lab Manual

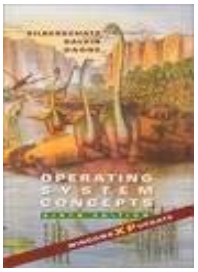
Department of IT and Engineering

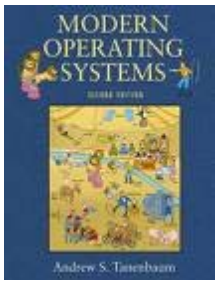
Course Name: Operating Systems
Instructor Name: Farhan Sohail, Naveed Ahmad

Operating Systems Lab Manual

Course Name: Operating Systems







Faculty of IT/Engineering

National University of Modern Languages Islamabad

Table of Contents

Lab No. 1

1.1- Lab objective

Objective of this lab is to give some background of Linux and elaborate the procedure of installing Linux.

1.2- Introduction

In recent years Linux has become a phenomenon. Hardly a day goes by without Linux cropping up in the media in some way. We've lost count of the number of applications that have been made available on Linux and the number of organizations that have adopted it, including some government departments and city administrations.

Major hardware vendors like IBM and Dell now support Linux, and major software vendors like Oracle support their software running on Linux. Linux truly has become a viable operating system, especially in the server market. Linux owes its success to systems and applications that preceded it: UNIX and GNU software.

1.3- What is UNIX

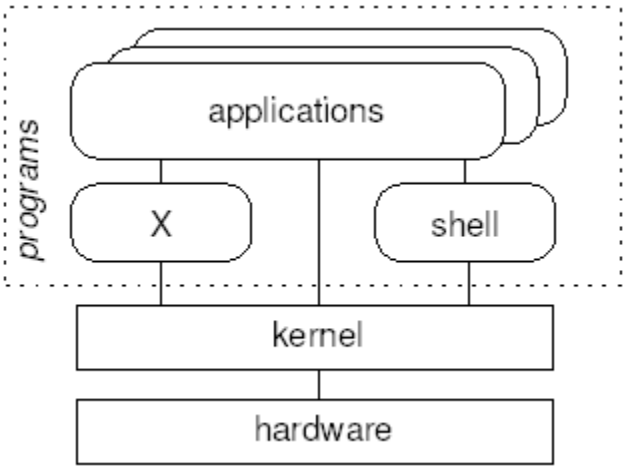
The UNIX operating system was originally developed at Bell Laboratories, once part of the telecommunications giant AT&T. Designed in the 1970s for Digital Equipment PDP computers, UNIX has become a very popular multi-user, multitasking operating system for a wide variety of hardware platforms, from PC workstations to multiprocessor servers and supercomputers.

1.4.0- Background

1.4.1 Unix and Linux

- Linux is based on Unix
  - Unix philosophy
  - Unix commands
  - Unix standards and conventions

1.4.2 Unix System Architecture



- The shell and the window environment are programs
- Programs' only access to hardware is via the kernel

1.4.3 Unix Philosophy

The UNIX operating system, and hence Linux, encourages a certain programming style. Following are a few characteristics shared by typical UNIX programs and systems:

- Multi-user
  - A **user** needs an **account** to use a computer
  - Each user must **log in**
  - Complete separation of different users' files and configuration settings

- Simplicity

- Many of the most useful UNIX utilities are very simple and, as a result, small and easy to understand.

- Small components
  - Each component should perform a single task
  - Multiple components can be combined and chained together for more complex tasks
  - An individual component can be substituted for another, without affecting other components

1.4.4 What is Linux?

- Linux kernel
  - Developed by Linus Torvalds
  - Strictly speaking, 'Linux' is just the kernel
- Associated utilities
  - Standard tools found on (nearly) all Linux systems
  - Many important parts come from the **GNU** project
    - Free Software Foundation's project to make a free Unix
    - Some claim the OS as a whole should be 'GNU/Linux'
- Linux distributions
  - Kernel plus utilities plus other tools, packaged up for end users
  - Generally with installation program
  - Distributors include: Red Hat, Debian, SuSE, Mandrake

1.5- Installation

Pre installation instructions

- Free some space on your hard disk for installing Linux and delete it. Now when Linux installation will run you will have to select this unpartitioned area for your Linux installation..
- Perform media check on Linux installation CDs to confirm the integrity of the installing media.
- There are two modes of installation
    - Texture Interface for professionals
    - Graphical User Interface for novice people

      Its is recommended to select the later option..
- At the time of partitioning, you will be prompted to select 'manual partitioning' or 'automatic partitioning'. Automatic partitioning is recommended for new users.
- For running NS-2 or some other development tools later in Linux, it is recommended to install all development packages at the time of installation. Otherwise you can install them later just like 'add/remove window components' in windows.
- Select the 'Boot from CD option' and start installation..
- To help you during the installation procedure, some tips are normally provided on the left top corner of the screen.

1.6- Exercise

a. What is the file system used by the Linux?

b. Name two types of boot loaders available.

c. What are the names of partitions created for Linux?

Lab No. 2

2.1- Lab objective

This lab introduces few of the basic commands of Linux.

2.2.0- Getting Started with Linux

2.2.1 Using a Linux System

- Login prompt displayed
  - When Linux first loads after booting the computer
  - After another user has logged out
- Need to enter a **username** and **password**
- The login prompt may be graphical or simple text
- If text, logging in will present a **shell**
- If graphical, logging in will present a **desktop**
  - Some combination of mouse movements and keystrokes will make a **terminal window** appear
  - A shell runs in the terminal window

2.2.2 Linux Command Line

- The shell is where commands are invoked
- A command is typed at a **shell prompt**
  - Prompt usually ends in a dollar sign ($)
- After typing a command press Enter to invoke it
  - The shell will try to obey the command
  - Another prompt will appear
- Example:

```
$ date
Sat March 01 11:59:05 BST 2008
$
```

  - The dollar represents the prompt in this course, do not type it

2.2.3 Logging Out

- To exit from the shell, use the exit command
- Pressing Ctrl+D at the shell prompt will also quit the shell
  - Quitting all programs should log you out
  - If in a text-only single-shell environment, exiting the shell should be sufficient
- In a window environment, the window manager should have a log out command for this purpose
- After logging out, a new login prompt should be displayed

2.2.4 Command Syntax

- Most commands take **parameters**
  - Some commands *require* them
  - Parameters are also known as **arguments**
  - For example, echo simply displays its arguments:

    $ **echo**

    $ **echo Hello there**

    Hello there
- Commands are case-sensitive
  - Usually lower-case

    $ **echo whisper**

    whisper

    $ **ECHO SHOUT**

    bash: ECHO: command not found

2.2.5 Files

- Data can be stored in a **file**
- Each file has a **filename**
  - A label referring to a particular file
  - Permitted characters include letters, digits, hyphens (-), underscores (_), and dots (.)
  - Case-sensitive — *NewsCrew.mov* is a different file from *NewScrew.mov*
- The ls command lists the names of files

2.2.6 Creating Files with cat

- There are many ways of creating a file
- One of the simplest is with the cat command:

```
$ cat > shopping_list
cucumber
bread
yoghurts
fish fingers
```

- Note the greater-than sign (>) — this is necessary to create the file
- The text typed is written to a file with the specified name
- Press Ctrl+D after a line-break to denote the end of the file
  - The next shell prompt is displayed
- ls demonstrates the existence of the new file

2.2.7 Displaying Files' Contents with cat

- There are many ways of viewing the contents of a file
- One of the simplest is with the cat command:

```
$ cat shopping_list
cucumber
bread
yoghurts
fish fingers
```

- Note that no greater-than sign is used
- The text in the file is displayed immediately:
  - Starting on the line after the command
  - Before the next shell prompt

2.2.8 Deleting Files with rm

- To delete a file, use the rm ('remove') command
- Simply pass the name of the file to be deleted as an argument:

```
$ rm shopping_list
```

- The file and its contents are removed
  - There is no recycle bin
  - There is no 'unrm' command
- The ls command can be used to confirm the deletion

2.2.9 Unix Command Feedback

- Typically, successful commands do not give any output
- Messages are displayed in the case of errors
- The rm command is typical
    - If it manages to delete the specified file, it does so silently
    - There is no 'File shopping list has been removed' message
    - But if the command fails for whatever reason, a message is displayed
- The silence can be off-putting for beginners
- It is standard behavior, and doesn't take long to get used to

2.2.10 Copying and Renaming Files with cp and mv

- To copy the contents of a file into another file, use the cp command:

  $ **cp CV.pdf old-CV.pdf**
- To rename a file use the mv ('move') command:

  $ **mv commitee_minutes.txt committee_minutes.txt**
    - Similar to using cp then rm
- For both commands, the existing name is specified as the first argument and the new name as the second
    - If a file with the new name already exists, it is overwritten

2.2.11 Filename Completion

- The shell can making typing filenames easier
- Once an unambiguous prefix has been typed, pressing Tab will automatically 'type' the rest
- For example, after typing this:

  $ **rm sho**

  pressing Tab may turn it into this:

  $ **rm shopping_list**

- This also works with command names
    - For example, da may be completed to date if no other commands start 'da'

2.2.12 Command History

- Often it is desired to repeat a previously-executed command
- The shell keeps a **command history** for this purpose
    - Use the *Up* and Down cursor keys to scroll through the list of previous commands
    - Press *Enter* to execute the displayed command
- Commands can also be edited before being run
    - Particularly useful for fixing a typo in the previous command
    - The Left and Right cursor keys navigate across a command
    - Extra characters can be typed at any point
    - Backspace deletes characters to the left of the cursor
    - Del and Ctrl+D delete characters to the right
        - Take care not to log out by holding down Ctrl+D too long

2.3- Skills Developed

By completing the second lab, one should have basic understanding of Linux environment and few Linux commands.

2.4- Exercises

Q1

a. Log in.
b. Log out.
c. Log in again. Open a terminal window, to start a shell.
d. Exit from the shell; the terminal window will close.
e. Start another shell. Enter each of the following commands in turn.
    i. date
    ii. whoami
    iii. hostname
    iv. uname
    v. uptime

Q2

a. Use the ls command to see if you have any files.
b. Create a new file using the cat command as follows:

    $ **cat > hello.txt**

    Hello world!

    This is a text file.

c. Press Enter at the end of the last line, then Ctrl+D to denote the end of the file.
d. Use ls again to verify that the new file exists.
e. Display the contents of the file.
f. Display the file again, but use the cursor keys to execute the same command again without having to retype it.

Q3

a. Create a second file. Call it *secret-of-the-universe*, and put in whatever content you deem appropriate.

b. Check its creation with ls.

c. Display the contents of this file. Minimize the typing needed to do this:

   i. Scroll back through the command history to the command you used to create the file.

   ii. Change that command to display *secret-of-the-universe* instead of creating it.

Q4

After each of the following steps, use ls and cat to verify what has happened.

a. Copy *secret-of-the-universe* to a new file called *answer.txt*. Use Tab to avoid typing the existing file's name in full.

b. Now copy *hello.txt* to *answer.txt*. What's happened now?

c. Delete the original file, *hello.txt*.

d. Rename *answer.txt* to *message*.

e. Try asking rm to delete a file called *missing*. What happens?

f. Try copying *secret-of-the-universe* again, but don't specify a filename to which to copy. What happens now?

Lab No. 3

3.1- Lab objective

In this lab, you will explore the Linux file system, including the basic concepts of files and directories and their organization in a hierarchical tree structure.

3.2.0- Background

3.2.1 File and Directories

- A **directory** is a collection of files and/or other directories
  - Because a directory can contain other directories, we get a directory **hierarchy**
- The 'top level' of the hierarchy is the **root directory**
- Files and directories can be named by a **path**
  - Shows programs how to find their way to the file
  - The root directory is referred to as /
  - Other directories are referred to by name, and their names are separated by slashes (/)
- If a path refers to a directory it can end in /
  - Usually an extra slash at the end of a path makes no difference

3.3.0- Linux Files and Directories

3.3.1 Examples of Absolute Paths

- An **absolute path** starts at the root of the directory hierarchy, and names directories under it:

  /etc/hostname

- Meaning the file called *hostname* in the directory *etc* in the root directory

- We can use ls to list files in a specific directory by specifying the absolute path:

  $ **ls /usr/share/doc/**

3.3.2 Current Directory

- Your shell has a **current directory** — the directory in which you are currently working
- Commands like ls use the current directory if none is specified
- Use the pwd (print working directory) command to see what your current directory is:

  $ **pwd**

  /home/fred
- Change the current directory with cd:

  $ **cd /mnt/cdrom**

  $ **pwd**

  /mnt/cdrom
- Use cd without specifying a path to get back to your home directory

3.3.3 Making and Deleting Directories

- The mkdir command makes new, empty, directories
- For example, to make a directory for storing company accounts:

  $ **mkdir Accounts**
- To delete an empty directory, use rmdir:

  $ **rmdir OldAccounts**
- Use rm with the -r (recursive) option to delete directories and all the files they contain:

$ **rm -r OldAccounts**

- Be careful — rm can be a dangerous tool if misused

3.3.4 Relative Paths

- Paths don't have to start from the root directory
  - A path which doesn't start with / is a **relative path**
  - It is relative to some other directory, usually the current directory
- For example, the following sets of directory changes both end up in the same directory:

  $ **cd /usr/share/doc**

  $ **cd /**

  $ **cd usr**

  $ **cd share/doc**
- Relative paths specify files inside directories in the same way as absolute ones

3.3.5 Special Dot Directories

- Every directory contains two special filenames which help making relative paths:
  - The directory .. points to the parent directory
    - ls .. will list the files in the parent directory
  - For example, if we start from */home/fred*:

    $ **cd ..**

    $ **pwd**

    /home

    $ **cd ..**

    $ **pwd**

    /
- The special directory . points to the directory it is in

- So ./foo is the same file as foo

3.3.6 Using Dot Directories in Paths

- The special .. and . directories can be used in paths just like any other directory name:

  $ **cd ../other-dir/**

  - Meaning "the directory *other-dir* in the parent directory of the current directory"
- It is common to see .. used to 'go back' several directories from the current directory:

  $ **ls ../../../../far-away-directory/**
- The . directory is most commonly used on its own, to mean "the current directory"

3.3.7 Hidden Files

- The special . and .. directories don't show up when you do ls
  - They are **hidden files**
- Simple rule: files whose names start with . are considered 'hidden'
- Make ls display all files, even the hidden ones, by giving it the -a (all) option:

  $ **ls -a**

  . .. .bashrc .profile report.doc
- Hidden files are often used for configuration files
  - Usually found in a user's home directory
- You can still read hidden files — they just don't get listed by ls by default

3.3.8 Paths to Home Directories

- The symbol ~ (tilde) is an abbreviation for your home directory
  - So for user 'fred', the following are equivalent:

    $ **cd /home/fred/documents/**

    $ **cd ~/documents/**
- The ~ is **expanded** by the shell, so programs only see the complete path
- You can get the paths to other users' home directories using ~, for example:

  $ **cat ~alice/notes.txt**
- The following are all the same for user 'fred':

  $ **cd**

  $ **cd ~**

  $ **cd /home/fred**

3.3.9 Looking for Files in the System

- The command locate lists files which contain the text you give
- For example, to find files whose name contains the word 'mkdir':

  $ **locate mkdir**

  /usr/man/man1/mkdir.1.gz

  /usr/man/man2/mkdir.2.gz

  /bin/mkdir

  ...
- locate is useful for finding files when you don't know exactly what they will be called, or where
- they are stored
- For many users, graphical tools make it easier to navigate the filesystem
  - Also make file management simpler

3.3.10 Running Programs

- Programs under Linux are files, stored in directories like */bin* and */usr/bin*
  - Run them from the shell, simply by typing their name
- Many programs take options, which are added after their name and prefixed with -
- For example, the -l option to ls gives more information, including the size of files and the date
- they were last modified:

  $ **ls -l**

drwxrwxr-x	2	fred	users	4096	Mar 01	10:57	Accounts
-rw-rw-r--	1	fred	users	345	Mar 01	10:57	notes.txt
-rw-r--r--	1	fred	users	3255	Mar 01	10:57	report.txt

- Many programs accept filenames after the options
  - Specify multiple files by separating them with spaces

3.3.11 Specifying Multiple Files

- Most programs can be given a list of files
  - For example, to delete several files at once:

    $ **rm oldnotes.txt tmp.txt stuff.doc**
  - To make several directories in one go:

    $ **mkdir Accounts Reports**
- The original use of cat was to join multiple files together
  - For example, to list two files, one after another:

    $ **cat notes.txt morenotes.txt**
- If a filename contains spaces, or characters which are interpreted by the shell (such as *), put
- single quotes around them:

  $ **rm 'Beatles - Strawberry Fields.mp3'**

  $ **cat '* important notes.txt *'**

3.3.12 Finding Documentation for Programs

- Use the man command to read the manual for a program
- The manual for a program is called its **man page**
    - Other things, like file formats and library functions also have man pages
- To read a man page, specify the name of the program to man:

  $ **man mkdir**
- To quit from the man page viewer press q
- Man pages for programs usually have the following information:
    - A description of what it does
    - A list of options it accepts
    - Other information, such as the name of the author

3.3.12 Specifying Files with Wildcards

- Use the * wildcard to specify multiple filenames to a program:

  $ **ls -l *.txt**

  -rw-rw-r-- 1 fred users 108 Nov 16 13:06 report.txt

  -rw-rw-r-- 1 fred users 345 Jan 18 08:56 notes.txt

- The shell expands the wildcard, and passes the full list of files to the program
- Just using * on its own will expand to all the files in the current directory:

  $ **rm ***

  - (All the files, that is, except the hidden ones)
- Names with wildcards in are called **globs**, and the process of expanding them is called **globbing**

3.3.13 Chaining Programs Together

- The who command lists the users currently logged in
- The wc command counts bytes, words, and lines in its input
- We combine them to count how many users are logged in:

  $ **who | wc -l**
- The | symbol makes a **pipe** between the two programs
  - The output of who is fed into wc
- The -l option makes wc print only the number of lines
- Another example, to join all the text files together and count the words, lines and characters in
- the result:

  $ **cat *.txt | wc**

3.3.14 Graphical and Text Interfaces

- Most modern desktop Linux systems provide a **graphical user interface** (GUI)
- Linux systems use the X window system to provide graphics
  - X is just another program, not built into Linux
  - Usually X is started automatically when the computer boots
- Linux can be used without a GUI, just using a command line
- Use Ctrl+Alt+F1 to switch to a text console — logging in works as it does in X
  - Use Ctrl+Alt+F2, Ctrl+Alt+F3, etc., to switch between virtual terminals — usually about 6 are provided
  - Use Ctrl+Alt+F7, or whatever is after the virtual terminals, to switch back to X

3.3.15 Text Editors

- Text editors are for editing plain text files
  - Don't provide advanced formatting like word processors
  - Extremely important — manipulating text is Unix's *raison d'être*
- The most popular editors are Emacs and Vim, both of which are very sophisticated, but take
- time to learn
- Some programs run a text editor for you
  - They use the $EDITOR variable to decide which editor to use
  - Usually it is set to vi, but it can be changed
  - Another example of the component philosophy

3.4- Exercises

Q1

a. Use the pwd command to find out what directory you are in.
b. If you are not in your home directory (*/home/USERNAME*) then use cd without any arguments to go there, and do pwd again.
c. Use cd to visit the root directory, and list the files there. You should see *home* among the list.
d. Change into the directory called *home* and again list the files present. There should be one directory for each user, including the user you are logged in as (you can use whoami to check that).
e. Change into your home directory to confirm that you have gotten back to where you started.

Q2

a. Create a text file in your home directory called *shakespeare*, containing the following text:
   Shall I compare thee to a summer's day?
   Thou art more lovely and more temperate
b. Rename it to *sonnet-18.txt*.
c. Make a new directory in your home directory, called *poetry*.
d. Move the poem file into the new directory.
e. Try to find a graphical directory-browsing program, and find your home directory with it. You should also be able to use it to explore some of the system directories.
f. Find a text editor program and use it to display and edit the sonnet.

Q3

a. From your home directory, list the files in the directory */usr/share*.
b. Change to that directory, and use pwd to check that you are in the right place. List the files in the current directory again, and then list the files in the directory called *doc*.
c. Next list the files in the parent directory, and the directory above that.
d. Try the following command, and make sure you understand the result:

   $ **echo ~**

e. Use cat to display the contents of a text file which resides in your home directory (create one if you haven't already), using the ~/ syntax to refer to it. It shouldn't matter what your current directory is when you run the command.

Q4

a. Use the hostname command, with no options, to print the hostname of the machine you are using.
b. Use man to display some documentation on the hostname command. Find out how to make it print the IP address of the machine instead of the hostname. You will need to scroll down the manpage to the 'Options' section.

c. Use the locate command to find files whose name contains the text 'hostname'. Which of the filenames printed contain the actual hostname program itself? Try running it by entering the program's absolute path to check that you really have found it.

Q5

a. The * wildcard on its own is expanded by the shell to a list of all the files in the current directory. Use the echo command to see the result (but make sure you are in a directory with a few files or directories first)

b. Use quoting to make echo print out an actual * symbol.

c. Augment the *poetry* directory you created earlier with another file, *sonnet-29.txt*:

   When in disgrace with Fortune and men's eyes,

   I all alone beweep my outcast state,

d. Use the cat command to display both of the poems, using a wildcard.

e. Finally, use the rm command to delete the *poetry* directory and the poems in it.

Lab No. 4

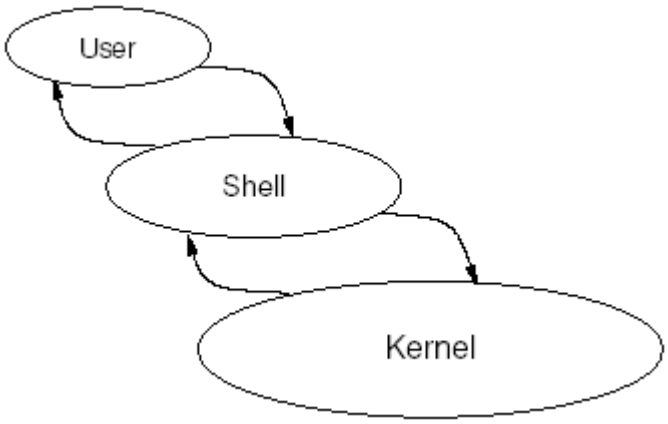
4.1.0- Lab objective

This lab will give overview of Linux shells. You will get insight of 'bash' shell.

4.1.1- Background

Shells

- A **shell** provides an interface between the user and the operating system kernel
- Either a **command interpreter** or a graphical user interface
- Traditional Unix shells are **command-line interfaces** (CLIs)
- Usually started automatically when you log in or open a terminal



4.2.0- Work Effectively on the Linux Command Line

4.2.1 The Bash Shell

Linux's most popular command interpreter is called bash

- The **Bourne-Again Shell**
  - More sophisticated than the original sh by Steve Bourne
  - Can be run as sh, as a replacement for the original Unix shell
  - Gives you a prompt and waits for a command to be entered
- Although this course concentrates on Bash, the shell tcsh is also popular
  - Based on the design of the older C Shell (csh)

4.2.2 Shell Commands

- Shell commands entered consist of words
  - Separated by spaces (whitespace)
  - The first word is the command to run
  - Subsequent words are options or arguments to the command
- For several reasons, some commands are built into the shell itself
  - Called **builtins**
  - Only a small number of commands are builtins, most are separate programs

4.2.3 Command-Line Arguments

- The words after the command name are passed to a command as a list of **arguments**
- Most commands group these words into two categories:
  - Options, usually starting with one or two hyphens
  - Filenames, directories, etc., on which to operate

- The options usually come first, but for most commands they do not need to
- There is a special option '--' which indicates the end of the options
  - Nothing after the double hyphen is treated as an option, even if it starts with -

4.2.4 Syntax of Command-Line Options

- Most Unix commands have a consistent syntax for options:
  - Single letter options start with a hyphen, e.g., -B
  - Less cryptic options are whole words or phrases, and start with two hyphens, for example
    --ignore-backups
- Some options themselves take arguments
  - Usually the argument is the next word: sort -o *output_file*
- A few programs use different styles of command-line options
  - For example, long options (not single letters) sometimes start with a single – rather than - -

4.2.5 Examples of Command-Line Options

- List all the files in the current directory:
  $ **ls**
- List the files in the 'long format' (giving more information):
  $ **ls -l**
- List full information about some specific files:
  $ **ls -l notes.txt report.txt**
- List full information about all the *.txt* files:
  $ **ls -l *.txt**
- List all files in long format, even the hidden ones:
  $ **ls -l -a**

$ **ls -la**

4.2.6 Setting Shell Variables

- **Shell variables** can be used to store temporary values
- Set a shell variable's value as follows:

  $ **files="notes.txt report.txt"**

  - The double quotes are needed because the value contains a space
  - Easiest to put them in all the time
- Print out the value of a shell variable with the echo command:

  $ **echo $files**

  - The dollar ($) tells the shell to insert the variable's value into the command line
- Use the set command (with no arguments) to list all the shell variables

4.2.7 Environment Variables

- Shell variables are private to the shell
- A special type of shell variables called **environment variables** are passed to programs run from the shell
- A program's **environment** is the set of environment variables it can access
  - In Bash, use export to export a shell variable into the environment:

    $ **files="notes.txt report.txt"**

    $ **export files**
  - Or combine those into one line:

    $ **export files="notes.txt report.txt"**
- The env command lists environment variables

4.2.8 Where Programs are Found

- The location of a program can be specified explicitly:
  - ./sample runs the sample program in the current directory
  - /bin/ls runs the ls command in the */bin* directory
- Otherwise, the shell looks in standard places for the program
  - The variable called PATH lists the directories to search in
  - Directory names are separated by colon, for example:

    $ **echo $PATH**

    /bin:/usr/bin:/usr/local/bin
  - So running whoami will run */bin/whoami* or */usr/bin/whoami* or */usr/local/bin/whoami* (whichever is found first)

4.2.9 Bash Configuration Variables

- Some variables contain information which Bash itself uses
  - The variable called PS1 (Prompt String 1) specifies how to display the shell prompt
- Use the echo command with a $ sign before a varable name to see its value, e.g.

  $ **echo $PS1**

  [\u@\h \W]\$
- The special characters \u, \h and \W represent shell variables containing, respectively, your user/login name, machine's hostname and current working directory, i.e.,
  - $USER, $HOSTNAME, $PWD

4.2.10 Using History

- Previously executed commands can be edited with the Up or Ctrl+P keys
- This allows old commands to be executed again without re-entering
- Bash stores a **history** of old commands in memory
  - Use the built-in command history to display the lines remembered
  - History is stored between sessions in the file *~/.bash_history*
- Bash uses the readline library to read input from the user
  - Allows Emacs-like editing of the command line
  - Left and Right cursor keys and Delete work as expected

4.2.11 Reusing History Items

- Previous commands can be used to build new commands, using **history expansion**
- Use !! to refer to the previous command, for example:

  $ **rm index.html**

  $ **echo !!**

  echo rm index.html

  rm index.html
- More often useful is !*string*, which inserts the most recent command which started with *string*
  - Useful for repeating particular commands without modification:

    $ **ls *.txt**

    notes.txt report.txt

    $ **!ls**

    ls *.txt

    notes.txt report.txt

4.2.12 Retrieving Arguments from the History

- The event designator !$ refers to the last argument of the previous command:

  $ **ls -l long_file_name.html**

  -rw-r--r-- 1 jeff users 11170 Feb 20 10:47 long_file_name.html

  $ **rm !$**

  rm long_file_name.html

- Similarly, !^ refers to the first argument
- A command of the form ^*string*^*replacement*^ replaces the first occurrence of *string* with *replacement* in the previous command, and runs it:

  $ **echo $HOTSNAME**

  $ **^TS^ST^**

  echo $HOSTNAME

  tiger

4.2.13 Summary of Bash Editing Keys

- These are the basic editing commands by default:
  - Right — move cursor to the right
  - Left — move cursor to the left
  - Up — previous history line
  - Down — next history line
  - Ctrl+A — move to start of line
  - Ctrl+E — move to end of line
  - Ctrl+D — delete current character
- There are alternative keys, as for the Emacs editor, which can be more comfortable to use than the cursor keys
- There are other, less often used keys, which are documented in the bash man page (section 'Readline')

4.2.14 Combining Commands on One Line

- You can write multiple commands on one line by separating them with ;
- Useful when the first command might take a long time:
  time-consuming-program; ls
- Alternatively, use && to arrange for subsequent commands to run only if earlier ones succeeded:
  time-consuming-potentially-failing-program && ls

4.2.15 Repeating Commands with 'for'

- Commands can be repeated several times using for
  - Structure: for *varname* in *list*; do *commands...*; done
- For example, to rename all *.txt* files to *.txt.old* :

```
$ for file in *.txt;
> do
> mv -v $file $file.old;
> done
barbie.txt -> barbie.txt.old
food.txt -> food.txt.old
quirks.txt -> quirks.txt.old
```

- The command above could also be written on a single line

4.2.16 Command Substitution

- **Command substitution** allows the output of one command to be used as arguments to another
- For example, use the locate command to find all files called *manual.html* and print information about them with ls:

```
$ ls -l $(locate manual.html)
$ ls -l 'locate manual.html'
```

- The punctuation marks on the second form are opening single quote characters, called **backticks**
  - The $() form is usually preferred, but backticks are widely used
- Line breaks in the output are converted to spaces
- Another example: use vi to edit the last of the files found:

  $ **vi $(locate manual.html | tail -1)**

4.2.17 Finding Files with locate

- The locate command is a simple and fast way to find files
- For example, to find files relating to the email program mutt:

  $ **locate mutt**
- The locate command searches a database of filenames
  - The database needs to be updated regularly
  - Usually this is done automatically with cron
  - But locate will not find files created since the last update
- The -i option makes the search case-insensitive
- -r treats the pattern as a regular expression, rather than a simple string

4.2.18 Finding Files More Flexibly: 'find'

- locate only finds files by name
- *find* can find files by any combination of a wide number of criteria, including name
- Structure: *find directories criteria*
- Simplest possible example: find .
- Finding files with a simple criterion:

  $ **find . -name manual.html**

  Looks for files under the current directory whose name is *manual.html*
- The *criteria* always begin with a single hyphen, even though they have long names

4.2.19 'find' Criteria

- find accepts many different criteria; two of the most useful are:
  - -name *pattern*: selects files whose name matches the shell-style wildcard *pattern*
  - -type d, -type f: select directories or plain files, respectively
- You can have complex selections involving 'and', 'or', and 'not'

4.2.20 'find' Actions: Executing Programs

- *find* lets you specify an action for each file found; the default action is simply to print out the name
  - You can alternatively write that explicitly as -print
- Other actions include executing a program; for example, to delete all files whose name starts with *manual*:

  find . -name 'manual*' -exec rm '{}' ';'
- The command rm '{}' is run for each file, with '{}' replaced by the filename
- The {} and ; are required by *find*, but must be quoted to protect them from the shell

4.3- Exercises

Q1

a. Use the df command to display the amount of used and available space on your hard drive.
b. Check the man page for df, and use it to find an option to the command which will display the free space in a more human-friendly form. Try both the single-letter and long-style options.
c. Run the shell, bash, and see what happens. Remember that you were already running it to start with. Try leaving the shell you have started with the exit command.

Q2

a. Try ls with the -a and -A options. What is the difference between them?
b. Write a for loop which goes through all the files in a directory and prints out their names with echo. If you write the whole thing on one line, then it will be easy to repeat it using the command line history.
c. Change the loop so that it goes through the names of the people in the room (which needn't be the names of files) and print greetings to them.
d. Of course, a simpler way to print a list of filenames is echo *. Why might this be useful, when we usually use the ls command?

Q3

a. Use the find command to list all the files and directories under your home directory. Try the -type d and -type f criteria to show just files and just directories.

b. Use 'locate' to find files whose name contains the string 'bashbug'. Try the same search with find, looking over all files on the system. You'll need to use the * wildcard at the end of the pattern to match files with extensions.

c. Find out what the find criterion -iname does.

Lab No. 5

5.1.0- Lab objective

This lab will give overview of Basic File and Directory Management utilities.

5.1.1 File system Objects

- A **file** is a place to store data: a possibly-empty sequence of bytes
- A **directory** is a collection of files and other directories
    - Directories are organized in a hierarchy, with the **root directory** at the top
- The root directory is referred to as /

$**cp rm jeff /**

5.1.2 Directory and File Names

- Files and directories are organized into a **file-system**
- Refer to files in directories and sub-directories by separating their names with /, for example:
    - /bin/ls
    - /usr/share/dict/words
    - /home/jeff/recipe
- Paths to files either start at / (absolute) or from some 'current' directory

5.1.3 File Extensions

- It's common to put an **extension**, beginning with a dot, on the end of a filename
- The extension can indicate the type of the file:
    - *.txt* Text file
    - *.gif* Graphics Interchange Format image

- *.jpg* Joint Photographic Experts Group image
- *.mp3* MPEG-2 Layer 3 audio
- *.gz* Compressed file
- *.tar* Unix 'tape archive' file
- *.tar.gz*, *.tgz* Compressed archive file

- On Unix and Linux, file extensions are just a convention
- The kernel just treats them as a normal part of the name
- A few programs use extensions to determine the type of a file

5.2 Going Back to Previous Directories

- The pushd command takes you to another directory, like cd
- But also saves the current directory, so that you can go back later
  - For example, to visit Fred's home directory, and then go back to where you started from:

```
$ pushd ~fred
$ cd Work
$ ls
...
$ popd
```

- popd takes you back to the directory where you last did pushd
- dirs will list the directories you can pop back to

5.3 Filename Completion

- Modern shells help you type the names of files and directories by completing partial names
- Type the start of the name (enough to make it unambiguous) and press Tab
- For an ambiguous name (there are several possible completions), the shell can list the options:

- For Bash, type Tab twice in succession
- For C shells, type Ctrl+D

- Both of these shells will automatically escape spaces and special characters in the filenames

5.4 Wildcard Patterns

- Give commands multiple files by specifying patterns
- Use the symbol * to match any part of a filename:

$ ls *.txt

accounts.txt letter.txt report.txt

- Just * produces the names of all files in the current directory
- The wildcard ? matches exactly one character:

$ rm -v data.?

removing data.1

removing data.2

removing data.3

- Note: wildcards are turned into filenames by the shell, so the program you pass them to can't tell that those names came from wildcard expansion

5.5.0 Copying Files with cp

- Syntax: cp [*options*] *source-file destination-file*
- Copy multiple files into a directory: cp *files directory*
- Common options:
  - -f, force overwriting of destination files
  - -i, interactively prompt before overwriting files
  - -a, archive, copy the contents of directories recursively

5.5.1 Examples of cp

- Copy */etc/smb.conf* to the current directory:

$ cp /etc/smb.conf .

- Create an identical copy of a directory called *work*, and call it *work-backup*:

$ cp -a work work-backup

- Copy all the GIF and JPEG images in the current directory into *images*:

$ cp *.gif *.jpeg images/

5.6 Moving Files with mv

- mv can rename files or directories, or move them to different directories
    - It is equivalent to copying and then deleting
    - But is usually much faster
- Options:
    - -f, force overwrite, even if target already exists
    - -i, ask user interactively before overwriting files
    - For example, to rename *poetry.txt* to *poems.txt*:

$ mv poetry.txt poems.txt

    - To move everything in the current directory somewhere else:

$ mv * ~/old-stuff/

5.7.0 Deleting Files with rm

- rm deletes ('removes') the specified files
- You must have write permission for the directory the file is in to remove it
    - Use carefully if you are logged in as root!
- Options:

- -f, delete write-protected files without prompting
- -i, interactive — ask the user before deleting files
- -r, recursively delete files and directories
- For example, clean out everything in */tmp*, without prompting to delete each file:

$ rm -rf /tmp/*

5.7.1 Deleting Files with Peculiar Names

- Some files have names which make them hard to delete
  - Files that begin with a minus sign:

$ rm ./-filename

$ rm -- -filename

- Files that contain peculiar characters — perhaps characters that you can't actually type on your keyboard:
- Write a wildcard pattern that matches *only* the name you want to delete:

$ rm -i ./name-with-funny-characters*

  - The ./ forces it to be in the current directory
- Using the -i option to rm makes sure that you won't delete anything else by accident

5.8.0 Making Directories with mkdir

- Syntax: mkdir *directory-names*
- Options:
  - -p, create intervening parent directories if they don't already exist
  - -m *mode*, set the access permissions to *mode*

- For example, create a directory called *mystuff* in your home directory with permissions so that only you can write, but eveyone can read it:

$ mkdir -m 755 ~/mystuff

- Create a directory tree in */tmp* using one command with three subdirectories called *one*, *two* and *three*:

$ mkdir -p /tmp/one/two/three

5.8.1 Removing Directories with rmdir

- rmdir deletes *empty* directories, so the files inside must be deleted first
  - For example, to delete the *images* directory:

$ rm images/*

$ rmdir images

- For non-empty directories, use rm -r *directory*
- The -p option to rmdir removes the complete path, if there are no other files and directories in it
- These commands are equivalent:

$ rmdir -p a/b/c

$ rmdir a/b/c a/b a

5.9 Changing Timestamps with touch

- Changes the **access** and **modification** times of files
- Creates files that didn't already exist
- Options:
  - -a, change only the access time
  - -m, change only the modification time
  - -t [*YYYY*]*MMDDhhmm*[*.ss*], set the timestamp of the file to the specified date and time

- GNU touch has a -d option, which accepts times in a more flexible format
    - For example, change the time stamp on *homework* to January 20 2001, 5:59p.m.

$ touch -t 200101201759 homework

5.10 Exercises

Q1

a. Use cd to go to your home directory, and create a new directory there called *dog*.

b. Create another directory within that one called *cat*, and another within that called *mouse*.

c. Remove all three directories. You can either remove them one at a time, or all at once.

d. If you can delete directories with rm -r, what is the point of using rmdir for empty directories?

e. Try creating the *dog/cat/mouse* directory structure with a single command.

Q2

a. Copy the file */etc/passwd* to your home directory, and then use cat to see what's in it.

b. Rename it to *users* using the mv command.

c. Make a directory called *programs* and copy everything from */bin* into it.

d. Delete all the files in the *programs* directory.

e. Delete the empty *programs* directory and the *users* file.

Q3

a. The touch command can be used to create new empty files. Try that now, picking a name for the new

file:

$ touch baked-beans

b. Get details about the file using the ls command:

$ ls -l baked-beans

c. Wait for a minute, and then try the previous two steps again, and see what changes. What happens

when we don't specify a time to touch?

d. Try setting the timestamp on the file to a value in the future.

e. When you're finished with it, delete the file.

Lab No. 6

6.1.0- Lab objective

This lab will give overview of Processing Text Streams using Text Processing Filters

6.1.1 Working with Text Files

- Unix-like systems are designed to manipulate text very well
- The same techniques can be used with plain text, or text-based formats
- Most Unix configuration files are plain text
- Text is usually in the **ASCII** character set
- Non-English text might use the ISO-8859 character sets
- Unicode is better, but unfortunately many Linux command-line utilities don't (directly) support it yet

6.1.2 Lines of Text

- Text files are naturally divided into lines
- In Linux a line ends in a **line feed** character
- Character number 10, hexadecimal 0x0A
- Other operating systems use different combinations
    - Windows and DOS use a carriage return followed by a line feed
    - Macintosh systems use only a carriage return
- Programs are available to convert between the various formats

6.2 Filtering Text and Piping

- The Unix philosophy: use small programs, and link them together as needed
- Each tool should be good at one specific job
- Join programs together with **pipes**
- Indicated with the pipe character: |
    - The first program prints text to its **standard output**
    - That gets fed into the second program's **standard input**
- For example, to connect the output of echo to the input of wc:

$ **echo "count these words, boy" | wc**

6.3 Displaying Files with less

- If a file is too long to fit in the terminal, display it with less:

$ **less README**

- less also makes it easy to clear the terminal of other things, so is useful even for small files
- Often used on the end of a pipe line, especially when it is not known how long the output will be:

$ **wc *.txt | less**

    - Doesn't choke on strange characters, so it won't mess up your terminal (unlike cat)

6.4 Counting Words and Lines with wc

- wc counts characters, words and lines in a file
- If used with multiple files, outputs counts for each file, and a combined total
- Options:
    - -c output character count
    - -l output line count
    - -w output word count

- Default is –clw

➢ Examples: display word count for *essay.txt*:

$ **wc -w essay.txt**

➢ Display the total number of lines in several text files:

$ **wc -l *.txt**

6.5 Sorting Lines of Text with sort

➢ The sort filter reads lines of text and prints them sorted into order

➢ For example, to sort a list of words into dictionary order:

$ **sort words > sorted-words**

➢ The -f option makes the sorting **case-insensitive**

➢ The -n option sorts numerically, rather than lexicographically

6.6 Removing Duplicate Lines with uniq

➢ Use uniq to find unique lines in a file

➢ Removes *consecutive* duplicate lines

➢ Usually give it sorted input, to remove all duplicates

- Example: find out how many unique words are in a dictionary:

$ **sort /usr/dict/words | uniq | wc –w**

➢ sort has a -u option to do this, without using a separate program:

$ **sort -u /usr/dict/words | wc -w**

➢ sort | uniq can do more than sort -u, though:

➢ uniq -c counts how many times each line appeared

➢ uniq -u prints only unique lines

➢ uniq -d prints only duplicated lines

6.7 Selecting Parts of Lines with cut

➢ Used to select columns or fields from each line of input

➢ Select a range of

- Characters, with
- Fields, with –f
- Field separator specified with -d (defaults to tab)

- A range is written as start and end position: e.g., 3-5
  - Either can be omitted
- The first character or field is numbered 1, not 0
  - Example: select usernames of logged in users:

$ **who | cut -d" " -f1 | sort -u**

6.8 Using fmt to Format Text Files

- Arranges words nicely into lines of consistent length
- Use -u to convert to uniform spacing
- One space between words, two between sentences
- Use -w *width* to set the maximum line width in characters
  - Defaults to 75
  - Example: change the line length of *notes.txt* to a maximum of 70 characters, and display it on the screen:

$ **fmt -w 70 notes.txt | less**

6.9 Reading the Start of a File with head

- Prints the top of its input, and discards the rest
- Set the number of lines to print with -n *lines* or *-lines*
- Defaults to ten lines
- Print the first line of a text file (two alternatives):

$ **head -n 1 notes.txt**

$ **head -1 notes.txt**

6.10 Reading the End of a File with tail

- Similar to head, but prints lines at the end of a file
- The -f option watches the file forever
- Continually updates the display as new entries are appended to the end of the file

- Kill it with Ctrl+C
- The option -n is the same as in head (number of lines to print)

6.11 Numbering Lines of a File with nl or cat

- Display the input with line numbers against each line
- There are options to finely control the formatting
- By default, blank lines aren't numbered
- The option -ba numbers every line
- cat -n also numbers lines, including blank ones

6.12 Dividing Files into Chunks with split

- Splits files into equal-sized segments
- Syntax: split [*options*] [*input*] [*output-prefix*]
- Use -l *n* to split a file into *n*-line chunks
- Use -b *n* to split into chunks of *n* bytes each
- Output files are named using the specified output name with *aa*, *ab*, *ac*, etc., added to the end of the prefix
    - Example: Split *essay.txt* into 30-line files, and save the output to files *short_aa*, *short_ab*, etc:

$ **split -l 30 essay.txt short_**

6.13 Reversing Files with tac

- Similar to cat, but in reverse
- Prints the last line of the input first, the penultimate line second, and so on
    - Example: show a list of logins and logouts, but with the most recent events at the end:

$ **last | tac**

6.14 Modifying Files with sed

- sed uses a simple script to process each line of a file
- Specify the script file with -f *filename*
- Or give individual commands with -e *command*
- For example, if you have a script called *spelling.sed* which corrects your most common mistakes, you can feed a file through it:

$ sed -f spelling.sed < report.txt > corrected.txt

6.15 Put Files Side-by-Side with paste

- paste takes lines from two or more files and puts them in columns of the output
  - Use -d *char* to set the delimiter between fields in the output
  - The default is tab
- Giving -d more than one character sets different delimiters between each pair of columns
- Example: assign passwords to users, separating them with a colon:

$ **paste -d: usernames passwords > .htpasswd**

6.16 Exercises

Q1

a. Type in the example on the cut slide to display a list of users logged in. (Try just who on its own first to see what is happening.)

b. Arrange for the list of usernames in who's output to be sorted, and remove any duplicates.

c. Try the command last to display a record of login sessions, and then try reversing it with tac. Which is more useful? What if you pipe the output into less?

d. Use sed to correct the misspelling 'enviroment' to 'environment'. Use it on a test file, containing a few lines of text, to check it. Does it work if the misspelling occurs more than once on the same line?

e. Use nl to number the lines in the output of the previous question.

Q2

a. Try making an empty file and using tail -f to monitor it. Then add lines to it from a different terminal using a command like this:

$ **echo "testing" >>filename**

b. Once you have written some lines into your file, use tr to display it with all occurances of the letters A–F changed to the numbers 0–5.

c. Try looking at the binary for the ls command (*/bin/ls*) with less. You can use the -f option to force it to display the file, even though it isn't text.

d. Try viewing the same binary with od. Try it in its default mode, as well as with the options shown on the slide for outputting in hexadecimal.

Q3

a. Use the split command to split the binary of the ls command into 1Kb chunks. You might want to create a directory especially for the split files, so that it can all be easily deleted later.

b. Put your split ls command back together again, and run it to make sure it still works. You will have to make sure you are running the new copy of it, for example ./my_ls, and make sure that the program is marked as 'executable' to run it, with the following command:

$ **chmod a+rx my_ls**

Lab No. 7

7.1.0- Lab objective

This lab will give overview of the use of Unix Streams, Pipes and Redirects.

7.1.1 Standard Files

- Processes are connected to three standard files
  - Standard output
  - Standard error
  - Standard input
- Many programs open other files as well

7.1.2 Standard Input

- Programs can read data from their **standard input** file
- Abbreviated to **stdin**
- By default, this reads from the keyboard
- Characters typed into an interactive program (e.g., a text editor) go to stdin

7.1.3 Standard Output

- Programs can write data to their **standard output** file
- Abbreviated to **stdout**
- Used for a program's normal output
- By default this is printed on the terminal

7.1.4 Standard Error

- Programs can write data to their **standard error** output
- Standard error is similar to standard output, but used for error and warning messages
- Abbreviated to **stderr**
- Useful to separate program output from any program errors
- By default this is written to your terminal
- So it gets 'mixed in' with the standard output

7.2 Pipes

- A **pipe** channels the output of one program to the input of another
- Allows programs to be chained together
- Programs in the chain run concurrently
- Use the vertical bar: |
- Sometimes known as the 'pipe' character
- Programs don't need to do anything special to use pipes
- They read from stdin and write to stdout as normal
- For example, pipe the output of echo into the program rev (which reverses each line of its input):

```
$ echo Happy Birthday! | rev
!yadhtriB yppaH
```

7.3.0 Connecting Programs to Files

- **Redirection** connects a program to a named file
- The < symbol indicates the file to read input from:

```
$ wc < thesis.txt
```

- The file specified becomes the program's standard input
- The > symbol indicates the file to write output to:

```
$ who > users.txt
```

- The program's standard output goes into the file
- If the file already exists, it is overwritten
- Both can be used at the same time:

$ **filter < input-file > output-file**

7.3.1 Appending to Files

- Use >> to append to a file:

$ **date >> log.txt**

- Appends the standard output of the program to the end of an existing file
- If the file doesn't already exist, it is created

7.3.2 Redirecting Multiple Files

- Open files have numbers, called **file descriptors**
- These can be used with redirection
- The three standard files always have the same numbers:
- **Name Descriptor**
    - Standard input 0
    - Standard output 1
    - Standard error 2

7.3.3 Redirection with File Descriptors

- Redirection normally works with stdin and stdout
- Specify different files by putting the file descriptor number before the redirection symbol:
- To redirect the standard error to a file:

$ **program 2> file**

- To combine standard error with standard output:

$ **program > file 2>&1**

- To save both output streams:

$ **program > stdout.txt 2> stderr.txt**

- The descriptors 3–9 can be connected to normal files, and are mainly used in shell scripts.

7.3.4 tee

- The tee program makes a 'T-junction' in a pipeline
- It copies data from stdin to stdout, and also to a file
- Like > and | combined
- For example, to save details of everyone's logins, and save Bob's logins in a separate file:

$ **last | tee everyone.txt | grep bob > bob.txt**

7.4 Exercises

Q1

a. Try the example on the 'Pipes' slide, using rev to reverse some text.

b. Try replacing the echo command with some other commands which produce output (e.g., whoami).

c. What happens when you replace rev with cat? You might like to try running cat with no arguments and entering some text.

Q2

a. Run the command ls --color in a directory with a few files and directories. Some Linux distributions have ls set up to always use the --color option in normal circumstances, but in this case we will give it explicitly.

b. Try running the same command, but pipe the output into another program (e.g., cat or less). You should spot two differences in the output. ls detects whether its output is going straight to a terminal (to be viewed by a human directly) or into a pipe (to be read by another program).

Lab No. 8

8.1.0- Lab objective

This lab will give overview the use of Regular Expressions for searching test files.

8.1.1 Searching Files with grep

- grep prints lines from files which match a pattern
- For example, to find an entry in the password file */etc/passwd* relating to the user 'nancy':

$ **grep nancy /etc/passwd**

- grep has a few useful options:
  - -i makes the matching case-insensitive
  - -r searches through files in specified directories, recursively
  - -l prints just the names of files which contain matching lines
  - -c prints the count of matches in each file
  - -n numbers the matching lines in the output
  - -v reverses the test, printing lines which don't match

8.1.2 Pattern Matching

- Use grep to find patterns, as well as simple strings
- Patterns are expressed as **regular expressions**
- Certain punctuation characters have special meanings
  - For example this might be a better way to search for Nancy's entry in the password file:
- $ **grep '^nancy' /etc/passwd**
- The caret (^) anchors the pattern to the start of the line
- In the same way, $ acts as an **anchor** when it appears at the end of a string, making the pattern match only at the end of a line

8.1.3 Matching Repeated Patterns

- Some regexp special characters are also special to the shell, and so need to be protected with quotes or backslashes
- We can match a repeating pattern by adding a modifier:

$ **grep -i 'continued\.*'**

- Dot (.) on its own would match any character, so to match an actual dot we escape it with \
- The * modifier matches the preceding character zero or more times
- Similarly, the \+ modifier matches one or more times

8.1.4 Matching Alternative Patterns

- Multiple subpatterns can be provided as alternatives, separated with \|, for example:

$ **grep 'fish\|chips\|pies' food.txt**

- The previous command finds lines which match at least one of the words
- Use \(...\) to enforce precedence:

$ **grep -i '\(cream\|fish\|birthday\) cakes' delicacies.txt**

- Use square brackets to build a **character class**:

$ **grep '[Jj]oe [Bb]loggs' staff.txt**

- Any single character from the class matches; and ranges of characters can be expressed as 'a-z'

8.2 Extended Regular Expression Syntax

- egrep runs grep in a different mode
- Same as grep -E
- Special characters don't have to be marked with \
- So \+ is written +, \(...\) is written (...), etc
- In extended regexps, \+ is a literal +

8.3 Exercises

Q1

a. Use grep to find information about the HTTP protocol in the file */etc/services*.

b. Usually this file contains some comments, starting with the '#' symbol. Use grep with the -v option to ignore lines starting with '#' and look at the rest of the file in less.

c. Add another use of grep -v to your pipeline to remove blank lines (which match the pattern ^$).

d. Use sed (also in the same pipeline) to remove the information after the '/' symbol on each line, leaving just the names of the protocols and their port numbers.

Lab No. 9

9.1.0- Lab objective

This lab will give overview of Job Control.

- Most shells offer **job control**
- The ability to stop, restart, and background a running process
- The shell lets you put & on the end of a command line to start it in the **background**
- Or you can hit Ctrl+Z to **suspend** a running foreground job
- Suspended and backgrounded jobs are given numbers by the shell
- These numbers can be given to shell job-control built-in commands
- Job-control commands include jobs, fg, and bg

9.2 jobs

- The jobs builtin prints a listing of active jobs and their job numbers:

$ jobs

[1]- Stopped vim index.html

[2] Running netscape &

3]+ Stopped man ls

- Job numbers are given in square brackets
    - But when you use them with other job-control builtins, you need to write them with percent signs, for example %1
    - The jobs marked + and - may be accessed as %+ or %- as well as by number
    - %+ is the shell's idea of the **current job** — the most recently active job
    - %- is the *previous* current job

9.3 fg

- Brings a backgrounded job into the foreground
- Re-starts a suspended job, running it in the foreground
    - fg %1 will foreground job number 1
    - fg with no arguments will operate on the current job

9.4 bg

- Re-starts a suspended job, running it in the background
- bg %1 will background job number 1
- bg with no arguments will operate on the current job
    - For example, after running gv and suspending it with Ctrl+Z, use bg to start it running again in the background.

9.5 Exercises

Q1

a. Start a process by running man bash and suspend it with Ctrl+Z.

- Run process in the background, using &.
- Use jobs to list the backgrounded and stopped processes.
- Use the fg command to bring man into the foreground, and quit from it as normal.
- Use fg to foreground the process, and terminate it with Ctrl+C.
- Run the process again, but this time without &. It should be running in the foreground (so you can't use the shell). Try suspending it with Ctrl+Z and see what happens. To properly put it into the background, use bg.

Lab No. 10

10.1.0- Lab objective

This lab will involve Creating, Monitoring, and Killing Processes.

10.1.1 What is a Process?

- The kernel considers each program running on your system to be a **process**
- A process 'lives' as it executes, with a lifetime that may be short or long
- A process is said to 'die' when it terminates
- The kernel identifies each process by a number known as a process id, or **pid**
- The kernel keeps track of various properties of each process

10.1.2 Process Properties

- A process has a user id (**uid**) and a group id (**gid**) which together specify what permissions it has
- A process has a parent process id (**ppid**) — the pid of the process which created it
- The kernel starts an init process with pid 1 at boot-up
- Every other process is a descendant of pid 1
- Each process has its own **working directory**, initially inherited from its parent process
- There is an **environment** for each process — a collection of named environment variables and their associated values
- A process's environment is normally inherited from its parent process

10.1.3 Parent and Child Processes

- The init process is the ancestor of all other processes:
  - init
  - bash
  - bash vi
  - apache
  - apache
  - apache
  - apache
- (Apache starts many child processes so that they can serve HTTP requests at the same time)

10.2.0 Process Monitoring: ps

- The ps command gives a snapshot of the processes running on a system at a given moment in time
- Very flexible in what it shows, and how:
- Normally shows a fairly brief summary of each process
- Normally shows only processes which are both owned by the current user and attached to a terminal
- Unfortunately, it doesn't use standard option syntax
  - Instead it uses a mixture of options with one of three syntaxes:
- Traditional BSD ps: a single letter *with no hyphen*
  - Unix98 ps: a single letter preceded by a hyphen
  - GNU: a word or phrase preceded by two hyphens

10.2.1 ps Options

- ps has many options
- Some of the most commonly used are:
- **Option Description**

- a Show processes owned by other users
- Display process ancestors in a tree-like format
- u Use the 'user' output format, showing user names and process start times
- w Use a wider output format. Normally each line of output is truncated; each use of the w option makes the 'window' wider
- x Include processes which have no controlling terminal
- -e Show information on *all* processes
- -l Use a 'long' output format
- -f Use a 'full' output format
- -C *cmd* Show only processes named *cmd*
- -U *user* Show only processes owned by *user*

10.2.2 Process Monitoring: pstree

- Displays a snapshot of running processes
- Always uses a tree-like display, like ps f
- But by default shows only the name of each command
- Normally shows all processes
- Specify a pid as an argument to show a specific process and its descendants
- Specify a user name as an argument to show process trees owned by that user

10.2.3 pstree Options

- **Option Description**
  - -a Display commands' arguments
  - -c Don't compact identical subtrees
  - -G Attempt to use terminal-specific line-drawing characters

- -h Highlight the ancestors of the current process
- -n Sort processes numerically by pid, rather than alphabetically by name
- -p Include pids in the output

10.3.0 Process Monitoring: top

- Shows full-screen, continuously-updated snapshots of process activity
  - Waits a short period of time between each snapshot to give the illusion of real-time monitoring
  - Processes are displayed in descending order of how much processor time they're using
  - Also displays system uptime, load average, CPU status, and memory information

10.3.1 top Command-Line Options

- **Option Description**
  - -b Batch mode — send snapshots to standard output
  - -n *num* Exit after displaying *num* snapshots
  - -d *delay* Wait *delay* seconds between each snapshot
  - -i Ignore idle processes
  - -s Disable interactive commands which could be dangerous if run by the superuser

10.3.2 top Interactive Commands

- **Key Behaviour**
  - q Quit the program
  - Ctrl+L Repaint the screen
  - h Show a help screen

- k Prompts for a pid and a signal, and sends that signal to that process
- n Prompts for the number of processes to show information; 0 (the default) means to show as many as will fit
- r Change the priority ('niceness') of a process
- s Change the number of seconds to delay between updates. The number may include fractions of a second (0.5, for example)

10.4.0 Signalling Processes

- A process can be sent a **signal** by the kernel or by another process
- Each signal is a very simple message:
- A small whole number
- With a mnemonic name
- Signal names are all-capitals, like INT
- They are often written with SIG as part of the name: SIGINT
- Some signals are treated specially by the kernel; others have a conventional meaning
- There are about 30 signals available, not all of which are very useful

10.4.1 Common Signals for Interactive Use

- The command kill -l lists all signals
- The following are the most commonly used:
- **Name Number Meaning**
  - INT 2 Interrupt — stop running. Sent by the kernel when you press Ctrl+C in a terminal.
  - TERM 15 "Please terminate." Used to ask a process to exit gracefully.

- KILL 9 "Die!" Forces the process to stop running; it is given no opportunity to clean up after itself.
- TSTP 18 Requests the process to stop itself temporarily. Sent by the kernel when you press Ctrl+Z in a terminal.

10.4.2 Sending Signals: kill

- The kill command is used to send a signal to a process
- It is a normal executable command, but many shells also provide it as a built-in
- You can specify more than one pid to signal all those processes

10.5 Exercises

Q1

a. Use top to show the processes running on your machine.

b. Make top sort by memory usage, so that the most memory-hungry processes appear at the top.

c. Restrict the display to show only processes owned by you.

d. Try killing one of your processes (make sure it's nothing important).

e. Display a list of all the processes running on the machine using ps (displaying the full command line for them).

f. Get the same listing as a tree, using both ps and pstree.

g. Have ps sort the output by system time used.

Lab No. 11

11.1.0- Lab objective

This lab will concentrate on Modifying Process Execution Priorities.

11.1 Concepts

- Not all tasks require the same amount of execution time
- Linux has the concept of **execution priority** to deal with this
- Process priority is dynamically altered by the kernel
- You can view the current priority by looking at top or ps -l and looking at the PRI column
- The priority can be biased using nice
- The current bias can be seen in the NI column in top

11.2.0 nice

- Starts a program with a given priority bias
- Peculiar name: 'nicer' processes require fewer resources
- Niceness ranges from +19 (very nice) to -20 (not very nice)
- Non-root users can only specify values from 1 to 19; the root user can specify the full range of values
- Default niceness when using nice is 10
- To run a command at increased niceness (lower priority):

$ nice -10 long-running-command &

$ nice -n 10 long-running-command &

- To run a command at decreased niceness (higher priority):

$ nice --15 important-command &

$ nice -n -15 important-command &

11.2.1 renice

- renice changes the niceness of existing processes
- Non-root users are only permitted to increase a process's niceness
- To set the process with pid 2984 to the maximum niceness (lower priority):

$ renice 20 2984

- The niceness is just a number: no extra - sign
- To set the process with pid 3598 to a lower niceness (higher priority):

$ renice -15 3598

- You can also change the niceness of all a user's processes:

$ renice 15 -u mikeb

11.3 Exercises

Q1

a. Create the following shell script, called forever, in your home directory:

```
#!/bin/sh
while [ 1 ]; do
echo hello... >/dev/null;
done
```

Make it executable and run it in the background as follows:

```
$ chmod a+rx forever
$ ./forever &
```

b. Use ps -l to check the script's nice level

c. Run the script with nice and give it a niceness of 15. Try running it alongside a less nice version, and see what the difference is in top

d. Try using nice or renice to make a process' niceness less than 0

Lab No. 12

12.1.0- Lab objective

This lab will concentrate on Managing File Ownership.

12.1.1 Users and Groups

- Anyone using a Linux computer is a **user**
- The system keeps track of different users, by username
- Security features allow different users to have different privileges
- Users can belong to **groups**, allowing security to be managed for collections of people with different requirements
- Use su to switch to a different user
- Quicker than logging off and back on again
- su prompts you for the user's password:

$ su - bob

Password:

  - The - option makes su behave as if you've logged in as that user

12.1.2 The Superuser: Root

- Every Linux system has a user called 'root'
- The root user is all-powerful
- Can access any files
- The root user account should only be used for system administration, such as installing software
- When logged in as root, the shell prompt usually ends in #
- Usually best to use su for working as root:

$ whoami

fred

$ su -

Password:

whoami

root

67

12.2 Changing File Ownership with chown

- The chown command changes the ownership of files or directories
- Simple usage:

 # chown aaronc logfile.txt

  - Makes *logfile.txt* be owned by the user aaronc

- Specify any number of files or directories
- Only the superuser can change the ownership of a file
- This is a security feature — quotas, set-uid

12.3.0 Changing File Group Ownership with chgrp

- The chgrp command changes the group ownership of files or directories
- Simple usage:

 # chgrp staff report.txt

  - Makes staff be the group owner of the file *logfile.txt*

- As for chown, specify any number of files or directories
- The superuser may change the group ownership of any file to any group
- The owner of a file may change its group ownership
- But only to a group of which the owner is a member

12.3.1 Changing the Ownership of a Directory and Its Contents

- A common requirement is to change the ownership of a directory and its contents
- Both chown and chgrp accept a -R option:

 # chgrp -R staff shared-directory

Mnemonic: 'recursive'

  - Changes the group ownership of *shared-directory* to staff And its contents And its subdirectories, recursively

- Changing user ownership (superuser only):

 # chown -R root /usr/local/share/misc/

12.3.2 Changing Ownership and Group Ownership Simultaneously

- The chown command can change the user-owner and group-owner of a file simultaneously:

 # chown aaronc:www-docs public_html/interesting.html

  - Changes the user owner to aaronc and the group owner to www-docs
  - Can use the -R option as normal
  - A dot (.) may be used instead of a colon:

 # chown -R aaronc.www-docs /www/intranet/people/aaronc/

12.4 Exercises

Q1

a. Find out who owns the file */bin/ls* and who owns your home directory (in */home*).

b. Log on as root, and create an empty file with touch. The user and group owners should be 'root' — check with ls.

c. Change the owner of the file to be 'users'.

d. Change the group owner to be any non-root user.

e. Change both of the owners back to being 'root' with a single command.

Lab No. 13

13.1.0- Lab objective

This lab will give insight to Use of File Permissions to Control Access to Files.

13.1.1 Basic Concepts: Permissions on Files

- Three types of permissions on files, each denoted by a letter
- A permission represents an action that can be done on the file:
- **Permission Letter Description**
    - Read r Permission to read the data stored in the file
    - Write w Permission to write new data to the file, to truncate the file, or to overwrite existing data
    - Execute x Permission to attempt to execute the contents of the file as a program
- Occasionally referred to as 'permission bits'
- Note that for scripts, you need both execute permission *and* read permission
    - The script interpreter (which runs with your permissions) needs to be able to read the script from the file

13.1.2 Basic Concepts: Permissions on Directories

- The r, w, x permissions also have a meaning for directories
- The meanings for directories are slightly different:
- **Permission Letter Description**
    - Read r Permission to get a listing of the directory
    - Write w Permission to create, delete, or rename files (or subdirectories) within the directory
    - Execute x Permission to change to the directory, or to use the directory as an intermediate part of a path to a file

- The difference between read and execute on directories is specious — having one but not the other is almost never what you want

13.1.3 Basic Concepts: Permissions for Different Groups of People

- As well as having different types of permission, we can apply different sets of permissions to different sets of people
- A file (or directory) has an **owner** and a **group owner**
- The r, w, x permissions are specified separately for the owner, for the group owner, and for everyone else (the 'world')

13.2 Examining Permissions: ls -l

- The ls -l command allows you to look at the permissions on a file:

$ ls -l

drwxr-x--- 9 aaronc staff 4096 Oct 12 12:57 accounts

-rw-rw-r-- 1 aaronc staff 11170 Dec 9 14:11 report.txt

  - The third and fourth columns are the owner and group-owner
  - The first column is the permissions:
  - One character for the file type: d for directories, - for plain files
  - Three characters of rwx permissions for the owner (or a dash if the permission isn't available)
  - Three characters of rwx permissions for the group owner
  - Three characters of rwx permissions for everyone else

13.3 Preserving Permissions When Copying Files

- By default, the cp command makes no attempt to preserve permissions (and other attributes like timestamps)
- You can use the -p option to preserve permissions and timestamps:

$ cp -p important.txt important.txt.orig

- Alternatively, the -a option preserves all information possible, including permissions and timestamps

13.4 How Permissions are Applied

- If you own a file, the per-owner permissions apply to you
- Otherwise, if you are in the group that group-owns the file, the per-group permissions apply to you
- If neither of those is the case, the for-everyone-else permissions apply to you

13.5.0 Changing File and Directory Permissions: chmod

- The chmod command changes the permissions of a file or directory
- A file's permissions may be changed only by its owner or by the superuser
- chmod takes an argument describing the new permissions
- Can be specified in many flexible (but correspondingly complex) ways
- Simple example:

$ chmod a+x new-program

  - adds (+) executable permission (x) for *a*ll users (a) on the file new-program

13.5.1 Specifying Permissions for chmod

- Permissions can be set using letters in the following format:
  - [ugoa][+=-][rwxX]
- The first letters indicate who to set permissions for:
  - u for the file's owner, g for the group owner, o for other users, or a for all users

- = sets permissions for files, + adds permissions to those already set, and – removes permissions
- The final letters indicate which of the r, w, x permissions to set
- Or use capital X to set the x permission, but only for directories and already-executable files

13.5.2 Changing the Permissions of a Directory and Its Contents

- A common requirement is to change the permissions of a directory and its contents
- chmod accepts a -R option:

$ chmod -R g+rwX,o+rX public-directory

Mnemonic: 'recursive'

- Adds rwx permissions on *public-directory* for the group owner, and adds rx permissions on it for everyone else
- And any subdirectories, recursively
- Any any contained executable files
- Contained non-executable files have rw permissions added for the group owner, and r permission for everyone else

13.6 Special Directory Permissions: 'Sticky'

- The */tmp* directory must be world-writable, so that anyone may create temporary files within it
- But that would normally mean that anyone may delete *any* files within it — obviously a security hole
- A directory may have 'sticky' permission:
- Only a file's owner may delete it from a sticky directory
- Expressed with a t (mnemonic: *t*emporary directory) in a listing:

$ ls -l -d /tmp

drwxrwxrwt 30 root root 11264 Dec 21 09:35 /tmp

- Enable 'sticky' permission with:

 # chmod +t /data/tmp

13.7 Special Directory Permissions: Setgid

- If a directory is **setgid** ('set group-id'), files created within it acquire the group ownership of the directory
- And directories created within it acquire both the group ownership *and* setgid permission
- Useful for a shared directory where all users working on its files are in a given group
- Expressed with an s in 'group' position in a listing:

$ ls -l -d /data/projects

drwxrwsr-x 16 root staff 4096 Oct 19 13:14 /data/projects

- Enable setgid with:

 # chmod g+s /data/projects

13.8 Special File Permissions: Setgid

- Setgid permission may also be applied to executable files
- A process run from a setgid file acquires the group id of the file
- Note: Linux doesn't directly allow scripts to be setgid — only compiled programs
- Useful if you want a program to be able to (for example) edit some files that have a given group owner
- Without letting individual users access those files directly

13.9 Special File Permissions: Setuid

- Files may also have a **setuid** ('set user-id') permission
- Equivalent to setgid: a process run from a setuid file acquires the user id of the file

- As with setgid, Linux doesn't allow scripts to be setuid
- Expressed with an s in 'user' position in a listing:

$ ls -l /usr/bin/passwd

-r-s--x--x 1 root root 12244 Feb 7 2000 /usr/bin/passwd

- Enable setuid with:

 # chmod u+s /usr/local/bin/program

13.10 Displaying Unusual Permissions

- Use ls -l to display file permissions
- Setuid and Setgid permissions are shown by an s in the user and group execute positions
- The sticky bit is shown by a t in the 'other' execute position
- The letters s and t cover up the execute bits
- But you can still tell whether the execute bits are set
    - Lowercase s or t indicates that execute is enabled (i.e., there is an x behind the letter)
    - Uppercase S or T indicates that execute is disabled (there is a - behind the letter)

13.11 Exercises

Q1

a. Find out what permissions are set on your home directory (as a normal user). Can other users access files inside it?

b. If your home directory is only accessible to you, then change the permissions to allow other people to read files inside it, otherwise change it so that they can't.

c. Check the permissions on */bin* and */bin/ls* and satisfy yourself that they are reasonable.

d. Check the permissions available on */etc/passwd* and */etc/shadow*.

e. Write one command which would allow people to browse through your home directory and any subdirectories inside it and read all the files.

Lab No. 14

14.1.0- Lab objective

This lab will give an overview of The Shell as a Programming Language

14.1.1- Background

Shells

- A **shell** provides an interface between the user and the operating system kernel
- Either a **command interpreter** or a graphical user interface
- Traditional Unix shells are **command-line interfaces** (CLIs)
- Usually started automatically when you log in or open a terminal
- There are two ways of writing shell programs.
    - You can type a sequence of commands and allow the shell to execute them interactively.
    - Or you can store those commands in a file that you can then invoke as a program.

14.1.2- Creating a Script

- Using any text editor, you need to create a file containing the commands; create a file called first that looks like this:

```
#!/bin/sh
# first
# This file looks through all the files in the current
# directory for the string POSIX, and then prints the names of
# those files to the standard output.
for file in *
do
if grep -q POSIX $file
```

```
then
echo $file
fi
done
exit 0
```

- Comments start with a # and continue to the end of a line.
- Conventionally, though, # is kept in the first column.
- The first line, #!/bin/sh, is a special form of comment;
- The #! characters tell the system that the argument that follows on the line is the program to be used to execute this file.
- In this case, /bin/sh is the default shell program.
- The exit command ensures that the script returns a sensible exit code (more on this later).
- This is rarely checked when programs are run interactively, but if you want to invoke this script from another script and check whether it succeeded, returning an appropriate exit code is very important.
- Even if you never intend to allow your script to be invoked from another, you should still exit with a reasonable code.
- A zero denotes success in shell programming. Since the script as it stands can't detect any failures, it always returns success.
- Notice that this script does not use any filename extension or suffix; Linux, and UNIX in general, rarely makes use of the filename extension to determine the type of a file.
- You could have used .sh or added a different extension, but the shell doesn't care. Most preinstalled scripts will not have any filename extension,

14.1.1 Making a Script Executable

- The simpler way is to invoke the shell with the name of the script file as a parameter:

$ /bin/sh first

- Do this by changing the file mode to make the file executable for all users using the chmod command:

$ chmod +x first

or

chmod u=rwx,go=rx /usr/local/bin/first

- Note that for scripts, you need both execute permission *and* read permission
    - The script interpreter (which runs with your permissions) needs to be able to read the script from the file

14.2.0 Shell Syntax

- You can use the bash shell to write quite large, structured programs. The next few sections cover the following:
- Variables: strings, numbers, environments, and parameters
- Conditions: shell Booleans
- Program control: if, elif, for, while, until, case
- Lists
- Functions

14.2.1 Variables

- You don't usually declare variables in the shell before using them.
- Instead, you create them by simply using them (for example, when you assign an initial value to them). By default, all variables are considered and stored as strings, even when they are assigned numeric values.
- The shell and some utilities will convert numeric strings to their values in order to operate on them as required.
- Linux is a case-sensitivesystem, so the shell considers the variable foo to be different from Foo, and both to be different from FOO.

14.2.2 Access the contents of a variable

- By Within the shell you can access the contents of a variable by preceding its name with a $.
- Whenever you extract the contents of a variable, you must give the variable a preceding $.
- When you assign a value to a variable, just use the name of the variable, which is created dynamically if necessary.
- An easy way to check the contents of a variable is to echo it to the terminal, preceding its name with a $.
- On the command line, you can see this in action when you set and check various values of the variable salutation:

$ **salutation=Hello**

$ **echo $salutation**

Hello

$ **salutation="Yes Dear"**

$ **echo $salutation**

Yes Dear

$ **salutation=7+5**

$ **echo $salutation**

7+5

- Note how a string must be delimited by quote marks if it contains spaces. In addition, there can't be any spaces on either side of the equals sign.

14.2.3 Assign user input to a variable

- You can assign user input to a variable by using the **read** command.
- This takes one parameter, the name of the variable to be read into, and then waits for the user to enter some text.
- The read normally completes when the user presses Enter. When reading a variable from the terminal, you don't usually need the quote marks:

$ read salutation

Wie geht's?

$ echo $salutation

Wie geht's?

14.2.4 Quoting

- Normally, parameters in scripts are separated by whitespace characters (e.g., a space, a tab, or a newline character).
- If you want a parameter to contain one or more whitespace characters, you must quote the parameter.
- The behavior of variables such as **$foo** inside quotes depends on the type of quotes you use.

- If you enclose a $ variable expression in double quotes, then it's replaced with its value when the line is executed.
- If you enclose it in single quotes, then no substitution takes place. You can also remove the special meaning of the $ symbol by prefacing it with a \.
- Usually, strings are enclosed in double quotes, which protects variables from being separated by white space but allows $ expansion to take place.

```
#!/bin/sh
myvar="Hi there"
echo $myvar
echo "$myvar"
echo '$myvar'
echo \$myvar
echo Enter some text
read myvar
echo '$myvar' now equals $myvar
exit 0
```

This behaves as follows:

```
$ ./variable
Hi there
Hi there
$myvar
$myvar
Enter some text
Hello World
$myvar now equals Hello World
```

14.2.5 How It Works

- The variable myvar is created and assigned the string Hi there.
- The contents of the variable are displayed with the echo command, showing how prefacing the variable with a $ character expands the contents of the variable.
- You see that using double quotes doesn't affect the substitution of the variable, while single quotes and the backslash do.
- You also use the read command to get a string from the user.

14.2.6 Environment Variables

- When a shell script starts, some variables are initialized from values in the environment.
- These are normally in all uppercase form to distinguish them from user-defined (shell) variables in scripts, which are conventionally lowercase.
- The variables created depend on your personal configuration. Many are listed in the manual pages, but the principal ones are listed in the following table:

Environment Variable	Description
`$HOME`	The home directory of the current user
`$PATH`	A colon-separated list of directories to search for commands
`$PS1`	A command prompt, frequently `$`, but in bash you can use some more complex values; for example, the string `[\u@\h \W]$` is a popular default that tells you the user, machine name, and current directory, as well as providing a $ prompt.
`$PS2`	A secondary prompt, used when prompting for additional input; usually >.
`$IFS`	An input field separator. This is a list of characters that are used to separate words when the shell is reading input, usually space, tab, and newline characters.
`$0`	The name of the shell script
`$#`	The number of parameters passed
`$$`	The process ID of the shell script, often used inside a script for generating unique temporary filenames; for example `/tmp/tmpfile_$$`

14.2.7 Parameter Variables

- If your script is invoked with parameters, some additional variables are created. If no parameters are passed, the environment variable $# still exists but has a value of 0.
- The parameter variables are listed in the following table:

Parameter Variable	Description
$1, $2, …	The parameters given to the script
$*	A list of all the parameters, in a single variable, separated by the first character in the environment variable IFS. If IFS is modified, then the way $* separates the command line into parameters will change.
$@	A subtle variation on $*; it doesn't use the IFS environment variable, so parameters are not run together even if IFS is empty.

It's easy to see the difference between $@ and $* by trying them out:

$ **IFS=''**

$ **set foo bar bam**

$ **echo "$@"**

foo bar bam

$ **echo "$*"**

foobarbam

$ **unset IFS**

$ **echo "$*"**

foo bar bam

- Within double quotes, $@ expands the positional parameters as separate fields, regardless of the IFS value.

- In general, if you want access to the parameters, $@ is the sensible choice.
- In addition to printing the contents of variables using the echo command, you can also read them by using the read command.

14.2.8 Manipulating Parameter and Environment Variables

- The following script demonstrates some simple variable manipulation.

```
#!/bin/sh
salutation="Hello"
echo $salutation
echo "The program $0 is now running"
echo "The second parameter was $2"
echo "The first parameter was $1"
echo "The parameter list was $*"
echo "The user's home directory is $HOME"
echo "Please enter a new greeting"
read salutation
echo $salutation
echo "The script is now complete"
exit 0
```

- If you run this script, you get the following output:

```
$ ./try_var foo bar baz
Hello
The program ./try_var is now running
The second parameter was bar
The first parameter was foo
The parameter list was foo bar baz
The user's home directory is /home/rick
```

Please enter a new greeting

Sire

Sire

The script is now complete

$

14.3.0 Conditions

- Fundamental to all programming languages is the ability to test conditions and perform different actions based on those decisions.
- A shell script can test the exit code of any command that can be invoked from the command line, including the scripts that you have written yourself. That's why it's important to always include an exit command with a value at the end of any scripts that you write.

14.3.1 The test or [Command

- In practice, most scripts make extensive use of the [or test command, the shell's Boolean check.
- On some systems, the [and test commands are synonymous, except that when the [command is used, a trailing] is also used for readability.
- We'll introduce the test command using one of the simplest conditions: checking to see whether exists. The command for this is test -f <filename>, so within a script you can write

if test -f fred.c

then

...

fi

- You can also write it like this:

```
if [ -f fred.c ]
then
...
fi
```

- The test command's exit code (whether the condition is satisfied) determines whether the conditional code is run.
- Note that you must put spaces between the [braces and the condition being checked.
- You can remember this by remembering that [is just the same as writing test, and you would always leave a space after the test command.
- If you prefer putting then on the same line as if, you must add a semicolon to separate the test from the then :

```
if [ -f fred.c ]; then
...
fi
```

- The condition types that you can use with the test command fall into three types: string comparison, arithmetic comparison, and file conditionals.
- The following table describes these condition types:

String Comparison	Result
`string1 = string2`	True if the strings are equal
`string1 != string2`	True if the strings are not equal
`-n string`	True if the string is not null
`-z string`	True if the string is null (an empty string)

Arithmetic Comparison	Result
`expression1 -eq expression2`	True if the expressions are equal
`expression1 -ne expression2`	True if the expressions are not equal
`expression1 -gt expression2`	True if `expression1` is greater than `expression2`
`expression1 -ge expression2`	True if `expression1` is greater than or equal to `expression2`
`expression1 -lt expression2`	True if `expression1` is less than `expression2`
`expression1 -le expression2`	True if `expression1` is less than or equal to `expression2`

File Conditional	Result
`-d file`	True if the file is a directory
`-e file`	True if the file exists. Note that historically the `-e` option has not been portable, so `-f` is usually used.
`-f file`	True if the file is a regular file
`-g file`	True if `set-group-id` is set on `file`
`-r file`	True if the file is readable
`-s file`	True if the file has nonzero size
`-u file`	True if `set-user-id` is set on `file`
`-w file`	True if the file is writable
`-x file`	True if the file is executable

- Following is an example of how you would test the state of the file /bin/bash, just so you can see what these look like in use:

```
#!/bin/sh
if [ -f /bin/bash ]
then
echo "file /bin/bash exists"
fi
if [ -d /bin/bash ]
```

```
then
echo "/bin/bash is a directory"
else
echo "/bin/bash is NOT a directory"
fi
```

14.4.0 Control Structures

- The shell has a set of control structures, which are very similar to other programming languages.

14.4.1 if

- The if statement is very simple: It tests the result of a command and then conditionally executes a group of statements:

```
if condition
then
    statements
else
    statements
fi
```

- A common use for if is to ask a question and then make a decision based on the answer:

```
#!/bin/sh
echo "Is it morning? Please answer yes or no"
read timeofday
if [ $timeofday = "yes" ]; then
    echo "Good morning"
```

```
else
    echo “Good afternoon”
fi
exit 0
```

- This would give the following output:

```
Is it morning? Please answer yes or no
yes
Good morning
$
```

14.4.2 elif

- It allows you to add a second condition to be checked when the else portion of the if is executed.
- You can modify the previous script so that it reports an error message if the user types in anything other than yes or no. Do this by replacing the else with elif and then adding another condition:

```
#!/bin/sh
echo “Is it morning? Please answer yes or no”
read timeofday
if [ $timeofday = “yes” ]
then
echo “Good morning”
elif [ $timeofday = “no” ]; then
echo “Good afternoon”
else
echo “Sorry, $timeofday not recognized. Enter yes or no”
exit 1
fi
```

exit 0

14.4.3 A Problem with Variables

- This fixes the most obvious defect, but a more subtle problem is lurking. Try this new script, but just press Enter (or Return on some keyboards), rather than answering the question. You'll get this error message:

[: =: unary operator expected

- The problem is in the first if clause.
- When the variable timeofday was tested, it consisted of a blank string. Therefore, the if clause looks like

if [= "yes"]

- which isn't a valid condition. To avoid this, you must use quotes around the variable:

if ["$timeofday" = "yes"]

An empty variable then gives the valid test:

if ["" = "yes"]

The new script is as follows:

```
#!/bin/sh
echo "Is it morning? Please answer yes or no"
read timeofday
if [ "$timeofday" = "yes" ]
then
    echo "Good morning"
elif [ "$timeofday" = "no" ]; then
    echo "Good afternoon"
else
    echo "Sorry, $timeofday not recognized. Enter yes or no"
exit 1
```

```
fi
exit 0
```

14.4.4 for

- Use the for construct to loop through a range of values, which can be any set of strings.
- They could be simply listed in the program or, more commonly, the result of a shell expansion of filenames.
- The syntax is simple:

```
for variable in values
do
    statements
done
```

14.4.4 Using a for Loop with Fixed Strings

- The values are normally strings, so you can write the following:

```
#!/bin/sh
for foo in bar fud 43
do
    echo $foo
done
exit 0
```

- That results in the following output:

```
bar
fud
43
```

14.4.5 How It Works

- This example creates the variable foo and assigns it a different value each time around the for loop.
- Since the shell considers all variables to contain strings by default, it's just as valid to use the string 43 as the string fud.

14.4.6 Using a for Loop with Wildcard Expansion

- it's common to use the for loop with a shell expansion for filenames.
- This means using a wildcard for the string value and letting the shell fill out all the values at run time.
- Imagine that you want to print all the script files starting with the letter "f" in the current directory, and you know that all your scripts end in .sh.
- You could do it like this:

```
#!/bin/sh
for file in $(ls f*.sh); do
    lpr $file
done
exit 0
```

14.4.7 while

- Because all shell values are considered strings by default, the for loop is good for looping through a series of strings, but is not so useful when you don't know in advance how many times you want the loop to be executed.
- When you need to repeat a sequence of commands, but don't know in advance how many times they should execute, you will normally use a **while loop**, which has the following syntax:

```
while condition do
    statements
done
```

- For example, here is a rather poor password-checking program:

```
#!/bin/sh
echo "Enter password"
read trythis
while [ "$trythis" != "secret" ]; do
    echo "Sorry, try again"
    read trythis
done
exit 0
```

- An example of the output from this script is as follows:

```
Enter password
password
Sorry, try again
secret
$
```

- Clearly, this isn't a very secure way of asking for a password, but it does serve to illustrate the while statement.
- The statements between do and done are continuously executed until the condition is no longer true. In this case, you're checking whether the value of trythis is equal to secret.

- The loop will continue until $trythis equals secret. You then continue executing the script at the statement immediately following the done.
- This means using a wildcard for the string value and letting the shell fill out all the values at run time.

14.4.8 until

- The until statement has the following syntax:

until condition

do

statements

done

- This is very similar to the while loop, but with the condition test reversed. In other words, the loop continues until the condition becomes true, not while the condition is true.

14.5.0 case

- The case construct is a little more complex than those you have encountered so far. Its syntax is as follows:

case variable in

pattern [**|** pattern] **...)** statements**;;**

pattern [**|** pattern] **...)** statements**;;**

...

Esac

- This may look a little intimidating, but the case construct enables you to match the contents of a variable against patterns in quite a

sophisticated way and then allows execution of different statements, depending on which pattern was matched.

- It is much simpler than the alternative way of checking several conditions, which would be to use multiple if, elif, and else statements.
- Notice that each pattern line is terminated with double semicolons (;;). You can put multiple statements between each pattern and the next, so a double semicolon is needed to mark where one statement ends and the next pattern begins.
- The capability to match multiple patterns and then execute multiple related statements makes the case construct a good way of dealing with user input.
- The best way to see how case works is with an example.

14.5.0 Case I : User Input

- You can write a new version of the input-testing script and, using the case construct, make it a little more selective and forgiving of unexpected input:

```
#!/bin/sh
echo “Is it morning? Please answer yes or no”
read timeofday
case “$timeofday” in
  yes) echo “Good Morning”;;
  no ) echo “Good Afternoon”;;
   y ) echo “Good Morning”;;
   n ) echo “Good Afternoon”;;
   * ) echo “Sorry, answer not recognized”;;
esac
```

```
exit 0
```

14.5.2 How It Works

- When the case statement is executing, it takes the contents of timeofday and compares it to each string in turn.
- As soon as a string matches the input, the case command executes the code following the) and finishes.
- The case command performs normal expansion on the strings that it's using for comparison. You can therefore specify part of a string followed by the * wildcard. Using a single * will match all possible strings, so always put one after the other matching strings to ensure that the case statement ends with some default action if no other strings are matched.
- This is possible because the case statement compares against each string in turn. It doesn't look for a best match, just the first match.
- The default condition often turns out to be the impossible condition, so using * can help in debugging scripts.

14.5.3 Case II : Putting Patterns Together

- The preceding case construction is clearly more elegant than the multiple if statement version, but by putting the patterns together, you can make a much cleaner version:

```
#!/bin/sh
echo "Is it morning? Please answer yes or no"
read timeofday
case "$timeofday" in
  yes | y | Yes | YES ) echo "Good Morning";;
 n* | N* ) echo "Good Afternoon";;
  * ) echo "Sorry, answer not recognized";;
```

```
Esac
exit 0
```

14.5.4 Case I I I : Executing Multiple Statements

- Finally, to make the script reusable, you need to have a different exit value when the default pattern is used because the input was not understood:

```
#!/bin/sh
echo “Is it morning? Please answer yes or no”
read timeofday
case “$timeofday” in
    yes | y | Yes | YES )
            echo “Good Morning”
            echo “Up bright and early this morning”
            ;;
    [nN]*)
            echo “Good Afternoon”
            ;;
    *)
            echo “Sorry, answer not recognized”
            echo “Please answer yes or no”
            exit 1
            ;;
esac
exit 0
```

14.6.0 Lists

- Sometimes you want to connect commands in a series.
- For instance, you may want several different conditions to be met before you execute a statement:

```
if [ -f this_file ]; then
    if [ -f that_file ]; then
        if [ -f the_other_file ]; then
            echo "All files present, and correct"
            fi
        fi
    fi
```

- Although these can be implemented using multiple if statements, you can see that the results are awkward.
- The shell has a special pair of constructs for dealing with lists of commands: the AND list and the OR list.
- These are often used together, but we'll review their syntax separately.

14.6.1 The AND List

- The AND list construct enables you to execute a series of commands, executing the next command only if all the previous commands have succeeded.
- The syntax is

statement1 **&&** statement2 **&&** statement3 **&& ...**

- Starting at the left, each statement is executed; if it returns true, the next statement to the right is executed.
- This continues until a statement returns false, after which no more statements in the list are executed.

- The && tests the condition of the preceding command.
- Each statement is executed independently, enabling you to mix many different commands in a single list, as the following script shows. The AND list as a whole succeeds if all commands are executed successfully, but it fails otherwise.

```
#!/bin/sh
touch file_one
rm -f file_two
if [ -f file_one ] && echo "hello" && [ -f file_two ] && echo " there"
then
    echo "in if"
else
    echo "in else"
fi
exit 0
```

- Try the script and you'll get the following result:

```
hello
in else
```

14.6.2 The OR List

- The OR list construct enables us to execute a series of commands until one succeeds, and then not execute any more.
- The syntax is as follows:

statement1 **||** statement2 **||** statement3 **|| ...**

- Starting at the left, each statement is executed. If it returns false, then the next statement to the right is executed.
- This continues until a statement returns true, at which point no more statements are executed.

- The || list is very similar to the && list, except that the rule for executing the next statement is that the previous statement must fail.

```
#!/bin/sh
rm -f file_one
if [ -f file_one ] || echo “hello” || echo “ there”
then
    echo “in if”
else
    echo “in else”
fi
exit 0
```

- This results in the following output:

hello
in if

14.7.0 Functions

- You can define functions in the shell; and if you write shell scripts of any size, you'll want to use them to structure your code.
- To define a shell function, simply write its name followed by empty parentheses and enclose the statements in braces:

```
function_name () {
statements
}
```

- Let's look at a really simple function:

```
#!/bin/sh
foo() {
    echo “Function foo is executing”
}
```

```
echo “script starting”
foo
echo “script ended”
exit 0
```

- Running the script will output the following:

```
script starting
Function foo is executing
script ending
```

14.7.1 Returning a Value

- The next script, my_name, shows how parameters to a function are passed and how functions can return a true or false result.
- You call this script with a parameter of the name you want to use in the question.
- After the shell header, define the function yes_or_no:

```
#!/bin/sh
yes_or_no() {
echo “Is your name $* ?”
while true
do
    echo -n “Enter yes or no: “
    read x
case “$x” in
    y | yes ) return 0;;
    n | no ) return 1;;
    * ) echo “Answer yes or no”
esac
done
}
```

- Then the main part of the program begins:

```
echo "Original parameters are $*"
if yes_or_no "$1"
then
        echo "Hi $1, nice name"
else
        echo "Never mind"
fi
exit 0
```

- **Typical output from this script might be as follows:**

```
$ ./my_name Rick Neil
Original parameters are Rick Neil
Is your name Rick ?
Enter yes or no: yes
Hi Rick, nice name
$
```

Advance Topics

Lab No. 15(OPTIONAL TOPICS)

15.1.0- Lab objective

This lab will guide students on how to Create Partitions and Filesystems on a LINUX environment.

15.1.1 Concepts: Disks and Partitions

- A hard disk provides a single large storage space
- Usually split into **partitions**
- Information about partitions is stored in the **partition table**
- Linux defaults to using partition tables compatible with *Microsoft Windows*
- For compatibility with *Windows*, at most four primary partitions can be made But they can be **extended partitions**, which can themselves be split into smaller **logical partitions**
- Extended partitions have their own partition table to store information about logical partitions

15.2 Disk Naming

- The device files for IDE hard drives are */dev/hda* to */dev/hdd*
- *hda* and *hdb* are the drives on the first IDE channel, *hdc* and *hdd* the ones on the second channel
- The first drive on each channel is the IDE 'master', and the second is the IDE 'slave'
    - Primary partitions are numbered from 1–4

- Logical partitions are numbered from 5
- The devices */dev/hda*, etc., refer to whole hard disks, not partitions
- Add the partition number to refer to a specific partition
- For example, */dev/hda1* is the first partition on the first IDE disk
- SCSI disks are named */dev/sda*, */dev/sdb*, etc

15.3 Using fdisk

- The fdisk command is used to create, delete and change the partitions on a disk
- Give fdisk the name of the disk to edit, for example:

 # fdisk /dev/hda

- fdisk reads one-letter commands from the user
- Type m to get a list of commands
- Use p to show what partitions currently exist
- Use q to quit without altering anything
- Use w to quit and write the changes
- Use with caution, and triple-check what you're doing!

15.4 Making New Partitions

- Create new partitions with the n command
- Choose whether to make a primary, extended or logical partition
- Choose which number to assign it
- fdisk asks where to put the start and end of the partition
- The default values make the partition as big as possible
- The desired size can be specified in megabytes, e.g., +250M
- Changes to the partition table are only written when the w command is given

15.5 Changing Partition Types

- Each partition has a type code, which is a number
- The fdisk command l shows a list of known types
- The command t changes the type of an existing partition
- Enter the type code at the prompt
- Linux partitions are usually of type 'Linux native' (type 83)
- Other operating systems might use other types of partition, many of which can be understood by Linux

15.6 Making Filesystems with mkfs

- The mkfs command initializes a filesystem on a new partition
    - Warning: any old data on the partition will be lost
    - For example, to make an ext2 filesystem on /dev/hda2:

 # mkfs -t ext2 -c /dev/hda2

    - -t sets the filesystem type to make, and -c checks for bad blocks on the disk
- mkfs uses other programs to make specific types of filesystem, such as mke2fs and mkdosfs

Lab No. 16(OPTIONAL TOPICS)

16.1.0- Lab objective

This lab will teach students to Control Filesystem Mounting and Unmounting

16.1.1 Mounting Filesystems

- As far as many parts of a Linux system are concerned, a partition contains entirely arbitrary data
- When installing, you set things up so that a partition contains a filesystem — a way of organising data into files and directories
- One filesystem is made the **root filesystem**: the root directory on that filesystem becomes the directory named /
- Other filesystems can be **mounted**: the root directory of that filesystem is grafted onto a directory of the root filesystem
- This arranges for every file in every mounted filesystem to be accessible from a single unified name space
- The directory grafted onto is called the **mount point**

16.1.2 Mounting a Filesystem: mount

- 'Important' filesystems are mounted at boot-up; other filesystems can be mounted or unmounted at any time
- The mount command mounts a filesystem
- You usually need to have root permission to mount a filesystem

- mount makes it easy to mount filesystems configured by the system administrator
- For example, many systems are configured so that

$ mount /mnt/cdrom

  - will mount the contents of the machine's CD-ROM drive under the directory */mnt/cdrom*
  - Linux System Administration Module 17. Control Filesystem Mounting and Unmounting

16.1.3 Mounting Other Filesystems

- mount /dev/sdb3 /mnt/extra mounts the filesystem stored in the /dev/sdb3 device on the mount point */mnt/extra*
- You may occasionally need to specify the filesystem type explicitly:

 # mount -t vfat /dev/hdd1 /mnt/windows

- Allowable filesystem types are listed in the mount(8) manpage
- To see a list of the filesystems currently mounted, run mount without any options

16.1.4 Unmounting a Filesystem: umount

- A filesystem can be unmounted with umount
- Note the spelling!
- umount /mnt/extra unmounts whatever is on the */mnt/extra* mount point
- umount /dev/sdb3 unmounts the filesystem in the /dev/sdb3 device, wherever it is mounted
- You normally need to have root permission to unmount a filesystem
- It's also impossible to unmount a 'busy' filesystem
- A filesystem is busy if a process has a file on it open
- Or if a process has a directory within it as its current directory

16.1.5 Configuring mount: */etc/fstab*

- The */etc/fstab* file contains information about filesystems that are known to the system administrator
- Specifying a filesystem in */etc/fstab* makes it possible to use its mount point as the only argument to mount
- */etc/fstab* also configures which filesystems should be mounted at boot-up
- Each line in */etc/fstab* describes one filesystem
- Six columns on each line

16.1.6 Sample */etc/fstab*

- A sample */etc/fstab* file:

```
# device mount-point type options (dump) pass-no
/dev/hda3 / ext2 defaults 1 1
/dev/hda1 /boot ext2 defaults 1 2
/dev/hda5 /usr ext2 defaults 1 2
/dev/hdb1 /usr/local ext2 defaults 1 2
/dev/hdb2 /home ext2 defaults 1 2
none /proc proc defaults 0 0
/dev/scd0 /mnt/cdrom iso9660 noauto,users,ro 0 0
/dev/fd0 /mnt/floppy auto noauto,users 0 0
```

16.2.0 Filesystem Types

- The most common filesystem types are:
- **Type Usage**
    - ext2 The standard Linux filesystem
    - Iso9660 The filesystem used on CD-ROMs

- proc Not a real filesystem, so uses none as the device. Used as a way
- for the kernel to report system information to user processes
- vfat The filesystem used by Windows 95
- auto Not a real filesystem type. Used as a way of asking the mount command to probe for various filesystem types, particularly for removable media

- Networked filesystems include nfs (Unix-specific) and smbfs (Windows or Samba).Other, less common types exist; see mount(8)

16.2.1 Mount Options

- Comma-separated options in */etc/fstab*
- Alternatively, use comma-separated options with -o on the mount command line
- Common mount options:
- **Option Description**
  - noauto In */etc/fstab*, prevents the filesystem being mounted at bootup.
  - Useful for removable media
  - ro Mount the filesystem read-only
  - users Let non-root users mount and unmount this filesystem
  - user Like users, but non-root users can only unmount filesystems that they themselves mounted
- Other less common mount options exist, as well as many options for individual filesystem types

16.2.2 Other Columns in */etc/fstab*

- The fifth column is called dump
- Used by the dump and restore backup utilities

- Few people use those tools
- Just use 1 for normal filesystems, and 0 for removable filesystems
- The sixth column is called pass-no
- Controls the order in which automatically-mounted filesystems are checked by fsck
- Use 1 for the root filesystem
- Use 0 for filesystems that aren't mounted at boot-up
- Use 2 for other filesystems

17.2.3 Mounting a File

- Using **loop devices**, Linux can mount a filesystem stored in a normal file, instead of a disk
- Useful for testing images of CD-ROMs before burning them to disk
- For example, to create a filesystem of roughly floppy-disk size:

 # dd if=/dev/zero of=disk.img bs=1024 count=1400

 # mke2fs -F disk.img

- To mount the file so that its contents is accessible through */mnt/disk*:

 # mount -o loop disk.img /mnt/disk

17.11 Exercises

Q1

a. Use mount to find out which filesystems are mounted.

b. Check the */etc/fstab* file to see whether the floppy drive is configured properly, and find out what its mount point is set to.

c. Mount a floppy disk at the default mount point.

d. Copy a file onto the floppy disk. Does Linux write it immediately? Unmount the floppy to ensure that everything on it is properly written, and it is safe to remove.

e. Try the commands on the last slide to mount a file, and try copying some files into it. Try using the df command to see how much space is available in the file. Unmount */mnt/disk* as you would any other filesystem.

Lab No. 17(OPTIONAL TOPICS)

17.1.0- Lab objective

This lab will teach students to Maintain the Integrity of Filesystems

17.1.1 Filesystem Concepts

- The files stored on a disk partition are organised into a **filesystem**
- There are several filesystem types; the common Linux one is called **ext2**
- A filesystem contains a fixed number of **inodes**
- An inode is the data structure that describes a file on disk
- It contains information about the file, including its type (file/directory/device), modification time, permissions, etc.
- A file name refers to an inode, not to the file directly
- This allows **hard links**: many file names referring to the same inode

18.1.2 Potential Problems

- Over time, an active filesystem can develop problems:
- It can fill up, causing individual programs or even the entire system to fail
- It can become corrupted, perhaps due to a power failure or a system crash
- It can run out of space for inodes, so no new files or directories can be created
- Monitoring and checking filesystems regularly can help prevent and correct problems like these

18.2.0 Monitoring Space: df

- Run df with no arguments to get a listing of free space on all mounted filesystems
- Usually better to use the -h option, which displays space in human-readable units:

$ df -h

Filesystem Size Used Avail Use% Mounted on

/dev/hda8 248M 52M 183M 22% /

/dev/hda1 15M 5.6M 9.1M 38% /boot

/dev/hda6 13G 5.0G 7.4G 41% /home

/dev/hda5 13G 4.6G 7.8G 37% /usr

/dev/hda7 248M 125M 110M 53% /var

  - The Use% column shows what percentage of the filesystem is in use
  - You can give df directories as extra arguments to make it show space on the filesystems those directories are mounted on

18.2.1 Monitoring Inodes: df

- Filesystems rarely run out of inodes, but it would be possible if the filesystem contains many small files
- Run df -i to get information on inode usage on all mounted filesystems:

$ df -i

Filesystem Inodes IUsed IFree IUse% Mounted on

/dev/hda8 65736 8411 57325 13% /

/dev/hda1 4160 30 4130 1% /boot

/dev/hda6 1733312 169727 1563585 10% /home

/dev/hda5 1733312 138626 1594686 8% /usr

/dev/hda7 65736 1324 64412 2% /var

- In this example, every filesystem has used a smaller percentage of its inodes (IUse%) than of its file space
- This is a good sign!

18.3.0 Monitoring Disk Usage: du

- df shows a summary of the *free* space on a partition
- du, on the other hand, shows information about disk space *used* in a directory tree Takes one or more directories on the command line:

$ du /usr/share/vim

2156 /usr/share/vim/vim58/doc

2460 /usr/share/vim/vim58/syntax

36 /usr/share/vim/vim58/tutor

16 /usr/share/vim/vim58/macros/hanoi

16 /usr/share/vim/vim58/macros/life

40 /usr/share/vim/vim58/macros/maze

20 /usr/share/vim/vim58/macros/urm

156 /usr/share/vim/vim58/macros

100 /usr/share/vim/vim58/tools

5036 /usr/share/vim/vim58

5040 /usr/share/vim

18.3.1 du Options

- **Option Description**
    - -a Show all files, not just directories

- -c Print a cumulative total for all directories named on the command line
- -h Print disk usage in human-readable units
- -s Print only a summary for each directory named on the command line
- -S Make the size reported for a directory be the size of only the files in
- that directory, not the total including the sizes of its subdirectories

18.4.0 Finding and Repairing Filesystem Corruption: fsck

- Sometimes filesystems do become corrupted
- Perhaps there was a power failure
- Or maybe your kernel version has a bug in it
- The fsck program checks the integrity of a filesystem
- And can make repairs if necessary
- Actually has two main parts:
  - A 'driver program', fsck, which handles any filesystem type
  - One 'backend program' for each specific filesystem type
  - The backend program for ext2 is e2fsck, but it is always invoked through fsck

18.4.1 Running fsck

- fsck is normally run at system startup
- So it gets run automatically if the system was shut down uncleanly
- It can also be run manually:

 # fsck /dev/sdb3

- Interactively asks whether to fix problems as they are found

- Use -f to force checking the filesystem, even if fsck thinks it was cleanly umounted
- Use -y to automatically answer 'yes' to any question
- Usually a bad idea to run fsck on a mounted filesystem!

18.5 Exercises

Q1

a. Check the free disk space on the computer.

b. Display just the usage information for the partition that contains */usr/*. Display this in human-readable units.

c. Look at the free space and inodes of the partition of */var/tmp* first. Then run these commands:

$ mkdir /var/tmp/foo

$ seq -f '/var/tmp/foo/bar-%04.f' 0 2000 | xargs touch

What has happened? Look at the free space and inodes again.

Remove the files when you have finished.

Q2.

Go into the */var/* directory. Run each of the following commands as root, and explain the difference in their

output:

a. # du

b. # du -h

c. # du -h *

d. # du -hs

e. # du -hs *

f. # du -hsS *

g. # du -hsc *

h. # du -bsc *