

UNIX

Lesson 00:

IGATE is now a part of Capgemini

People matter, results count.

 Capgemini  
CONSULTING TECHNOLOGY OUTSOURCING

Copyright © 2011 IGATE Corporation (a part of Capgemini Group). All rights reserved.  
No part of this publication shall be reproduced in any way, including but not limited to  
photocopy, photographic, magnetic, or other record, without the prior written  
permission of IGATE Corporation (a part of Capgemini Group).

IGATE Corporation (a part of Capgemini Group) considers information included in  
this document to be confidential and proprietary.

## Document History

Date	Course Version No.	Software Version No.	Developer / SME	Change Record Remarks
	1.0	UNIX	Veena Deshpande	
30-Sep-2009	1.1	UNIX	Kishori Khadilkar	Revamped with new methodology
14-Oct-2009	1.1	UNIX	CLS team	Review
08-Jun-2011	1.2	UNIX	Rathnajothi Perumalsamy	Revamp/Refinement

## Course Goals and Non Goals

### ➤ Course Goals

- To understand UNIX operating system, Shell commands, Introduction to Processes, Writing simple shell scripts

### ➤ Course Non Goals

- NA



Pre-requisites

- None

February 1, 2016 Proprietary and Confidential - 4 -

 Capgemini  
CONSULTING TECHNOLOGIES OUTSOURCING

Intended Audience

➤ Novice Users



February 1, 2016 Proprietary and Confidential 54

 Capgemini  
CONSULTING TECHNOLOGY OUTSOURCING

## Day Wise Schedule

### ➤ Day 1:

- Lesson 1: Introduction to UNIX and Basic UNIX commands
- Lesson 2: UNIX file system
- Lesson 3: Filters

### ➤ Day 2:

- Lesson 3: Filters (contd..)
- Lesson 4: Introduction to Bourne Shell
- Lesson 5: Vi Editor

### ➤ Day 3:

- Lesson 6: Processes and related commands
- Lesson 7: AWK

### ➤ Day 4:

- Lesson 8: Shell Programming
- Lesson 9: Source Code Control System

February 1, 2016

Proprietary and Confidential

• 6 •



## Table of Contents

- **Lesson 1: Introduction to UNIX Operating System and Basic UNIX commands**
  - 1.1: Operating System
  - 1.2: Basic UNIX Commands
- **Lesson 2: UNIX File System**
  - 2.1: File System
  - 2.2: File Types
  - 2.3: File Permissions
  - 2.4: File Related Commands
- **Lesson 3 : Filters**
  - 3.1: Simple Filters
  - 3.2: Advanced Filters

## Table of Contents

- **Lesson 4: Introduction to Bourne Shell**
  - 4.1: Shell types
  - 4.2: Working of shell
  - 4.3: Metacharacter
  - 4.4: Shell redirections
  - 4.5: Command substitution
- **Lesson 5: Vi Editor**
  - 5.1: Vi Editor
  - 5.2: Input Mode Commands
  - 5.3: Vi Editor – Save & Quit
  - 5.4: Cursor Movement Commands
  - 5.5: Paging Functions
  - 5.6: Search and Repeat Commands
  - 5.7: Vi Editor – Other Features
  - 5.8: SED – Introduction to SED
  - 5.9: SED Commands

## Table of Contents

### ➤ **Lesson 6: Processes and related commands**

- 6.1: Process
- 6.2: Parent and Child Processes
- 6.3: Running a Command
- 6.4: ps Command
- 6.5: Jobs in the Background
- 6.6: Process related commands
- 6.7: Overview of Process Scheduling
- 6.8: Lowering job execution priority – nice command
- 6.9: Process related commands

### ➤ **Lesson 7: AWK Programming**

- 7.1: Advanced Filter - awk
- 7.2: AWK variables
- 7.3: AWK
- 7.4: Logical and Relational operators
- 7.5: Advanced Filter - awk
- 7.6: Number Processing

February 1, 2016

Proprietary and Confidential

v9



## Table of Contents

- **Lesson 7 :AWK Programming (contd..)**
  - 7.7: awk Command
  - 7.8: BEGIN and END
  - 7.9: Positional parameters and shell variable
  - 7.10: Built in functions - numeric
  - 7.11: If construct in AWK
  - 7.12: Pattern Matching
  
- **Lesson 8 : Shell Programming (contd..)**
  - 8.1: Shell Variables
  - 8.2: Environmental Variables
  - 8.3: Shell script Commands
  - 8.4: Arithmetic Operations
  - 8.5: Command Substitution
  - 8.6: Command Line Arguments
  - 8.7: Conditional Execution

## Table of Contents

- **Lesson 8 : Shell Programming (contd..)**
  - 8.8: if Statement Format
  - 8.9: Test - String Comparison
  - 8.10: The Case Statement
  - 8.11: While Statement
  - 8.12: Break & Continue Statement
  - 8.13: Until Statement
  - 8.14: Shell functions
  - 8.15: Using arrays.
- **Lesson 9 :Source Code Control System**
  - 9.1: SCCS – Source Code Control System

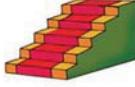
## References

- UNIX Concepts and Application; by Sumitabha Das
- The Unix Programming Environment; by Kernighan and Pike.
- UNIX Primer Plus, Third Edition; by Don Martin, Stephen Prata, Mitchell Waite, Michael Wessler, and Dan Wilson
- Advanced Unix : a programmers guide; by Stephen Prata



Next Step Courses (if applicable)

➤ **Linux Internal**



February 1, 2016 Proprietary and Confidential • 13 •

 Capgemini  
CONSULTING TECHNOLOGY OUTSOURCING

Other Parallel Technology Areas

➤ Windows OS

February 1, 2016 Proprietary and Confidential • 14 •

 Capgemini  
CONSULTING TECHNOLOGY OUTSOURCING

# UNIX

Lesson 01 : Introduction to UNIX Operating System and Basic UNIX Commands

## Lesson Objectives

➤ In this lesson, you will learn:

- Operating System
  - Functions of Operating System
  - History of UNIX
  - Features of UNIX
  - UNIX System Architecture
- Basic UNIX Commands



1.1: Operating System

## Overview

➤ An **Operating System (OS)** is the software that manages the sharing of the resources of a computer and provides programmers with an interface that is used to access those resources.

February 6, 2016 | Proprietary and Confidential | - 3 -

 Capgemini  
CONSULTING TECHNOLOGY OUTSOURCING

### Operating System: Overview:

An **Operating System (OS)** is the software that manages the sharing of the resources of a computer and provides programmers with an interface used to access those resources.

An Operating System processes **system data** and **user input**, and responds by allocating and managing tasks and internal system resources as a service to users and programs of the system. At the foundation of all system software, an Operating System performs basic tasks such as controlling and allocating memory, prioritizing system requests, controlling input and output devices, facilitating networking and managing file systems. Most Operating Systems come with an application that provides a user interface for managing the operating system, such as a command line interpreter or graphical user interface. The operating system forms a platform for other system software and for application software.

The most commonly used contemporary desktop and laptop (notebook) OS is **Microsoft Windows**. More powerful servers often employ Linux, FreeBSD, and other Unix-like systems. However, these Unix-like operating systems, especially Mac OS X, are also used on personal computers.

1.1: Operating System

## Functions of an Operating System

➤ Following are some of the important functions of an OS:

- Process Management
- Main-Memory Management
- Secondary-Storage Management
- I/O System Management
- File Management
- Protection System
- Networking
- Command-Interpreter System

February 6, 2016 | Proprietary and Confidential | - 4 -

 Capgemini  
CONSULTING TECHNOLOGY OUTSOURCING

### Functions of an OS:

Following are the various services provided by the Operating system:

1. **Process Management:** It involves creation and deletion of user and system processes, deadlock handling, and so on.
2. **Main-Memory Management:** It involves keeping track of the parts of memory that are being used, allocating/deallocating memory space as required, etc.
3. **Secondary-Storage Management:** It involves free-space management, disk scheduling, storage allocation.
4. **I/O System Management:** It deals with hardware specific drivers for devices, keeps it all hidden from the rest of the system.
5. **File Management:** It involves creating/deleting files and directories, backup, and so on.
6. **Protection System:** It involves controlling access to programs, processes, or users
7. **Networking:** It generalizes the network access.
8. **Command-Interpreter System:** It provides an interface between the user and the OS.

1.2: History of UNIX

## History

- UNIX evolved at AT&T Bell Labs in the late sixties.
- The writers of Unix are Ken Thomson, Rudd Canaday, Doug McIlroy, Joe Ossanna, and Dennis Ritchie.
- It was originally written as OS for PDP-7 and later for PDP-11.
- Liberal licensing: Various versions.
- System V in 1983 - Unification of all variants.

February 1, 2016 Proprietary and Confidential - 45 -



### History of UNIX:

Unix had a modest beginning at AT&T Bell Laboratories in late sixties, when AT&T withdrew its team from the MULTICS project. The immediate motivation towards development of Unix was a Space Travel game that Thompson had developed for PDP-7. Finding the game slow on the PDP machine, he requested for a DEC-10 machine. The request was rejected. This led to the idea of designing a new operating system for PDP-7.

As a result, a multitasking system supporting two users was developed. It had an elegant file system, a command interpreter, and a set of utilities. Initially called UNICS (as a pun on MULTICS), by 1970, it came to be known as Unix. Ken Thompson, along with Rudd Canaday, Doug McIlroy, Joe Ossanna, and Dennis Ritchie, were the writers of this operating system. Ritchie helped the system to be moved to PDP-11 system in 1970. Ritchie and Kernighan also wrote a compiler for "C".

Unix was originally written in Assembly language. In 1973, Ritchie and Thompson rewrote the Unix kernel in "C". This was a revolutionary step, as earlier the operating systems were written in assembly languages. The idea of writing it in "C" was so that the Operating system could now be ported (the speed, in effect, was traded off). It also made the system easier to maintain and adapt to particular requirements.

Around 1974, the system was licensed to universities for educational purposes, and was later available for commercial use. The licensing by AT&T was very liberal – the source code was made available to universities, industries, and government organizations. This led to development of various versions of Unix as everybody added their own utilities. The important ones amongst them include BSD (Berkeley Software Distribution from University of California, Berkeley), SunOS (from Sun Microsystems).

1.3: Features of Unix

## Features

➤ **UNIX OS exhibits the following features:**

- It is a simple User Interface.
- It is Multi-User and Multiprocessing System.
- It is a Time Sharing Operating System.
- It is written in "C" (HLL).
- It has a consistent file format - the Byte Stream.
- It is a hierarchical file system.
- It supports Languages such as FORTRAN, BASIC, PASCAL, Ada, COBOL, LISP, PROLOG, C, C++, and so on.

February 6, 2016 Proprietary and Confidential | 6/6



### Features of UNIX:

UNIX is a **timesharing operating system** written in "C". It is a **multi-user** as well as a **multiprocessing system**. Many users can work with multiple tasks at the same time. **Round Robin Scheduling** with fixed time slices is the algorithm that is used by the CPU for scheduling tasks.

UNIX, as it is written in "C", is portable. It is available on a variety of platforms ranging from a 8088 based PC to a parallel processing system like Cray 2.

Everything in UNIX is a **file** – even if it is a directory or a device. The file format is consistent. A file is nothing but a **stream of bytes** – it is not considered as containing fixed length records. UNIX follows a hierarchical file system. All files are related with root at the top of the hierarchy – it follows an inverted tree structure.

The kernel is a set of "C" program that controls the resources of a computer and allocates them amongst its users. It lets users run their programs, controls peripheral devices, and provides a file system to manage programs and data. It is loaded into memory when system is booted. It is often called as "the" operating system.

The approach in UNIX is that there is no need to have separate utilities to solve each and every complex problem. Instead, by combining several commands (each of which is capable of doing a simple job), it is possible to solve bigger problems. Pipes and filters allow building solutions from simple commands – they also make UNIX powerful and flexible.

## Services

### ➤ Services Provided by UNIX:

- Process Management:
  - ★ It involves Creation, Termination, Suspension, and Communication between processes.
- File Management:
  - ★ It involves aspects related to files like creation and deletion, file security, and so on.

February 6, 2016

Proprietary and Confidential

– 7 –



### Services provided by UNIX:

Amongst the various services that UNIX operating system provides are Process Management (Creation, Termination, Suspension and communication between processes) and File Management (creation and deletion of files, allowing for dynamic growth of files, security of files etc). Files and processes are two major entities in Unix. A file resides in memory – it is static in nature – as against a process, which is alive and exists in time.

UNIX has a very simple user interface: programmers have written it and it is meant for programmers, hence there are absolutely no “frills”. However, graphical interfaces are becoming available in later releases.

UNIX includes editors and compilers (for various languages like C, C++, Pascal, Fortran, Prolog, Lisp, Java, and so on). It has several utilities like calculators, graphics and statistical packages, and tools for document preparation.

UNIX includes an online help facility, which is very useful to look at the syntax involving several different options for the many commands of UNIX.

### **File Management:**

It helps to create or delete files. It assigns read, write, and execute permissions for users, groups, and others to restrict access to file.

1.4: UNIX System Architecture

## UNIX System Architecture

➤ Following is a pictorial representation of the UNIX system:

**Parts of the UNIX System**

The diagram illustrates the hierarchical structure of the UNIX system. It consists of three concentric circles. The innermost circle is yellow and labeled 'Kernel'. The middle circle is purple and labeled 'Shell'. The outermost circle is light blue and labeled 'Tools & Apps'. The text 'Parts of the UNIX System' is centered above the diagram.

February 1, 2016 | Proprietary and Confidential | + 8 +

**Capgemini**  
CONSULTING TECHNOLOGY OUTSOURCING

### UNIX System Architecture:

The UNIX system is functionally organized at three levels:

- The **kernel**, which schedules tasks and manages storage
- The **shell**, which connects and interprets users' commands, calls programs from memory, and executes them
- The **tools and applications** that offer additional functionality to the operating system

There is a policy of division of labour that is followed between the kernel and the shell on the Unix systems. The **kernel** interacts with the machine hardware, and the **shell** interacts with the User.

#### The kernel:

The heart of the operating system, the kernel controls the hardware and turns part of the system on and off at the programmer's command. Suppose you ask the computer to list all the files in a directory. Then the kernel tells the computer to read all the files in that directory from the disk and display them on your screen.

#### The shell:

The **shell** is technically a **UNIX** command that interprets the user's requests to execute commands and programs. The **shell** acts as the main interface for the system, communicating with the **kernel**. The shell is also programmable in UNIX. There are many different types of shells like **Bourne Shell**, **Korn Shell**, and **C Shell**, most notably the command driven **Bourne Shell** and the **C Shell**, and menu-driven shells that make it easier for beginners to use. Whatever shell is used, its purpose remains the same – to act as an interpreter between the user and the computer.  
We will see more about shell later.

#### Tools and applications:

There are hundreds of tools available to UNIX users, although some have been written by third party vendors for specific applications. Typically, tools are grouped into categories for certain functions, such as word processing, business applications, or programming.

1.5: Basic Unix Command

## Logging In and Out Commands

➤ **Logging In and Out:**

- Logon name and password are required.
- Successful logon places user in home directory.

February 5, 2016 Proprietary and Confidential | 9



### Basic Unix Commands:

In order to work with Unix, one needs a login name and password, which the system administrator needs to assign. With these in hand, one can "log on" to a Unix System and start a "session" with Unix by typing out the login name and password at the prompt. Once the session is through, one needs to "log off" from the system.

#### Logging In and Out:

Once the terminal is connected to a Unix system, it displays the "login:" message on screen. The login name, given by the system administrator needs to be typed here. If the password has been set, the system will prompt for the password (printing will be turned off when password is typed).

In case of valid **user name** and **password**, the user will get logged on to the system. The system displays a prompt, typically a **\$ sign** (it can be different as it is possible to change prompts) – which indicates that the system is ready to accept commands. There can be some messages and other notifications before the prompt.

On successful logon, user is automatically placed in home directory, the name of home directory being usually the same as the login name.

If there is an error in validating the user name and password, then the system requests the user to re-enter the same. Usually due to security considerations, system allows only a fixed number of attempts to re-enter information.

## man Command

### ➤ man command:

- The on line help provided by the **man** command includes brief description, options, and examples.
- Example:

```
$man <command>
```

February 6, 2016Proprietary and Confidential+ 10 +

### **man Command:**

#### **Online help using the man command:**

Using the **man** command, it is possible to get a brief description about a command including all its options as well as some suitable examples.

**Example:** To get help on **passwd** command, use the following syntax:

```
$ man passwd
```

## cal Command

### ➤ cal command:

- The **cal** command is used to display calendar from the year 1 to 9999.
- Example:

```
$cal 9 2001
```

- The above syntax can be used to print the calendar for the 9<sup>th</sup> month of the year 2001.

February 6, 2016

Proprietary and Confidential



### The calendar – cal command:

Any calendar from the year 1 to 9999 (either for a month or complete year), can be displayed using this command.

An example is shown in the above slide.

September 2001  
Su Mo Tu We Th Fr Sa  
1  
2 3 4 5 6 7 8  
9 10 11 12 13 14 15  
16 17 18 19 20 21 22  
23 24 25 26 27 28 29  
30

**Example:** Using the cal command, observe the calendar for September 2001!

## date Command

➤ **date command:**

- The **date** command is used to see current date and time.
- Date can be displayed in different formats
- Example:

```
$ date
```

\* **Output:** Fri Apr 6 11:14:46 IST 2001

```
$ date "+%T"
```

-- %t is used to display only time

\* **Output:** 11:15:20

```
$ date "+%d %h"
```

-- To display date and month name

\* **Output:** 6 Apr



February 6, 2016 | Proprietary and Confidential | - 12 -

**Displaying System Date – date:**

Unix has an internal clock, which actually stores the number of seconds elapsed from 1 Jan. 1970; and it is used for time stamping. A number of options are separately available to retrieve various components of the date.

**Table:** Commands used with date command

Option	Description
m	Month
h	Month Name
d	Day of month
y	Year (Last 2 digits)
H	Hour
M	Minutes
S	Second
T	Time in hh:mm:ss
A	Day of week (Sun to Sat)
R	Time in AM/PM

## lp Command

### ➤ lp command:

- The lp command is used for printing files.
  - Examples:

```
$lp myfile.txt
```

```
$lp -n 10 myfile.txt
```

```
$lpq
```

```
$lprm -Pps99 11042
```

February 1, 2016 | Proprietary and Confidential | +13+



### lp command – for printing files:

Jobs can be queued up for printing by using the spooling facility of Unix. Several users can print their files without any conflict. This command can be used to print the file, it returns the job number that can later be used to check the status of job.

The command is used as follows:

lp -n 10 mytextfile.txt

-n option sets number of copies to print from 1 to 100.

lpq

Prints jobs in queue.

lprm -Pps99 11042

the lprm command lets you cancel the printing of a file. To remove a print job, first use the lpq command to find the job number of the print request you want to cancel.

Then use the lprm command to kill the job by specifying the job number.

```
$ lp file1.txt
```

## nl Command

### ➤ nl command:

- The nl command is used to print file contents along with line numbers.
- Options:
  - -w : width of the number
  - -v : Indicate first line number
  - -i : increment line number by
- Example:

```
$ nl myfile.txt
1 line one
2 line two
```

February 6, 2016

Proprietary and Confidential



### Line Numbering – nl :

This command elaborates schemes for numbering lines.

#### Options:

- -w : width of the number
- -v : Indicate first line number
- -i : increment line number by

**Example:** Using the nl command:

```
$nl -w2 -v40 -i7 file1.txt
40    one
47    two
54    three
61    four
```

## tty Command

### ➤ **tty Command:**

- Unix treats a terminal also as a file. In order to display the device name of a terminal, the **tty** (teletype) command is used.
- print the file name of the terminal connected to standard input
- Example: Using **tty** command

```
$ tty  
/dev/ttyp3
```

February 6, 2016

Proprietary and Confidential

&lt; 15 &gt;



### **tty Commands:**

Unix treats terminal also as a file. In order to display the device name of a terminal, the command used is **tty** (teletype).

The above slide shows an example on using the **tty** command.

In order to display the current terminal related settings, the **stty** command can be used.

The output of the **stty** command depends on the Unix implementation.

This command can also be used to change some settings. For example, in order to use <Ctrl-a> instead of <Ctrl-d> as end of file indicator.

## who Command

➤ **who Command:**

- To list all users who are currently logged in
- Example:

```
$who
```

• Output:

ssdesh	ttypo	Mar 29 09:00
root	tty01	Mar 29 10:32
root	tty03	Mar 29 10:37

➤ **\$who am I Command:**

- To see the current user

February 3, 2016

Proprietary and Confidential



### Login details – who command:

Unix maintains an account of all current users of system, the list of which can be printed using this command. The command can also be used to get one's own login details.

**Example 1:** Using the who command

```
$ who
```

ssdesh	ttypo	Mar 29 09:00
root	tty01	Mar 29 10:32
root	tty03	Mar 29 10:37
root	tty11	Mar 29 09:52
pmaint	tty12	Mar 29 10:00
deshpavn	ttyp1	Mar 29 10:38
munageka	ttyp2	Mar 28 16:55
deshpavn	ttyp3	Mar 29 10:49

**Example 2:**

```
deshpavn ttyp3 Mar 29 10:49
```

```
$ who am i
```

## Summary

➤ **In this lesson, you have learnt:**

- UNIX is multi-user, multiprocessor, time sharing operating system.
- It uses hierarchical file system.
- The UNIX system is functionally organized at three levels: Kernel, shell, tools and applications.



### Review Questions

- Question 1: \_\_\_ controls system hardware.
- Question 2: The kernel interacts with the machine hardware, and the shell interacts with the User.
  - True / False
- Question 3: \_\_\_ command displays details of all users currently logged in.



## UNIX

UNIX File System

February 1, 2010 | Proprietary and Confidential | 414



## Lesson Objectives

➤ **In this lesson, you will learn:**

- UNIX File system
- File types
- File permissions
- Commands related to file permission
  - \* mkdir, cd, cat etc...



2.1: File System

## Overview

Let us discuss a File System with respect to the following:

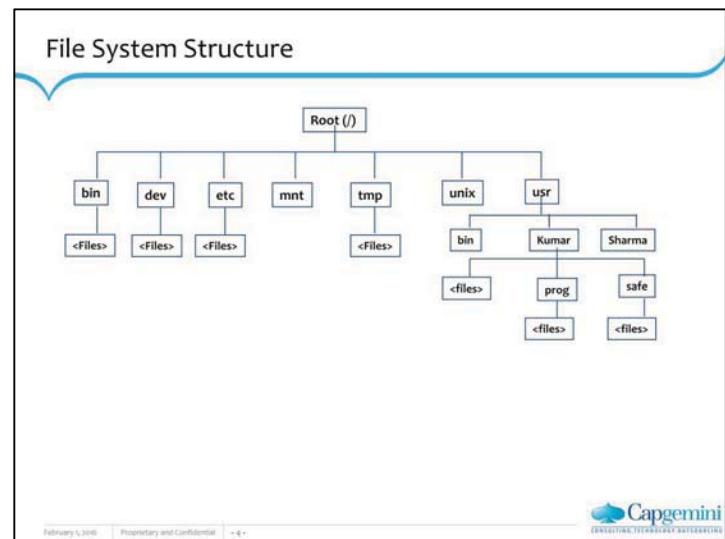
- Hierarchical Structure
- Consistent Treatment of Data: Lack of file format
- The Treatment of Peripheral Devices as Files
- Protection of File Data

February 1, 2016 | Proprietary and Confidential | +3+

 Capgemini

### The UNIX File System:

UNIX works with a large number of files, which can belong to several different users. It becomes imperative for UNIX to organize files in a systematic fashion. The simple file system of UNIX is designed as an elaborate **Storage System** with separate compartment becoming available to store files. The system is widely adopted by different systems including DOS.



#### File System Structure:

The file system in UNIX is hierarchical. The **root**, a directory file represented by `/`, is at the top of the hierarchy and has several subdirectories (branches) under it. An example of a typical UNIX file structure is given in the above slide.

There can be more than one file system, each with its own root, in a single machine. The number of file systems cannot be less than the number of physical disks but can certainly exceed the latter. However, a file system cannot span multiple disks. If there are more than one file systems, there will always be one main file system.

## File System Structure

- / bin : commonly used UNIX Commands like who, ls
- /usr/bin : cat, wc etc. are stored here
- /dev : contains device files of all hardware devices
- /etc : contains those utilities mostly used by system administrator
  - Example: passwd, chmod, chown

## File System

- /tmp : used by some UNIX utilities especially vi and by user to store temporary files
- /usr : contains all the files created by user, including login directory
- /unix : kernel
- **Release V:**
  - It does not contain / bin.
  - It contains / home instead of /usr.

2.2: File Types

## File Types in UNIX

➤ We have the following file types in UNIX:

- Regular File
- Directory File
- Device File

February 4, 2016 | Proprietary and Confidential | -7-

 Capgemini  
PERPETUAL INNOVATION

### UNIX Files:

A UNIX file consists of a sequence of characters, and there are no restrictions on the file structure. The file consists only the actual bytes – and no extra information like its own size, attributes, or even an end of file marker. Extra information is stored in a structure called as **inode (index node)**.

In UNIX, for every file, an inode is stored irrespective of its type. Everything in UNIX is treated as a file, may be it is a user created or system file, a directory or a peripheral device. UNIX treats all the files in a consistent format. The lack of file formats and the consequent uniformity of files is of advantage to the programmer, as programmers do not need to worry about the file types. Further most of the standard programs work with any file.

Though everything is treated as a file, the significance of the file attributes is dependent on the categorization of files as Ordinary files, Directory Files and Device files.

#### **Ordinary Files or Regular File :**

These are also called as regular files. This is the “traditional” file, which can contain programs, data, object, and executable code, as well as all the UNIX programs and other user created files.

All text files belong to this category.

2.3: File Permissions

## File Permissions in UNIX

### File Access Permissions

`rwx rwx rwx`

user      group      others

February 1, 2016 | Proprietary and Confidential | 8+

 Capgemini  
PERPETUAL INNOVATION

#### File Permissions:

Every file in UNIX has a set of permissions using which it is determined “who can do what” with the file. This is stored as part of **inode** information, and is useful for maintaining file security.

There are three categories of users: **Owner** (the user), **Group** (user is a member of which group), and **Others** (everybody else on the system). Access permissions of read (examining contents), write (changing contents) and execute (running program) are assigned to a file. These permissions are assigned to file owner, group and others.

It is the file permissions based on which reading, writing or executing a file is decided for a particular user. It helps in giving restricted access to a file. The commands to assign permissions and also to change the same are discussed later.

## File Permissions in UNIX

- Permissions are associated with every file, and are useful for security.
- There are three categories of users:
  - Owner (u)
  - Group (g)
  - Others (o)
- There are three types of “access permissions”:
  - Read (r)
  - Write (w)
  - Execute (e)

2.4: File Related Commands

## pwd Command

➤ The **pwd** command checks current directory.

```
$ pwd
```

➤ **Output:** /usr/Kumar

February 1, 2018 | Proprietary and Confidential | 4 of 6

 Capgemini  
INNOVATION PRACTITIONERS

**pwd Command:**

The **pwd** command is used to print the name of the current working directory.

```
$ pwd
```

**Output:** /usr1/deshpavn

## cd Command

- The cd command changes directories to specified directory
- The directory name can be specified by using absolute path (Full Path) or relative path

```
$ pwd
```

Output: /usr/kumar

```
$ cd Prog
$ pwd
```

Output: /usr/kumar/Prog

February 1, 2016

Proprietary and Confidential

91



### cd (change directory) Command:

The cd command is used to change the current directory to the directory specified as the argument to the command. The argument can contain absolute as well as relative paths.

```
$ cd deshpavn
$ pwd
```

Output: /usr1/deshpavn

### Absolute Path and Relative Path

To locate the file or directory which is not in current working directory full path can be given. If the specified path starts from root directory i.e from '/' the it is called as **absolute path or full path**. Otherwise it is called as **relative path** because the given path is relative to the current working directory

e.g If current working directory is /usr, then to change the directory to prog  
 Use following command  
 \$pwd  
 /usr  
 \$cd kumar/prog

In the above example, specified path to prog directory is not starting with '/'. It is relative path from /usr directory(current working directory).

The same command can be given as  
 \$cd /usr/kumar/prog

In above example, since the given path is starting from root directory i.e '/' hence this is called as absolute path or full path.

## cd Command

- Moving one level up:

```
$ cd ..
```

Switching to home directory:

```
$ cd
```

- Switching to /usr/sharma:

```
$ cd /usr/Sharma
```

- Switching to root directory:

```
$ cd /
```

February 1, 2016Proprietary and Confidential~ 92 ~

### cd (change directory) Command:

To move one level up in the file system, the command is used as follows:

```
$ pwd
```

Output: /usr1/deshpavn

```
$ cd ..
```

```
$ pwd
```

Output: /usr1

When the command is used without any parameters, it switches to home directory as the current directory.

Output: /usr1

```
$ pwd
```

Output: /usr1/deshpavn

```
$ cd
```

```
$ pwd
```

## logname Command

➤ The logname command checks the login directory.

```
$ logname
```

Output: Kumar

February 1, 2016Proprietary and Confidential13/14

### logname Command - Checking Login Directory :

The **logname** command can be used to display the name of login directory, irrespective of what is the current working directory.

```
$ logname
```

Output: deshpavn

## ls Command

- The ls command lists the directory contents.
- Example:

```
$ ls
```

**Output:**

a.out  
chap1  
chap2  
test  
test.c

**ls Command – Listing directory contents:**

The ls command is used to list all the contents of the specified directory (in case no argument is specified, current directory is assumed).

```
$ ls
```

**Output:**

file1.txt  
file2.txt  
file3.txt  
Mail

```
$ ls
```

**Output:**

newfile.txt

## ls Command

### ➤ Options available in ls command:

Option	Description
-x	Displays multi columnar output (prior to Release 4)
-F	Marks executables with * and directories with /
-r	Sorts files in reverse order (ASCII collating sequence by default)
-l	The long listing showing seven attributes of a file
-d	Forces listing of a directory
-a	Shows all files including ., .. And those beginning with a dot

February 1, 2016 | Proprietary and Confidential | 19/16



### ls Command – Listing directory contents (contd.):

The ls command comes with several options, which are listed in the table. Many of these options can be combined to get relevant output. The command can also be used with more than one file name that is specified.

## ls Command

### ➤ Options available in ls command:

Option	Description
-t	Sorts files by modification time
-R	Recursive listing of all files in sub-directories
-u	Sorts files by access time (when used with the -t option)
-i	Shows i-node number of a file
-s	Displays number of blocks used by a file

February 1, 2016

Proprietary and Confidential



### ls Command – Listing directory contents (contd.):

Some of the examples are considered here:

**Example 1: To produce multiple column output (-x):**

```
$ ls -x
```

**Output:** file1.txt file2.txt file3.txt mail

**Example 2: To identify directories and executable files (-F):**

Here two tags, that is \* and /, are used to indicate executable file and a directory respectively.

## ls Command

➤ Example:

```
$ ls -l
```

- It displays output as follows which includes 7 columns  
total 8:  
 -rw-rw-rw- 1 Kumar group 44 May 9 09:08 dept.h  
 -rw-rw-rw- 1 Kumar group 212 May 9 09:08 dept.q  
 -rw-rw-rw- 1 Kumar group 154 May 9 09:08 emp.h

February 1, 2016 | Proprietary and Confidential | + 12 +



### ls Command – Listing directory contents (contd.):

Example 3: To get an informative listing i.e. long listing (-l)

```
$ ls -l
```

```
total 12
-rw-r--r-- 1 deshpavn group 60 Mar 29 10:43 file1.txt
-rw-r--r-- 1 deshpavn group 61 Mar 29 10:44 file2.txt
-rw-r--r-- 1 deshpavn group 60 Mar 29 10:44 file3.txt
drwx----- 2 deshpavn group 512 Mar 14 10:00 mail
drwxr-xr-x  2 deshpavn group 512 Mar 29 10:46 testdir1
drwxr-xr-x  2 deshpavn group 512 Mar 29 10:47 testdir2
```

Seven attributes are listed which are explained in further slides in order as they appear in listing.

#### Type and Permissions associated with file:

The first bit “d” indicates that file is a **directory file**, while “-” indicates that file is a **regular file**. In case of device files, one would normally find bit “c” for **character based device** and bit “b” for **block based devices**.

File permissions are the read, write, and execute permissions set for owner, group, and others. Significance of these permissions with respect to directories are discussed in another example.

## ls Command

➤ Consider the first column:  
Field1 → mode  
- rwx rwx rwx  
\* \* ●  
\* → user permissions  
\* → group permissions  
● → others permissions

February 4, 2016 | Proprietary and Confidential | + 18 +

**ls Command – Listing directory contents (contd.):**

Permissions are interpreted as follows:

For a directory: The “r” permission implies that one can find out the contents of the directory using commands like ls. The “w” permission implies that it is possible to create and delete files in this directory. It is possible to remove even the files that are write-protected. The “x” permission means search rather than execute. That is, it implies that the directory can be searched for a file. Also, when ‘x’ permission is set for a directory, one can do change dir to that directory and not otherwise.

Example: Using “r.” will ensure that users can see the directory contents using ls, but cannot really use the contents.

To list all hidden files (-a):

The ls command does not list all the hidden files unless –a attribute is used.

To display inode information and number of blocks used for file:

```
$ ls -i
```

```
658 file1.txt
3952 file2.txt
3956 file3.txt
434 mail
3957 testdir1
3960 testdir2
```

```
$ ls -i testdir1
```

```
658 fileln1.txt
```

## ls Command

➤ **File type**

- 1st character represents file type:
  - **rwx rwx rwx**
  - -> regular file
  - d-> directory file
  - c-> character - read
  - b-> block read

February 1, 2010 | Proprietary and Confidential | + 19 +

 Capgemini

## ls Command

- Field2 : indicates number of links
- Field3 : File owner id
- Field4 : Group id
- Field5 : File size in bytes
- Field6 : Date/time last altered
- Field7 : Filename

## cat Command

- The **cat** command is used for displaying and creating files.

- To display file:

```
$ cat dept.lst
```

```
0]accounts|6213  
0]admin|5423  
1  
06]training|1006
```

- To create a file:

```
$cat > myfile
```

- This is a new file
    - Press ctrl-d to save the contents in file myfile



### cat Command (Displaying and Creating files):

The **cat** command can be used to display one or more files (if there is more than one file, then contents of the remaining immediately follow without any header information). To control the scrolling of text on screen, you can use <Control-s> and <Ctrl-q>.

```
$ cat file1.txt
```

**Output:** This is a sample text file.

This is the first file created.

```
$ cat file1.txt file2.txt
```

**Output:** This is a sample text file.

This is the first file created.

This is a sample text file.

This is the second file created.

The command can also be used to create a file as described below. The meaning and significance of the “>” symbol will be discussed later.

```
$ cat > newfile.txt
```

**Output:** This is a new file being created.  
Use the redirections to save contents into a file.  
To indicate end of input, press <Ctrl-d>

## cat Command

- The **cat** command can be used to display contents of more than one file.

- It displays contents of chap2 immediately after displaying chap1.

```
$ cat chap1 chap2
```

## Input and Output Redirection

- **Standard Input: Keyboard**
- **Standard Output: Monitor**
- **Standard Error: Monitor**
- **Redirection operators:**
  - < : Input Redirection
  - > : Output Redirection
  - 2> : Error Redirection
  - >> : Append Redirection

February 1, 2016 | Proprietary and Confidential | 43 / 42



### Input and Output Redirection:

Many commands work with **character streams**. The default is the keyboard for input (standard input, file number 0), and terminal for the output (standard output, file number 1). In case of any errors, the system messages get written to standard error (file number 2), which defaults to a terminal.

UNIX treats each of these streams as files, and these files are available to every command executed by the shell. It is the shell's responsibility to assign sources and destinations for a command. The shell can also replace any of the standard files by a physical file, which it does with the help of **metacharacters** for redirection.

## Redirection

- Input redirection: **Instead of accepting i/p from standard i/p(keyboard) we can change it to file.**
  - Example: `$cat < myfile` will work same as `$cat myfile`
  - < indicates, take i/p from myfile and display o/p on standard o/p device.
- Output redirection: **To redirect o/p to some file use >**
  - Example: `$cat < myfile > newfile`
  - The above command will take i/p from myfile and redirect o/p to new file instead of standard o/p (monitor).

February 1, 2016 | Proprietary and Confidential | 4/34



### Redirection:

In order to take the input from a file (instead of standard i/p), the character < is used.

**Example:** `$ cat < file1.txt`

This is example of i/p redirection.

To redirect the output to a file, > is used. If outfile does not exist, it is first created; otherwise, the contents are overwritten. In order to append output to the existing contents, >> is used.

In the following command, the `cat` command will take i/p from file `file1.txt` and send the o/p to `result` file. If result file exists, then contents will be overwritten, otherwise new result file will be created.

**Example:**

`$ cat < file1.txt > result` is same as

`$cat file1.txt > result`

`$ cat result`

`2 12 60`

## Redirection

- `$ cat < file1.txt > result` is same as `$cat file1.txt > result`.

```
$ cat result
```

Output: 2 12 60

- `>>` is append redirection
- The given command will append the contents of `file1.lst` in `result` file.

```
$ cat < file1.lst >> result  
$ cat result
```

Output: 2 12 60  
4 4 8

**Note:**

`>>` - is append redirection

If you want to retain previous contents of a file and append new contents to a file then use append redirection operator.

```
$ cat < file1.lst >> result  
$ cat result  
2 12 60  
4 4 8
```

## cat file exist/not exist

- Consider an example of cat -(file exist/not exist):

```
$ cat abc.txt > pqr.txt 2> errfile.txt
```

- If file abc.txt exists:
  - \* Then contents of the file will be sent to pqr.txt. Since no error has occurred nothing will be transferred to errfile.txt.
- If abc.txt file does not exist:
  - \* Then the error message will be transferred to errfile.txt and pqr.txt will remain empty.

February 4, 2016 | Proprietary and Confidential | 4/30



### cat file ext/not exit:

As shown in the above slide, it is possible to combine redirection operators on a single command line. The order of the redirection symbols does not affect the working of the command.

To redirect the standard error, 2> is used.

```
$ cat xyzfile
```

```
$ cat xyzfile 2> errfile  
$ cat errfile
```

cat: cannot open xyzfile: No such file or directory (error 2)

```
$ cat xxfile 2>> errfile  
$ cat errfile
```

cat: cannot open xyzfile: No such file or directory (error 2)

cat: cannot open xyzfile: No such file or directory (error 2)  
cat: cannot open xxfile: No such file or directory (error 2)

## cp Command (copy file)

- The cp (copy file) command copies a file or group of files.
- The following example copies file chap1 as chap2 in test directory.
  - Example:

```
$ cp chap1 temp/chap2
Option -i (interactive)
$cp -i chap1 chap2
cp: overwrite chap2 ?y
Option -r (recursive) to copy entire directory
$cp -r temp newtemp
```

February 1, 2016 | Proprietary and Confidential | 427



### cp Command (Copying Files):

The copy command copies a file or groups of files.

```
$ cp newfile.txt anotherfile.txt
$ cp newfile.txt testdir1/nfile1.txt
```

Copy all files with extension txt:

```
$ cp *.txt *.dat
```

All files with extension will be copied with same name but extension will change to dat.

Example:

abc.txt will be copied as abc.dat  
pqr.txt will be copied as pqr.dat

Files cannot be copied if they are read protected, or if destination file/directory is write protected.

Copying file using redirection:

```
$cat newfile.txt > anotherfile.txt
```

The result of above command is same as \$ cp newfile.txt anotherfile.txt

## rm Command (delete file)

➤ The rm (remove file) command is used to delete files:

```
$ rm chap1    chap2    chap3
$ rm *
      Are you sure? y
Option -i (interactive delete)
$ rm -i chap1    chap2
chap1: ? Y
chap2: ? Y
Option -r (recursive delete) (Avoid using this option)
```

February 1, 2016 | Proprietary and Confidential | 138



### rm Command (Removing Files):

This command is used to delete files. There are options for **interactive (-i)** delete and **recursive (-r)** delete. The **-r** option will delete files from subfolders also.

```
$ rm newfile.txt
Option -r (recursive delete)
$ rm -r *
(Warning: Pl. do not use this option)
```

## mv Command

- The mv command is used to rename file or group of files as well as directories.

```
$ mv chap1 man1
```

- The destination file, if existing, gets overwritten:

- Example: \$ mv temp doc
- Example: \$ mv chap1 chap2 chap3 man1
  - It will move chap1, chap2 & chap3 to man1 directory

February 1, 2016 | Proprietary and Confidential | 4/29



### mv Command (Renaming Files):

Files as well as directories (belonging to the same parent) can be renamed using this command.

```
$ mv anotherfile.txt newfile.txt
$ mv testdir1 testdir2
$ ls -l
```

```
total 12
-rw-r--r-- 3 deshpavn group      60 Mar 29 10:43 file1.txt
-rw-r--r-- 1 deshpavn group      61 Mar 29 10:44 file2.txt
-rw-r--r-- 1 deshpavn group      60 Mar 29 10:44 file3.txt
drwx----- 2 deshpavn group     512 Mar 14 10:00 mail
-rw-r--r-- 1 deshpavn group    126 Mar 29 10:54 newfile.txt
drwxr-xr-x  3 deshpavn group    512 Mar 29 10:58 testdir2
```

## wc Command

- The wc command counts lines, words, and character depending on option.
- It takes one or more filename as arguments.
- no filename is given or - will accept data from standard i/p.

```
$ wc infile
3 20 103 infile
$wc   or $wc -
This is standard input
press ctrl-d to stop
```

\* Output: 2 8 44

February 4, 2016 | Proprietary and Confidential | 100%



### Line, word, and character counting – wc

This command can be used to count the number of lines (-l option), words (-w option), or characters (-c option) for one or more files. If we specify multiple files, then the list of files should be separated by space. If no file name is specified, then it will accept data from standard i/p, that is from the keyboard.

Example:

```
$ wc file1.txt
```

2 12 60 file1.txt

```
$ wc -lw file1.txt
```

212 file1.txt

## wc Command

\$ wc infile test

Output:	3	20	103	infile
	10	100	180	test
	13	120	283	total

\$ wc - l infile

Output:	3	infile
---------	---	--------

\$ wc - wl infile

Output:	20	3	infile
---------	----	---	--------

The following command will take i/p from infile and send o/p to result file

\$ wc < infile > result

\$ cat result

Output:	2	12	60
---------	---	----	----



### wc Command with Redirection:

In order to take the input from a file (instead of standard i/p), the character < is used.

\$ wc < file1.txt

Output: 2 12 60

To redirect the output to a file, > is used. If outfile does not exist, it is first created; otherwise, the contents are overwritten. In order to append output to the existing contents, >> is used.

In following command, wc will take i/p from file file1.txt and send the o/p to result file. If result file exists, contents will be overwritten, otherwise new result file will be created.

\$ wc < file1.txt > result

\$ cat result

Output: 2 12 60

\$ wc < cfile1.lst >> result

\$ cat result

Output: 2 12 60  
4 4 8

## cmp Command

➤ **cmp Command:**

```
$ cmp file1.txt file2.txt
file1.txt file2.txt differ: char 41, line 2
$ cmp file1.txt file1.txt
```

February 1, 2016 | Proprietary and Confidential | + 32 +

 Capgemini

### cmp Command (Comparing Files):

#### Using the cmp Command:

The cmp command can be used to compare if two files are matching or not. The comparison is done on a byte by byte basis. In case files are identical, no message is displayed. Otherwise, locations of mismatches are echoed.

```
$ cmp file1.txt file2.txt
file1.txt file2.txt differ: char 41, line 2
$ cmp file1.txt file1.txt
```

## comm Command

**➤ comm Command:**

- The comm command compares two sorted files. It gives a 3 columnar output:
  - First column contains lines unique to the first file.
  - Second column contains lines unique to the second file.
  - Third column displays the common lines.

February 1, 2016 | Proprietary and Confidential | 433

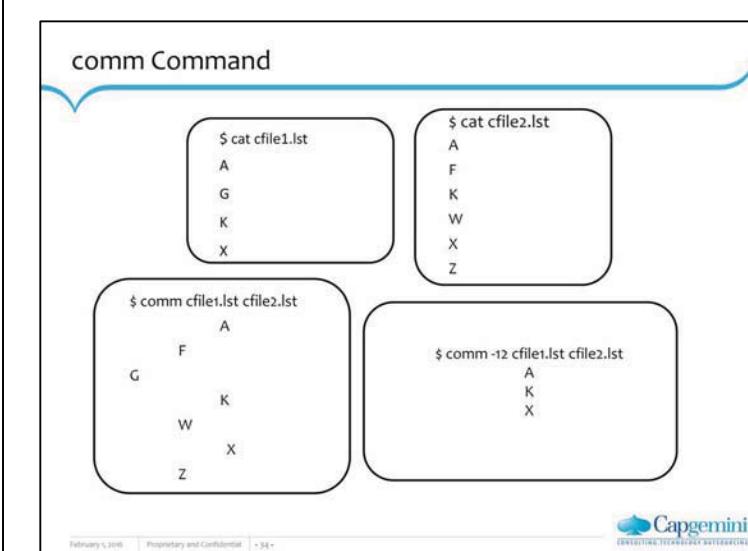
 Capgemini  
INNOVATION PRACTICING

### comm Commands (Comparing Files):

The comm command compares two sorted files. It gives a 3 columnar output.  
First column contains lines unique to the first file.

Second column contains lines unique to the second file.  
Third column displays the common lines.

Selective column output can be obtained by using options **-1**, **-2** or **-3**. It would drop the column(s) specified from the output.



In above example

\$ comm cfile1.lst cfile2.lst

This command is showing output in 3 columns

First column display lines unique to the cfile1.lst.

Second column display lines unique to the cfile2.lst.

Third column displays the lines common in both files.

\$ comm -12 cfile1.lst cfile2.lst

This command will hide first and second column and display only 3 rd column i.e lines common in both files.

## diff Command

- The diff command is used to display the file differences. It tells the lines of one file that need to be changed to make the two files identical.

— Example:

```
$ diff cfile1.lst cfile2.lst
2c2
< G
> F
3a4
> W
4a6
> Z
```

February 1, 2016 | Proprietary and Confidential | + 95 +



### diff Command:

The file differences can be displayed using the diff command. It tells which lines of one file need to be changed to make the two files identical.

**Example:** Using the diff command

```
$ diff cfile1.lst cfile2.lst
2c2      change line 2 of first file which is line 2 in 2nd file
< G      replacing this line
---
> F      with
3a4      this line
> W      append after line 3 in first file
4a6      this line
> Z      append after line 4 in first file
this line
```

## tr Command

- The tr command accepts i/p from standard input.
- This command takes two arguments which specify two character sets.
- The first character set is replaced by the equivalent member in the second character set.
- The -s option is used to squeeze several occurrences of a character to one character.

February 4, 2016

Proprietary and Confidential

• 98 •



### tr Command:

It accepts i/p from standard i/p.

Hence to give input to tr command we have to use i/p redirection operator.

Example:

```
$tr -s" " file1.txt
```

This is wrong command it will not take i/p from file1.txt.

The correct way of giving i/p is as follows:

```
$tr -s" " < file1.txt
```

## tr Command

- Example 1: To squeeze number of spaces by single space:  

```
$ tr -s " " < file1.txt
```
- Example 2: To convert small case into capital case:  

```
$ tr "[a-z]" "[A-Z]" < file1.txt
ONE
TWO
THREE
FOUR
```

February 1, 2016 | Proprietary and Confidential | 197

**tr Command:**

**Example:** To convert small case into capital case:

```
$ tr "[a-z]" "[A-Z]" < file1.txt
```

The tr command will replace a with A , b with B, and so on.

## more Command

- The more command, from the University of California, Berkeley, is a paging tool.
- The more command is used to view one page at a time. It is particularly useful for viewing large files.
- Syntax for more command is as follows:

```
more <options> <+linenumber> <+/pattern> <filename(s)>
```

- Example: To display file1.txt one screenful at a time

```
$ more file1.txt
```

February 1, 2016 | Proprietary and Confidential | 198



### more Command (Viewing a file one screen at a time):

This command from the University of California, Berkeley, is a paging tool – it can be used to view one page at a time. It is particularly useful for viewing large files.

Syntax of this is as follows:

```
more <options> <+linenumber> <+/pattern> <filename(s)>
```

There are a number of options available with the more command. Some of them are listed below:

Command	Description
Spacebar	Displays next screen
Ks	Skips K lines forward
Kf	Skips K screens forward
=	Displays current line number
:f	Displays current file name and line number
K:n	Skips to Kth next file specified on command line
K:p	Skips to Kth previous file specified on command line
/pattern	Searches forward for specified pattern
.	Repeats previous command
Q	Exits command

## chmod Command (Alter File Permissions)

- The chmod command is used to alter file permissions:
- Syntax:

```
chmod <category> <operation> <permission> <filenames>
```

Category	Operations	Attribute
u-user	+assigns permission	r-read
g-group	-remove permission	w-write
o-others	=assigns absolute permission	x-execute
a-all		

February 1, 2016 | Proprietary and Confidential | 49 of 99



In UNIX operating system every file is owned by the user who created that file. Only owner or Superuser can change the file permissions using chmod command.

### Superuser

In UNIX the superuser is responsible for system administration. **root** is the conventional name of the superuser who has all rights or permissions.

It is never good practice for anyone to use root as their normal user account, since simple typographical error in entering commands can cause major damage to the system. It is advisable to create a normal user account instead and then use the [su](#) command to switch to root user when necessary.

## chmod Command (Alter File Permissions)

### ➤ Example 1:

```
$ chmod u+x note
$ ls -l note
-rwx r-- r-- 1 ..... note
```

### ➤ Example 2:

```
$ chmod ugo+x note
$ ls -l note
-rwxr-xr-x ..... note
```

- When we use + symbol, the previous permissions will be retained and new permissions will be added.
- When we use = symbol, previous permissions will be overwritten.

February 1, 2016 | Proprietary and Confidential | 4/37



### chmod Command (Working with file permissions):

The **chmod** command is used to change the file permissions. Permissions can be specified in two ways:

- by using symbolic notation, or
- by using octal numbers

The table given below describes the category, operation, and attributes required by the **chmod** command:

Category	Operation	Attribute & value
u-user	+ assign permission	r-read (4)
g-group	- remove permission	w-write (2)
o-others	= assign absolute permission	x-execute (1)
A-all		

In **octal notation**, r is 4, w is 2, and x is 1. To give read and write permission to only user, we use number 600. 1<sup>st</sup> number indicates permissions for user(4+2), 2<sup>nd</sup> number for group, and 3<sup>rd</sup> number as others.

The following command will give read, write, and execute permissions to user(4+2+1=7), and read, write permissions to group(4+2), and execute (1) permission to others.

Let us consider some examples.

## chmod Command (Alter File Permissions)

### ➤ Example 3:

```
$ chmod u-x, go+r    note
$ chmod u+x          note      note1    note2
$ chmod o+wx          note
$ chmod ugo=r          note
```

February 1, 2016 | Proprietary and Confidential | - 41 -



### chmod Command (Working with file permissions):

#### Example 1:

```
$ chmod 761 file1.txt
$ ls -l file1.txt
-rw-rw-r-- 3 deshpavn group 60 Mar 29 10:43 file1.txt
```

#### Example 2:

```
$ chmod ugo-w file1.txt
$ ls -l file1.txt
-r--r--r-- 3 deshpavn group 60 Mar 29 10:43 file1.txt
```

#### Example 3:

```
$ chmod a+w file1.txt
$ ls -l file1.txt
-rw-rw-rw- 3 deshpavn group 60 Mar 29 10:43 file1.txt
```

#### Example 4:

```
$ chmod o-w,ug+x file1.txt file2.txt
$ ls -l file1.txt file2.txt
-rwxrwxr-- 3 deshpavn group 60 Mar 29 10:43 file1.txt
-rwxr-xr-- 1 deshpavn group 61 Mar 29 10:44 file2.txt
```

#### Example 5:

```
$ chmod 644 file1.txt file2.txt
$ ls -l file1.txt file2.txt
-rw-r--r-- 3 deshpavn group 60 Mar 29 10:43 file1.txt
-rw-r--r-- 1 deshpavn group 61 Mar 29 10:44 file2.txt
```

## chmod Command (Alter File Permissions)

### ➤ Octal notation:

- It describes both category and permission.
- It is similar to = operator (absolute assignment).
  - read permission: assigned value is 4
  - write permission: assigned value is 2
  - execute permission: assigned value is 1

### - Example t:

```
$ chmod 666 note
```

- It will assign read and write permission to all.

February 4, 2016 | Proprietary and Confidential | 42 /



## chmod Command (Working with file permissions):

### Setting Permissions:

The chmod command uses a string, which describes the permissions for a file, as an argument.

The permission description can be in the form of a number that is exactly three digits. Each digit of this number is a code for the permissions level of three types of people that might access this file:

- Owner
- Group (a group of users where owner is part of)
- Others (anyone else browsing around on the file system)

The value of each digit is set according to the rights that each of the types of people listed above have to manipulate that file.

Permissions are set according to numbers. **Read is 4**. **Write is 2**. **Execute is 1**. The sums of these numbers, give combinations of these permissions:

- 0 = no permissions whatsoever; this person cannot read, write, or execute the file
- 1 = execute only
- 2 = write only
- 3 = write and execute (1+2)
- 4 = read only
- 5 = read and execute (4+1)
- 6 = read and write (4+2)
- 7 = read and write and execute (4+2+1)

## chmod Command (Alter File Permissions)

### – Example 2:

```
$ chmod 777 note
```

\* It will assign all permissions to all.

### – Example 3:

```
$ chmod 753 note
```

February 4, 2016

Proprietary and Confidential

| + 43 +



### chmod Command (Working with file permissions):

Permissions are given using these digits in a sequence of three: one for owner, one for group, one for world.

Let us look at how I can make it impossible for anyone else to do anything with my apple.txt file but me:

```
$ chmod 700 apple.txt
```

If someone else tries to look into apple.txt, they get an error message:

```
$ cat apple.txt
cat: apple.txt: Permission denied
$
```

If I want other people to be able to read apple.txt, I would set the file permissions as shown below:

```
$ chmod 744 apple.txt
$
```

### Detecting File Permissions:

You can use the ls command with the -l option to show the file permissions set. For example, for apple.txt, I can do a change as follows:

```
$ ls -l apple.txt
-rw-r--r-- 1 december december 81 Feb 12 12:45 apple.txt
```

The sequence -rw-r--r-- tells the permissions set for the file apple.txt. The first - tells that apple.txt is a file. The next three letters, rwx, show that the owner has read, write, and execute permissions. Then the next three symbols, r-, show that the group permissions are read only. The final three symbols, r-, show that the others permissions are read only.

## mkdir Command

- The **mkdir** command creates a directory.

- Example: 1:

```
$ mkdir doc
```

- Example 2:

```
$ mkdir doc doc/example doc/data
```

- Example 3:

```
$ mkdir doc/example doc
```

- It will give error - Order important.

February 1, 2016Proprietary and Confidential– 64 –

### mkdir Command (Creating directory):

A single directory, or a number of subdirectories, can be created using the **mkdir** command. A directory will not get created if there is already another directory by the same name under the parent directory.

Besides, appropriate permissions will be required.

Example:

```
$ mkdir newdir1
$ ls -F
file1.txt
file2.txt
file3.txt
mail/
newdir1/
newfile.txt
testdir2/
$ mkdir newdir2 newdir2/subdir1 newdir2/subdir2
$ ls -l newdir2
total 4
drwxr-xr-x  2 deshpavn group      512 Mar 29 11:09 subdir1
drwxr-xr-x  2 deshpavn group      512 Mar 29 11:09 subdir2
```

## rmdir Command

- The **rmdir** command is used to remove directory.
- Only empty dir can be deleted.
- More than one dir can be deleted in a single command.
- Command should be executed from at least one level above in the hierarchy.

February 4, 2016 | Proprietary and Confidential | 48/50



### rmdir Command (Removing directory):

The **rmdir** command can be used to delete one or more directories.

Any directory can be deleted only if it is empty. It is important to note that the command needs to be issued from a directory, which is hierarchically above the directory to be deleted.

Example:

```
$ rmdir newdir1
```

## rmdir Command

➤ Example 1:

```
$ rmdir doc
```

➤ Example 2:

```
$ rmdir doc/example doc
```

➤ Example 3:

```
$ rmdir doc doc/example
```

— It will give error.

## Internal and External Commands:

### ➤ External commands

- A new process will be set up
- The file for external command should be available in BIN directory
- E.g – cat, ls , Shell scripts

### ➤ Internal commands

- shell's own built in statements, and commands
- No process is set up for such commands.
- E.g cd , echo

February 4, 2016 | Proprietary and Confidential | - 47 -



### Internal and External Commands:

The shell recognizes two types of commands – external and internal.

```
$ rmdir newdir1
```

#### External commands

- These are commands like cat, ls, and so on or utilities. Shell scripts also come under the category of external commands.
- A new process will be set up for the external commands.
- The file for external command should be available in BIN directory

#### Internal commands

- These are the shell's own built in statements, and commands such as cd, echo, and so on.
- No process is set up for such commands. For external command it is necessary that some commands are built into the shell itself and no process is set up. That is because it is very difficult or sometimes impossible to implement some commands as external commands.

## Summary

➤ **In this lesson, you have learnt:**

- UNIX organizes files in hierarchical manner.
- File access can be secured using different file permissions.
- <- Input Redirection
- > - Output Redirection
- 2> - Error Redirection
- chmod command is used to change file permissions.



## Review Questions

- Question 1: To copy all files with extension txt to mydir directory \_\_\_ command is used, if mydir is parent directory of current directory.
  - Option 1: cp \*.txt ..
  - Option 2: cp \*.txt ./mydir
  - Option 3: cp mydir \*.txt
- Question 2: 2> symbol is used as error redirection
  - True / False
- Question 3: cd . changes the directory to \_\_\_.
- Question 4: Which of the following command will give only read permission to all for file file1.txt?
  - Option 1: chmod a=r file1.txt
  - Option 2: chmod a+r file1.txt
  - Option 3: Chmod 666 file1.txt



## Review – Match the Following

1. To change directory to home directory

2. To remove all files with extension \*.dat

3. To display contents of file abc.txt

4. To create abc.txt file

a. rm \*.dat

b. cat <abc.txt

c. cat > abc.txt

d. cd

e. cd \

f. mkdir mydir



UNIX

Filters

UNIX  
Filters

February 1, 2010 | Proprietary and Confidential | 414



## Lesson Objectives

➤ **In this lesson, you will learn:**

- Filter commands in UNIX:
  - Simple Filters
  - Advance Filters



3.1: Simple Filters

## What is a Filter?

- Filters are central tools of the UNIX tool kit.
- Commands work as follows:
  - Accept some data as input.
  - Perform some manipulation on the inputted data.
  - Produce some output.
- Most of them work on set of records, with each field of a record delimited by a suitable delimiter.
- When used in combination, they can perform complex tasks too.

February 1, 2016 | Proprietary and Confidential | +3+

 Capgemini  
PERFORMING BUSINESS

### Simple Filters:

Filters are central tools of the UNIX tool kit. These commands accept some data as input, perform some manipulation on it and produce some output. Most of them work on set of records, with each field of a record delimited by a suitable delimiter. They serve as useful text manipulators. When used in combination, they can perform complex tasks as well. This lesson discusses some commonly used simple filters.

## head Command

➤ The head command, by default, will display the first 10 lines of a file.

– Example 1: To display first 10 lines from file employee:

```
$head employee
```

– Example 2: To display first 5 lines from file employee:

```
$head -5 employee
```

➤ Single command can be used to display lines from more than one file.

```
$ head -1 PuneEmp PKPEmp
```

Filehistory 1.3.000 | Proprietary and Confidential | Page 1



### The head Commands:

These are simple horizontal filters.

Using head, it is possible to display beginning of one or more lines from files. By default, the first 10 lines are displayed. Incase a numeric line count argument is specified, the command would display those many lines from the beginning of the file.

To display first 10 lines of the file bigfile, use the following syntax:

```
$ head bigfile
```

Output: cfile1.lst

```
cfi
e2.lst
errfile
file1.txt
file2.txt
file3.txt
mail
newdir1
newdir2
```

newfile.txt

To display first 3 lines of the file bigfile, use the following syntax:

```
$ head -3 bigfile
```

```
$ head -1 listing bigfile
```

Output:

```
cfi
e2.lst
errfile
```

Output: ==> listing <==  
cfi
e2.lst  
==> bigfile

<==

cfi
e2.lst Page 03-4

## tail Command

➤ The tail command is useful to display last few lines or characters of the file.

— Example 1: To display last ten lines from employee:

```
$tail employee
```

— Example 2: To display last seven lines:

```
$tail -7 employee
```

— Example 3: To display lines from the 10<sup>th</sup> line till end of the file:

```
$tail +10 employee
```

— Example 4: To display last 5 characters of the file:

```
$tail -5c employee
```

February 1, 2016 | Proprietary and Confidential | 45/46



### The tail Commands:

**Using the tail command:**

Using tail, the end of file can be displayed – default being last 10 lines.

```
$ tail -2 listing
```

<b>Output:</b>	result
	testdir2
	mail
	newdir1
	newdir2
	newfile.txt

```
$ tail +20 bigfile
```

**Output:** result  
Testdir2

To display last 6 characters from bigfile, use the following syntax:

```
$ tail -6c bigfile
```

le.txt

## cut Command

➤ The cut command retrieves selected fields from a file.

```
$ cut [options] <filename>
```

– Options :

- -c : selects columns specified by list
- -f : selects fields specified by list
- -d : field delimiter (default is tab)

February 1, 2016 Proprietary and Confidential 64



### cut Command:

You can slice the file vertically with the cut command, and paste laterally with the paste command.

The cut command can be used to retrieve specific column information from a file. In case of fixed record formats, the -c (columns) option can be used to specify column positions. If a delimiter has been used, -f (field) in conjunction with -d (delimiter) options can be used for retrieval. The default delimiter is tab.

```
$ cat bookDetails.lst
```

Output: 1001|Unix for You |375  
1002|Learning Unix |250  
1003|Unix Shell Programming |450  
1004|Unix Device Drivers |375  
1005|Advanced Unix Concepts |450

The following command will display first 4 characters followed by 31<sup>st</sup> to 35<sup>th</sup> characters:

```
$ cut -c1-4,31-35 bookDetails.lst
```

Output: 1001|375  
1002|250  
1003|450  
1004|375  
1005|450

## cut Command

- **Example 1:** To display 2<sup>nd</sup> and 3<sup>rd</sup> field from file bookDetails.lst:

```
$ cut -d"|" -f2,3 bookDetails.lst
```

- **Example 2:** To display characters from 1<sup>st</sup> to 4<sup>th</sup> and 31<sup>st</sup> to 35<sup>th</sup> from file bookDetails.lst :

```
$ cut -c1-4,31-35 bookDetails.lst
```

February 1, 2016 Proprietary and Confidential - 7 -



### cut Commands:

To display 2nd and 3rd field from file bookDetails.lst :

In the given command on the above slide, the -d option specifies field delimiter is |. Hence it will consider that in bookDetails file there are 3 fields separated by '|' character. The -f option will specify to display 2nd and 3rd field.

```
$ cut -d"|" -f2,3 bookDetails.lst
```

Output:	Unix for You	375
	Learning Unix	250
	Unix Shell Programming	450
	Unix Device Drivers	375
	Advanced Unix Concepts	450

## paste Command

- The paste command is used for horizontal merging of files.

```
$paste <file1><file2><Enter>
```

- Options : -d (Field delimiter)

- **Example 1:** To paste enum.lst and ename.lst files:

```
$ paste enum.lst ename.lst
```

- **Example 2:** To paste enum.lst and ename.lst files with '!' character as delimiter:

```
$ paste -d"!" enum.lst ename.lst
```

February 4, 2016 | Proprietary and Confidential | 8



### paste Command:

Several files can be pasted laterally, with specific delimiters, with the paste command.

```
$ cat enum.lst
```

Output: 1010  
2021  
3718  
4135  
5765

```
$ cat ename.lst
```

Output: Abc  
Zxc  
Qwe  
Jkl  
Uio

```
$ paste enum.lst ename.lst
```

Output: 1010 Abc  
2021 Zxc  
3718 Qwe  
4135 Jkl  
5765 Uio

```
$ paste -d"|" enum.lst ename.lst
```

Output: 1010|Abc  
2021|Zxc  
3718|Qwe  
4135|Jkl  
5765|Uio

## sort Command

- The sort command is useful to sort file in ascending order.

```
$sort <filename>
```

- Options are:

- **-r** : Reverse order
- **-n** : Numeric sort
- **-f** : Omit the difference between Upper and lower case alphabets
- **-t** : Specify delimiter
- **-k** : to specify fields as primary or secondary key

- Example:

```
$ sort -t"|" +1 bookDetails.lst
$sort -k3,3 -k2,2 employee
```

February 4, 2016 | Proprietary and Confidential | + 9 +



### sort Command:

#### Sorting a file with the sort command:

The sort command sorts a file (which may or may not contain fixed length records) on line by line basis. Default sorting is in the ascending ASCII order, which can be reversed by using the **-r** option.

Sorting can be done on one or more fields by specifying the delimiter using **-t** option. It is also possible to specify character positions within fields.

Using the **-m** option, it is also possible to merge any number of sorted files.

Since the sorting is done on the basis of ASCII collating sequence, incase of sorting of numbers, **-n** option needs to be used.

```
$ sort -t"|" +1 bookDetails.lst
```

Output:

1005	Advanced Unix Concepts	450
1002	Learning Unix	250
1004	Unix Device Drivers	375
1003	Unix Shell Programming	450
1001	Unix for You	375

To sort file employee on 3rd field as primary key and 2nd field as secondary key, use the following syntax:

```
$ sort -t"|" -k3,3 -k2,2 Employee
```

To consider only 3rd and 4th character from 2nd field for sorting employee file, use the following syntax:

```
$sort -t"|" -k2,3,2,4 employee
```

## uniq Command

- The **uniq** command fetches only one copy of redundant records and writes the same to standard output.
  - **-u** option: It selects only non-repeated lines.
  - **-d** option: It selects only one copy of repeated line.
  - **-c** option: It gives a count of occurrences.
- To find unique values, the file has to be sorted on that field.
  - Example: To find unique values from file duplist.lst

```
$ uniq duplist.lst
```

February 1, 2016 | Proprietary and Confidential | 101 |



### uniq Command:

The **uniq** command requires a sorted file as input. It fetches only one copy of redundant records and writes the same to standard output.

The **-u** option can be used to select only non-repeated lines, while the **-d** option can be used to select only one copy of repeated line. It is also possible to get a count of occurrences with the **-c** option.

#### Example 1:

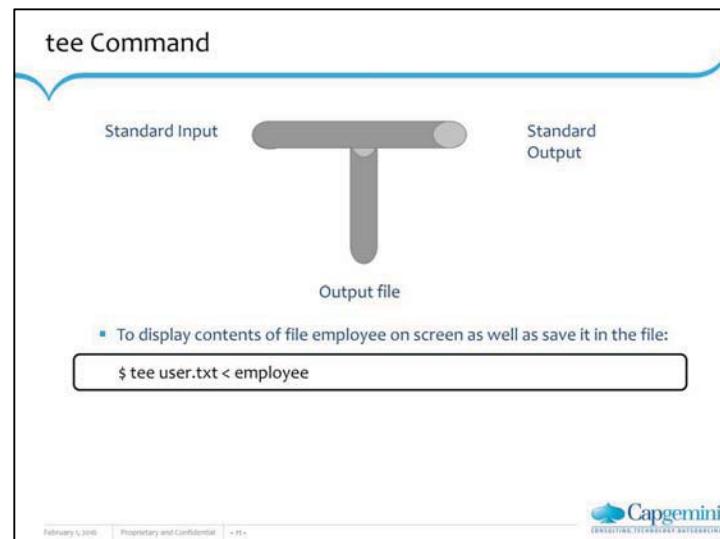
```
$ cat duplist.lst
```

Output: 1  
34  
30  
34  
1  
23  
23  
4

#### Example 2:

```
$ sort -n duplist.lst | uniq
```

Output: 1  
4  
23  
30  
34



#### tee Command:

The tee command copies the standard input to the standard output and also to the specified file.

If it is required to see the output on screen as well as to save output to a file, the **tee** command can be used. The **tee** command uses both standard input and standard output.

To display list of users and it's count both on screen, use the following:

```
Who | tee /dev/tty | wc -l
```

If we give command as **who|wc -l**, it will display only number of users on screen. However, in the above command **tee** will save the o/p to /dev/tty file which is terminal device file. Hence the o/p of **who** will be displayed on screen and also will be transferred as i/p to **wc** command.

In the following command, **sort** will sort the file and o/p will be transferred to **tee** command. It will display o/p on screen as well as store it in file **sorted\_file.txt**. The o/p will be given to **uniq** command which will give count of lines in the file. The **head** will find first 12 lines and store it in **top12.txt** file.

```
$sort somefile.txt | tee sorted_file.txt | uniq -c | head 12 > top12.txt
```

3.2: Advanced Filters

## find Command

➤ The find command locates files.

```
find <path list> <selection criteria> <action>
```

- Example 1: To locate the file named **.profile** starting at the root directory in the system **-print** specify the action:  
\$ find / -name .profile -print
- Example 2: To locate the file named **myfile** starting at the root directory in the system  
find / -type f -name "myfile" -print

February 1, 2016 | Proprietary and Confidential | 192

 Capgemini  
PERFECT IT. PERFECTION IT.

### find Command (locating files with find):

The **find** command is used to find files matching a certain set of selection criteria. The **find** command searches recursively in a hierarchy, and also for each pathname in the pathname-list (a list of one or more pathnames specified for searching).

The syntax of the find command is given in the following format:

```
$ find <path list> <selection criteria> <action>
```

The find command first looks at all the files in the directories specified in the path list. Subsequently, it matches the files for one or more selection criteria. Finally it takes action on those selected files.

```
$ find / -name .profile -print
```

The above command will locate the **.profile** files in the system.

```
$find . -name *stat
```

The above command will locate all file names ending with **stat**.

## grep Command

- The syntax for grep command is as follows:

```
grep <options> <pattern> <filename(s)>
```

- **Example:** The following example will search for the string Unix in the file **books.lst**. The lines which match the pattern will be displayed.

```
grep 'Unix' books.lst
```

February 1, 2016 | Proprietary and Confidential | - 13 -



### grep Command:

The **grep** command is used to locate a pattern / expression in a file / set of files. There are many options that are available for obtaining different types of outputs.

The syntax for the grep command is as follows:

```
grep <options> <pattern> <filename(s)>
```

The grep command scans the file(s) specified for the required pattern, and outputs the lines containing the pattern. Depending on the options used, appropriate output is printed. The grep command compulsorily requires a pattern to be specified, and the rest of the arguments are considered as file names in which the pattern has to be searched.

## grep Command

### ➤ Options of grep:

- c : It displays count of lines which match the pattern.
- n : It displays lines with the number of the line in the text file which match the pattern.
- v : It displays all lines which do not match pattern.
- i : It ignores case while matching pattern.
- w : It forces grep to select only those lines containing matches that form whole words

## grep Command

- **Example 1:** To print all lines containing "rose" regardless of case:

```
$grep -i rose flower.txt
```

- **Example 2:** To print all lines containing "rose" as a word:

```
$grep -w rose flower.txt
```

- **Example 3:** To print all lines not containing "rose":

```
$grep -v rose flower.txt
```

## grep Command

### ➤ Regular Expression:

Expression	Description
<code>^ (Caret)</code>	match expression at the start of a line, as in <code>^A</code> .
<code>\$ (Question)</code>	match expression at the end of a line, as in <code>A\$</code> .
<code>\ (Back Slash)</code>	turn off the special meaning of the next character, as in <code>\^</code> .
<code>[ ] (Brackets)</code>	match any one of the enclosed characters, as in <code>[aeiou]</code> . Use Hyphen <code>-</code> for a range, as in <code>[0-9]</code> .
<code>[^ ]</code>	match any one character except those enclosed in <code>[ ]</code> , as in <code>[^0-9]</code> .
<code>. (Period)</code>	match a single character of any value, except end of line.
<code>* (Asterisk)</code>	match zero or more of the preceding character or expression.
<code>{x,y}</code>	match <code>x</code> to <code>y</code> occurrences of the preceding.
<code>{x}</code>	match exactly <code>x</code> occurrences of the preceding.
<code>{x,}</code>	match <code>x</code> or more occurrences of the preceding.

## grep Command

### ➤ Examples of Regular Expression:

Example	Description
grep "smile" files	search files for lines with 'smile'
grep '^smile' files	'smile' at the start of a line
grep 'smile\$' files	'smile' at the end of a line
grep '^smile\$' files	lines containing only 'smile'
grep '^\$' files	lines starting with ''\$'; "\\" escapes the ^
grep '[Ss]mile' files	search for 'Smile' or 'smile'
grep 'B[oO][bB]' files	search for BOB, Bob, BOb or BoB
grep '^\$' files	search for blank lines
grep '[0-9][0-9]' file	search for pairs of numeric digits

February 4, 2016 | Proprietary and Confidential | +17 +



### grep Command: Some more examples:

Example	Description
grep '^From: '	list your mail
/usr/mail/SUSER	
grep '[a-zA-Z]'	any line with at least one letter
grep '[^a-zA-Z0-9]'	anything not a letter or number
grep '[0-9]{3}[-][0-9]{4}'	999-9999, like phone numbers
grep '^.\$'	lines with exactly one character
grep "smug"	'smug' within double quotes
grep ""smug""	'smug', with or without quotes
grep '^.'	any line that starts with a Period "."
grep '^.[a-z][a-z]'	line start with "." followed by 2 lowercase letters

## fgrep Command

- The fgrep command is similar to grep command.
- Syntax:

```
$fgrep [-e pattern_list] [-f pattern-file] [pattern] [Search file]
```

- The fgrep command is useful to search files for one or more patterns, which cannot be combined together.
- It does not use regular expressions. Instead, it does direct string comparison to find matching lines of text in the input.

February 1, 2016 | Proprietary and Confidential | + 18 +



### fgrep Command:

The **fgrep** command can also accept multiple patterns from command line as well as a file. However, it does not accept regular expressions – only fixed strings can be specified. The **fgrep** command is faster than **grep** and **egrep**, and should be used while using fixed strings.

The **egrep** and **fgrep** commands to some extent overcome the limitations of **grep**. However, the principal disadvantage of the grep family of filters is that there are no options available to identify fields. Also it is very difficult to search for an expression in a field. This is where the **awk** command is very useful.

## fgrep Command

### ➤ Options of fgrep command:

- **-e pattern\_list :**
    - It searches for a string in pattern-list.
  - **-f pattern-file :**
    - It takes the list of patterns from pattern-file.
  - **pattern**
    - It specifies a pattern to be used during the search for input.
    - It is same as grep command.
- E.g To search employee file for all patterns stored in mypattern file  
\$ fgrep -f mypattern employee.lst

February 5, 2016

Proprietary and Confidential

- 10 -



### fgrep Command:

Example using fgrep:

```
$ cat stud.lst
```

**Output:** R001|Pratik Sharma |425  
R002|Pallavi V. |398  
R003|Pratibha Aggarwal |400  
R004|Preeti Agrawal |390  
R005|Prerana Agarwal |421  
R006|Pranita agarwal |380

```
$cat mypattern
```

**Output:** Pratik  
Pratibha

```
$ fgrep -f mypattern stud.lst
```

**Output:** R001|Pratik Sharma |425  
R003|Pratibha Aggarwal |400

## egrep Command

- The egrep command works in a similar way. However, it uses extended regular expression matching.

- Syntax:

```
egrep [ -e pattern_list ] [-f file ] [ strings ] [ file ]
```

- Example: To find all lines with name “aggrawal” even though it is spelled differently:

```
$ egrep '[aA]gg?[ar]+wal' stud.lst
```

February 5, 2016 | Proprietary and Confidential | - 10 -



### egrep Command (extending grep):

The egrep command offers all the options of the grep command. In addition, it is possible to specify alternative patterns. The table given below gives the extended regular expression

Expression	Significance
ch+	Match with 1 or more occurrences of character ch
ch?	Match with 0 or more occurrences of character ch
exp1 exp2	Match with expressions exp1 or exp2
(a1 a2)a3	Match with expression a1a3 or a2a3

```
$ cat stud.lst
```

Some examples of using egrep are given:

Output:      R001|Pratik Sharma      |425  
                 R002|Pallavi V.      |398  
                 R003|Pratibha Aggarwal |400  
                 R004|Preeti Agrawal      |390  
                 R005|Prerana Agarwal      |421  
                 R006|Pranita agarwal      |380

```
$ egrep '[aA]gg?[ar]+wal' stud.lst
```

Output:      R003|Pratibha Aggarwal |400  
                 R004|Preeti Agrawal      |390  
                 R005|Prerana Agarwal      |421  
                 R006|Pranita agarwal      |380

## Summary

➤ **In this lesson, you have learnt:**

- The head and tail filter commands filter the file horizontally.
- The cut and paste commands filter the file vertically.
- -m option of sort command is used to merge two sorted files.
- The tee command helps us to send o/p to standard o/p as well as to file.
- grep, fgrep, and egrep commands use to search files for some pattern.



## Review Questions

- Question 1: \_\_\_ command to display directory listing on screen as well as store it in dirlist.lst.
- Question 2: \_\_\_ filter commands filter file vertically?
- Question 3: \_\_\_ filter commands filter file horizontally?



# UNIX

## Introduction to Bourne Shell

February 1, 2010 | Proprietary and Confidential | 414



## Lesson Objectives

➤ **To understand following topics:**

- Different shell types
- Working of shell
- Bourne shell metacharacters
- Shell redirection
- Command substitution



4.1: Shell Types

## Overview

➤ **Shell is:**

- The agency that sits between user and UNIX System
- Much more than command processor

➤ **Different shell types in the UNIX system are:**

- Bourne Shell	- sh
- K Shell	- ksh
- C Shell	- csh
- Restricted Shell	- rsh

February 1, 2010 | Proprietary and Confidential | +3+

 Capgemini  
PERPETUAL INNOVATION

### What is a Shell?

The shell is the agency that sits between the user and the Unix system. Whenever a command is issued, it is the shell that acts as the command interpreter. But the shell in Unix is much more than just a command processor. This chapter introduces some of the features of the Bourne Shell.

Shell is a process that creates an environment for you to work in.

When you login to UNIX machine you see a prompt because automatically a new shell process is started. This process will be terminated whenever user logs out.

How the command is executed

-The shell displays prompt and wait for you to enter a command

-When you type any command it scans the command line and processes all metacharacters to recreate simplified command. (e.g if the command is rm \*, then \* will be replaced by all file names in the current directory)

-Then it passes the command to kernel for execution

-And wait till execution of the command completes

-After command execution is complete, it displays prompt again to take up the next command

4.2.: Bourne Shell

## Introduction to Shell

➤ **Bourne Shell is:**

- Named after its founder Steve Bourne
- widely used - sh

➤ **C Shell is:**

- A product from the Univ. of California, Berkeley
- An advanced user interface with enhanced features - csh

➤ **Korn Shell is:**

- By David Korn of Bell Lab - ksh

February 1, 2016 | Proprietary and Confidential | 44

 Capgemini  
PERIODIC TABLE OF KNOWLEDGE

### Introduction to the shell:

Bourne Shell is one of the earliest and most widely used Unix shells. It is named after its founder Steve Bourne. The executable program sh in the /bin directory is the Bourne shell. There are other shells available on the Unix systems – popular amongst them are the C shell and the Korn shell.

The C shell is a product from the University of California, Berkeley. It has an advanced user interface with enhanced features. C shell, if present, is available as csh.

The Korn shell is from the Bell Labs. It is the most modern shell available currently, and is likely to become a standard. The Korn shell executable is ksh.

4.2: Bourne Shell

## Working of Shell

- **Executables in /bin directory**
  - sh indicates - Bourne Shell
  - csh if present indicates - C Shell
  - ksh if present indicated- Korn Shell

February 4, 2016 | Proprietary and Confidential | +5+

 Capgemini  
PERPETUAL INNOVATION EXPERIENCE

**Working of Shell:**

The shell is a Unix command – it is a program that starts when user logs in and terminates when user logs out. The job of the shell is to accept and interpret user requests (which are nothing but other Unix commands). Besides the shell is also programmable – this will be covered later.

The shell typically performs following activities in a cycle:

Issues a \$ prompt, and waits for user to enter a command

Scans and processes the command after user enters command

The command is passed on to the Kernel for execution and the shell waits for its conclusion

The \$ prompt appears so that user can enter next command. Hence the shell is in a continuous sleep – waking – waiting cycle

Some of the features of the shell are explored here.

### Working of Shell (contd..)

- **Continuous sleep-waking-waiting cycle**
- **Performs following activities:**
  - Issues a \$ prompt & waits for user to enter a command.
  - After user enters command, shell scans & processes the command.
  - The command is passed on to the Kernel for execution & the shell waits for its conclusion.
  - The \$ prompt appears so that the user can enter next command.

4.3: Metacharacters

## Description

➤ Following are the Bourne Shell metacharacters:

- \* : To match any number of characters
- ? : To match with a single character
- [] : Character class; Matching with any single character specified within []
- ! : To reverse matching criteria of character class
- \ : To remove special meaning attached to metacharacters
- ; : To give more than one command at the same prompt
- All redirection operators >, <, >> are also shell metacharacters

February 4, 2016 Proprietary and Confidential 47

 Capgemini  
INNOVATION PRACTICING

### Shell Metacharacters for pattern matching and combining commands:

Metacharacters are characters to which the shell attaches special meaning. The shell interprets these metacharacters and the command is rebuilt before it is passed on to the kernel.

#### Wildcards for Pattern Matching:

The wildcard \* is used to match any number of characters (including none). It however, does not match patterns beginning with a dot (.)

The wildcard ? matches a single character.

#### Examples:

emp\* : This would match all patterns that begin with emp, and may be followed by any number of characters (like emp, emppune, emptc, empseepz etc).

emp? : This would match all patterns that begin with emp and are followed by exactly one more character, which could be anything (like emp1, empa etc).

Patterns can be made more restrictive by using character class, represented by []. Any number of characters can be specified within the [], but the matching would occur for a single character within a class.

emp[abc]: This would match with patterns that begin with emp followed by a or b or c any one of it. (like empa, empb or empc)

Contd..

**Escaping with the Backslash:**

In order to remove the special meaning attached to metacharacters, the backslash can be used. For example, the expression emp\\* would match only with the pattern emp\*. In the given example \ removes special meaning of \* (i.e. 0 or more characters) and treat it as a character '\*'.

**Combining command using the semicolon:**

It is possible to give more than one command at the same prompt so that they will be executed in sequence one after the other. This is done with the character ; as in the example below:

**Example: Using the wild cards:**

```
$ ls -l file2.txt ; chmod u+x file2.txt ; ls -l file2.txt
-rwxr--r-- 1 deshpavn group 61 Mar 29 10:44 file2.txt
$
```

Here it is possible to assign permissions and subsequently check the same.

4.3: Redirections

## Shell Redirections

- Every Unix command has access to:
  - Standard input
  - Standard output
  - Standard error
- Shell can redirect I/p, o/p or error to any physical file using meta characters “<”, “>” & “2>”

February 4, 2016 | Proprietary and Confidential | 48

 Capgemini  
INNOVATION PRACTICING

**Redirections by Shell:**

Many commands work with character streams. The default is the keyboard for input (standard input, file number 0), and terminal for the output (standard output, file number 1). In case of any errors, the system messages get written to standard error (file number 2), which defaults to a terminal.

Unix treats each of these streams as files, and these files are available to every command executed by the shell. It is the shell's responsibility to assign sources and destinations for a command. The shell can also replace any of the standard files by a physical file, which it does with the help of meta characters for redirection. In order to take the standard input from a file, the character < is used.

```
$ wc < file1.txt
```

```
2 12 60
```

To redirect the output to a file, > is used. If outfile does not exist, it is first created; otherwise, the contents are overwritten. In order to append output to the existing contents, >> is used.

```
$ wc < file1.txt > result
```

```
$ cat result
```

```
2 12 60
```

### Shell Redirections (contd..)

➤ Examples:

```
$ ls > temp  
$ wc < file1.txt > result  
$ cat nonexistentfile 2> err
```

February 1, 2010 | Proprietary and Confidential | 4 / 10



Using redirections:

```
$ wc < cfile1.lst >> result  
$ cat result  
2 12 60  
4 4 8
```

As shown above, it is possible to combine redirection operators on a single command line. The order of the redirection symbols does not affect the working of the command.

Using redirections:

To redirect the standard error, 2> is used.

```
$ cat xyzfile  
cat: cannot open xyzfile: No such file or directory (error 2)  
$ cat xyzfile 2> errfile  
$ cat errfile  
cat: cannot open xyzfile: No such file or directory (error 2)  
$ cat xxfile 2> errfile  
$ cat errfile  
cat: cannot open xyzfile: No such file or directory (error 2)  
cat: cannot open xxfile: No such file or directory (error 2)
```

4.3: Redirections

## Building Block Primitives

- Pipe - allows stream of data to be passed between reader & writer process.
- O/p of first command is written into pipe and is input to the second command.
  - \$ who | wc -l
  - \$ ls | wc -l
  - \$ ls | wc -l > fcount
  - \$cat file1.txt | wc -l ( To display number of lines in file file1.txt)

February 4, 2016 Proprietary and Confidential 14 of 14

 Capgemini  
INNOVATION PRACTICING

### Connecting commands with Pipes:

The shell has a special operator called pipe (|), using which the output of one command can be sent as an input to another. The shell sets up the interconnection between commands. It eliminates the need of temporary files for storing intermediate results.

#### Creating a Pipeline:

```
$ who | wc -l  
8
```

When a sequence of commands are combined this way, a pipeline is said to have been formed. It is possible to combine redirection along with the pipe.

Combining pipe with redirection:

```
$ ls | wc -l > output  
$ cat output  
13
```

### Building Block Primitives (contd..)

- | - pipe symbol
- Any number of commands can be combined together to make a single command.

4.4: Command Substitution

## What is Command Substitution?

➤ Shell allows the argument of a command to be obtained from the output of another command:

- \$ cal 'date "+%m 20%"'
- January 2008
- Su Mo Tu We Th Fr Sa
- 1 2 3 4 5
- 6 7 8 9 10 11 12
- 13 14 15 16 17 18 19
- 20 21 22 23 24 25 26
- 27 28 29 30 31

February 1, 2010 | Proprietary and Confidential | 13+

 Capgemini  
INNOVATION PRACTICING

### What is Command Substitution?

The shell allows the argument of a command to be obtained from the output of another command – this feature is called command substitution. This is done by using a pair of backquotes. The backquoted command is executed first. The output of command is substituted in place of command. Then outer command will get executed.

#### Example:

```
$ cal `date "+%m 20%"'  
March 2008  
Su Mo Tu We Th Fr Sa  
1 2 3  
4 5 6 7 8 9 10  
11 12 13 14 15 16 17  
18 19 20 21 22 23 24  
25 26 27 28 29 30 31
```

4.5: Shell Script

## What is Shell Script?

- Group of commands that need to be executed frequently can be stored in a file, called as a shell script or a shell program.

```
$ cat script2.sh
echo 'Enter your name:
read uname
echo "Hi $uname"

```

```
O/P:$ script2.sh
Enter your name:
xyz
Hi xyz
```

- To assign values to variables, use the set command.

```
$ set uname="EveryOne"
$ echo Hi $uname
Hi EveryOne
```

Capgemini  
DEFINING YOUR FUTURE

### User defined shell variables and the echo command:

Since the shell is programmable, it is possible for users to define their own variables. No type is associated with the shell variables. The value of a variable is always string type. Shell variables are assigned values with the = operator, in the form variable=value. [There should be no spaces on either side of =]. Variable names can be combination of letters, digits & underscore, but the first character has to be a letter. Shell is sensitive to case. Even though the value is a string, if it contains only numerals, it can be used for numerical computation. In order to retrieve the value stored in a variable, the \$ sign needs to be used. The command echo can be used to display messages, as also the values of variables.

```
$ echo hi
hi
$ echo hi there
hi there
$ echo "hi there"
hi there
$ set msg="Hi there"
$ echo $msg
Hi there
```

4.6. eval command

## Command

- The eval command is used to assign values to variable
- Example: The following command will set \$day, \$month and \$year as separate variables that can then be used later in the script.

```
eval `date '+day=%d month=%m year=%Y'`
```

February 5, 2018 | Proprietary and Confidential | 15 | Capgemini CONSULTING TECHNOLOGY BUSINESS

### Using the shell eval command:

In shell scripts it is common to set variables using the output of a command, Example:

```
variable=`command`
```

The output from the command is normally a single value. If, however, the command returns multiple values you need to use some other program (e.g. awk ) to split the combined result into separate parts. Alternatively, you could pass an argument to the command to specify which result should be returned.

A neater solution is to use the shell's eval command. For example, say you wanted to return day, month and year from the date command. You'd have to write either:

```
day=`date +%d`  
month=`date +%m`  
year=`date +%Y`
```

```
result=`date '+%d %m %Y'`
```

or:

and then break up \$result with awk or cut for example. Instead you can write:

```
eval `date '+day=%d month=%m year=%Y'`
```

This will set \$day, \$month and \$year as separate variables that can then be used later in the script.

## Summary

- In UNIX different types of shells are available: CSH, KSH and Bourne Sh.
- Redirection operator can be used to redirect i/p or o/p to files or printer.
- Pipeline character can be used to send o/p of one command as i/p of another command.
- Group of commands that need to be executed frequently are stored in a file, called as a shell script.



### Review Questions

- Question 1: In shell, what are the different metacharacters available?
- Question 2: \_\_\_\_\_ symbol is used as output redirection.
- Question 3: \_\_\_\_\_ symbol is used as command substitution operator.



UNIX

**Instructor Notes:**

VI Editor

UNIX

VI Editor

February 1, 2010 Proprietary and Confidential 414



**Instructor Notes:**

## Lesson Objectives

- **Different modes of vi editor**
  - Input
  - Command
  - Esc mode
- **Input mode commands**
- **Vi editor – Save & Quit**
- **Navigation commands**
- **Paging functions**
- **Search and repeat commands**

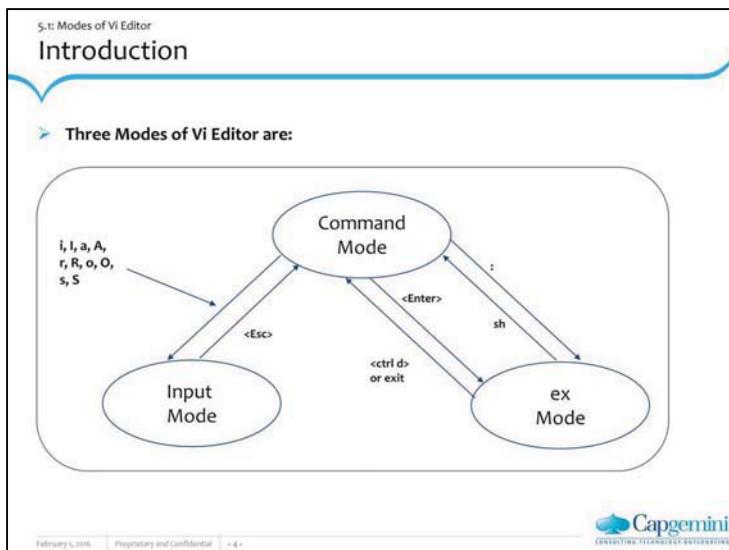


**Instructor Notes:**

### Lesson Objectives

- Vi editor – Other Features
- SED – Introduction to SED
- SED Commands



**Instructor Notes:****What is the vi Editor?**

Line editors, full screen editors and stream editors are all available on the Unix system. The editor “ed” was developed by Ken Thompson, and was the original editor that accompanied the Unix system. The line editor “ex” was created by William Joy, on the basis of “ed”. This chapter discusses the “vi” editor, which is a full screen editor, widely acknowledged as one of the most powerful editors available in any environment. The vi editor is also created by William Joy, and is in fact simply the visual mode of the line editor “ex”. It offers innumerable functions, but the terseness of its commands is considered to be a major handicap.

**Modes of vi editor:**

The vi editor works in 3 modes: Input, Command and ex mode. The relation between the three modes is depicted in the figure in the slide above.

**Command Mode, Input Mode, ex Mode**

**Instructor Notes:**

5.2: Input Mode Commands

## Contents

Command	Function
i	Insert text to left of cursor
I	Inserts text at beginning of line
a	Appends text to right of cursor
A	Appends text at the end of line
o	Opens line below
O	Opens line above

February 6, 2016 | Proprietary and Confidential | - 5 -


**What is the Input Mode?**

The Input mode is used to insert, append, replace or change text. A summary of input mode commands are given below:

Command	Function
• i	Inserts text to left of cursor
• I	Inserts text at beginning of line
• a	Appends text to right of cursor
• A	Appends text at end of line
• o	Opens line below
• O	Opens line above
• rch	Replaces single character at cursor with character ch
• R	Replaces text from cursor to right
• s	Replaces single character at cursor with any number of characters
• S	Replaces entire line

**Instructor Notes:****Contents (contd..)**

Command	Function
r	Replaces single character under cursor with character (no<Esc>)
R	Replace text from cursor to right

February 5, 2016 | Proprietary and Confidential | - 5 -



Add the notes here.

**Instructor Notes:**

5.3: VI Editor – Save & Quit

### Description

- **From input mode to command mode press <Esc>**
- **From command mode:**

To Save	:w
To Quit	:q
To Quit without saving	:q!
To save & quit	:wq
or	:x

February 6, 2016 | Proprietary and Confidential | - 7 -


**Saving and Quitting: The Last Line Mode**

vi uses the ZZ command to save and quit editor. The 'ex' mode, also referred to as last line mode, can also be used.

To switch from command mode to ex mode, a colon (:) is pressed, which appears as ex prompt in the bottom line. Any ex command can be entered at this prompt. Following commands can be used for saving and quitting from the ex mode:

<u>Command</u>	<u>Function</u>
• w	Write buffer into disk and remain in editing mode
• x	Save and quit the editor
• wq	Write and quit editor
• q	Quit editor

**The Repeat Factor**

A number can be prefixed to any command: most commands will interpret the instruction to repeat the command that many times. Hence this number is called as the repeat factor. For example, to insert a series of 30 asterisks in a line, 30\* can be used.

The repeat factor can be used with input as well as command mode.

**Instructor Notes:**

5.4: Navigation Commands

## Overview

Command	Function
h	Moves cursor left
j	Moves cursor down
k	Moves cursor up
l	Moves cursor right
^	Moves cursor to beginning of first
\$	Moves cursor to end of line
b	Moves cursor backwards to beginning of word
e	Moves cursor forward to end of word
w	Moves cursor forward to beginning of word

February 6, 2016 | Proprietary and Confidential | - 8 -

 Capgemini  
CONSULTING TECHNOLOGY OUTSOURCING

**Navigation (Cursor Movement commands):**

Command	Function
• h (or backspace)	Move cursor left
• j	Move cursor down
• k	Move cursor up
• l (or spacebar)	Move cursor right
• ^	Move cursor to beginning of first word of line
(no repeat factor)	
• o or	Move cursor to beginning of line (no repeat factor with o)
factor with o)	
• \$	Move cursor to end of line
• b	Move cursor back to beginning of word
• e	Move cursor forward to end of word
• w	Move cursor forward to beginning of word

**Instructor Notes:**

5.5: Paging Functions

### Details

Command	Function
<Control-f>	Full page forward
<Control-b>	Full page backward
<Control-d>	Half page forward
<Control-u>	Half page backward

February 6, 2016 | Proprietary and Confidential | + 9 +

**Paging Functions:**

Command	Function
• <Ctrl-f>	Full Page forward
• <Ctrl-b>	Full Page backward
• <Ctrl-d>	Half Page forward
• <Ctrl-u>	Half Page backward
• <Ctrl-l>	Redraw page screen (no repeat factor)

**Instructor Notes:**

5.6: Search and Repeat Commands

### Details

Commands	Functions
/pat	Searches forward for pat
?pat	Searches backward for pattern pat
n	Repeats search in the same direction along which the previous search was made (no repeat factor)
N	Repeats search in a direction opposite to that which the previous search was made (no repeat factor)

February 6, 2016 | Proprietary and Confidential | + 10 +

 Capgemini  
CONSULTING TECHNOLOGY OUTSOURCING

**Search and repeat commands:**

<u>Command</u>	<u>Function</u>
• /pat	Searches forward for pattern pat
• ?pat	Searches backward for pattern pat
• n	Repeats search in same direction as previous search (no repeat factor)
• N	Repeats search in opposite direction as previous search (no repeat factor)
• fch	Moves cursor forward to first occurrence of character ch in current line

**Instructor Notes:**

5.7: VI Editor – Other Features

## Using set command

- Set command is used to customize the behavior of the VI editor
- Some of the useful commands

Sr no.	Command	Description
1.	:set autoindent or :set ai	To set autoindent on
2	:set number or :set nu	To Displays lines with line numbers on the left side
3	:set smd or :set showmode	To show the actual mode of the editor that you are in at the bottom line.
4.	:set wm=x or :set wrapmargin=x	To automatically wrap the word on next line, x will be any nonzero value. (:set wm=2 sets the wrap margin to 2 characters)

February 6, 2016 Proprietary and Confidential | 15 |

 Capgemini  
CONSULTING TECHNOLOGY OUTSOURCING

To edit the behavior of vi editor. There are many options which can be used with :set command

To get the list of all options in set command use

**:set all**

**Some more commands**

Sr no.	Command	Description
1.	:set noautoindent or :set noai	To unset autoindentation
2.	:set nomesg	Turn off messages, so that nobody can bother you while using the editor.
3.	:set warn	To warns you if you have modified the file, but haven't saved it yet.
4.	:set tabstop=x or :set ts=x	To set the tabstop to x spaces (:set tabstop=8 the tab key will display 8 spaces)
5.	:set ignorecase or :set ic	To set ignore case by default while searching
6.	:set noignorecase or :set noic	To unset ignore case option
7.	:set linelimit=1048560	To set the maximum file size to edit
8.	:set list	To display hidden character like tabs or end of the line
9.	:set nolist	To display hide character like tabs or end of the line

**Instructor Notes:**

5.7: VI Editor – Other Features

### Details

- **Joining line:**
  - J - to join current line with next line
  - 4J - to join 4 lines from current line
- **Undo last Instruction - u**
- **Reverse all changes made to current line – U**
- **Using set command**

February 6, 2016 | Proprietary and Confidential | ~12 ~


**Operators**

vi uses a number of operators which can be used along with commands to perform complex editing functions. The most commonly used operators are:

d – delete  
c – change  
yy – yank (copy)  
! – filter to act on text

Operators can work only when combined with a command or itself. The operators also take a repeat factor.

**Some samples of using operators:**

Command	Function	
d\$ or D	Deletes from cursor to end of line	
5dd	Deletes five lines	
d/endif	Deletes from cursor up to the first occurrence of the string endif in the forward direction	
d30G	Deletes from cursor up to line number 30	
df.	Deletes from cursor to first occurrence of a dot	
c0	Changes from cursor to beginning of line	
c\$ or C	Changes from cursor to end of line	
3cw or c3w	Changes three words	

**Instructor Notes:**

### 5.8: SED – Introduction to SED

- SED("Stream EDitor") is a non-interactive stream oriented editor for filtering and transforming text.
- It reads input line by line, applying the operation which has been specified via the command line (or a sed script), and then outputs the line in a terminal or file.
- When to use SED?
  - To automate editing actions to be performed on one or more files.
  - To simplify the task of performing the same edits on multiple files.
  - To write conversion programs.

**Instructor Notes:**

5.9: SED Commands

## Invoking SED using Command Line

➤ **Syntax of SED Command**

```
sed options sed-script filename
```

- sed-script -> sed can use regular expressions for manipulating text on the input file.
- Options:
  - -n Suppress the default output.
  - -e Script is an edit command for sed . Used to specify multiple instructions by preceding with -e.

February 5, 2016 | Proprietary and Confidential | + 14 +

 Capgemini  
CONSULTING TECHNOLOGY OUTSOURCING

**Instructor Notes:**

5.9: SED Commands

## Invoking SED using script file

- Create a script file with long editing instructions to perform task on an input file.
- The sed command will then be used as:  
`sed -f scriptfile file`
- For Example,  
`sed -f sedsrc text`
  - sedsrc – script file contains editing instructions.
  - text – input file consists of data.

February 1, 2016 | Proprietary and Confidential | +15+



The sed command will then be used as:

`sed -f scriptfile file`

All the editing command that we need to execute are placed in a file, as shown:

```
$ cat sedsrc
s/WB/, West Bengal/
s/ BH/, Bihar/
s/ MH/, Maharashtra/
```

The following command reads all of the substitution commands in the sedsrc and applies them to each line in the input file “text”:

`$ sed -f sedsrc text`

```
Sidd B-1/250 Kalyani, West Bengal
Tito A-3/11 Thane, Maharashtra
Rayn D-17 LakeTown, West Bengal
Miter C/268 G.B.Road, Bihar
```

The above command will display the output in the Terminal.

The following command used for Redirecting the output to a file:

`$ sed -f sedsrc text > newtext`

**Instructor Notes:**

5.9: SED Commands

## Substitute Command

- **/s Command**
  - The substitute command changes all occurrences of the regular expression into a new value
- **Syntax:**  
`sed 's/old/new' file`
- **For Example:**  
`sed 's/Hi>Hello' data`  
 would substitute the occurrence of the word hi to hello in "data" file.

February 6, 2016 | Proprietary and Confidential | + 16 +



Let there be a text file called "text". The content of the file is as shown:

```
Sidd B-1/250 Kalyani WB
Tito A-3/11 Thane MH
Rayn D-17 LakeTown WB
Miter C/268 G.B.Road BH
```

The substitution command in sed:

```
$ sed 's/WB/WestBengal/' text
```

Two lines are affected by the instruction but in the above example all lines will be displayed. Enclosing the instruction in single quotes is not mandatory but its required if the substitution command contains spaces:

```
$ sed 's/ WB/, West Bengal/' text
```

**g option:** Used to make the command replace in all the instance of the word instead of first occurrence of the word in each input line.

```
$ sed 's/rat/cat/g' temp
```

```
cat cat
```

For example if we want to change 'rat' to 'cat' in lines that contain the word 'dog' we say:

```
$ sed '/dog/s/rat/cat/g' temp
```

```
$ sed 's/rat/cat/4'      # replaces only 4th instance in a line
```

## Instructor Notes:

5.9: SED Commands

## Multiple Instructions in SED Command

➤ There are three ways to specify multiple instructions on the command line:

- Separate instructions with semicolon
  - `sed 's/WB/, West Bengal/; s/ BH/, Bihar/' text`
- Precede each instruction by `-e`
  - `sed -e 's/WB/, West Bengal/' -e 's/ BH/, Bihar/' text`
- Use the multiline entry capability
  - `sed 's/WB/, West Bengal/  
s/ BH/, Bihar/' text`

February 6, 2016 Proprietary and Confidential | - 17 -

 Capgemini  
CONSULTING TECHNOLOGY OUTSOURCING

There are three ways to specify multiple instructions on the command line:

1. Separate instructions with semicolon  
`sed 's/WB/, West Bengal/; s/ BH/, Bihar/' text`

2. Precede each instruction by `-e`  
`sed -e 's/WB/, West Bengal/' -e 's/ BH/, Bihar/' text`

3. Use the multiline entry capability  
`sed 's/WB/, West Bengal/  
s/ BH/, Bihar/' text`

It is very easy to make mistake in the instruction or omit a required element.

```
$ sed 's/WB/, West Bengal' text
sed: command garbled: s/WB/, West Bengal
```

Notice the error message. Sed usually display any line that it cannot execute, but it does not tell what is wrong with the command. Here a slash at the end is missing.

## Instructor Notes:

5.9: SED Commands

## Other options

- **-n option**
  - Suppresses the display of all input lines with print command 'p'
  - For example
    - `$ sed -n 's/WB/WestBengal/p'` text - prints only the affected lines
- **d command**
  - Used to delete all lines and also to delete specific lines by either using regular expression or line number.
    - For Example: `$ sed d temp` # deletes all lines
- **-i option**
  - Used to substitute for the current given file. i.e the original file is changed.
- **\$ sed = temp # number each line of a file.**

February 6, 2016 | Proprietary and Confidential | +18+



The `-n` option suppresses the automatic output. When using this option each line needed to produce output must contain the print command, `p`.

```
$ sed -n 's/WB/WestBengal/p'
Sidd B-1/250 Kalyani WestBengal
Rayn D-17 LakeTown WestBengal
```

Here only the lines that were affected were printed.  
For printing only line 2 and 3, the command used is:

```
$ sed -n 2,3p text
```

If `-n` option is not present all the lines will get printed and line from 2 to 3 will get printed twice.  
d used to delete all lines and also to delete specific lines by either using regular expression or line number.

```
$ sed d temp # deletes all lines
$ sed 'd' temp #delete last line.
$ sed '1d' temp #delete first line.
$ sed '/^$/d' temp #delete all blank lines.
```

# number each line of a file (simple left alignment).  
\$ sed = temp

**Instructor Notes:**

5.9: SED Commands

## Other options

➤ -i option

- Used to edit content and save for the given file. In this case original file is changed.
  - Ex: \$sed -i 's/^/t' file
- If back up of original file is to be maintained then extension to -i option can be used. Extension used can be anything. It's just acts like another file which contains the original content.
  - Ex: \$sed -i.temp 's/^/t/' emp

February 1, 2016 Proprietary and Confidential 10/10

 Capgemini  
CONSULTING TECHNOLOGY OUTSOURCING

-i option is used to substitute for the given file. i.e. original file is actually changed. If any copy without changes is to be maintained then -i with some extension can be used as shown above.

For ex: \$sed -i 's/^/t/' emp

Above command will insert tab at beginning of the file. File 'emp' is changed. If you want to keep a copy of original file an extension can be used with option as shown below.

Ex: \$sed -i.temp 's/^/t/' emp

Now, emp.txt will have original file and emp is changed according to the command.

**Instructor Notes:**

5.9: SED Commands

## More Commands

Sl.No	Command	Description
1	<code>sed 10q temp</code>	print first 10 lines of file(emulates behavior of "head")
2	<code>sed q temp</code>	print first line of file(emulates "head -1")
3	<code>sed '\$!d' temp # method 1</code> <code>sed -n 'p' temp # method 2</code>	Prints last line of a file(emulates "tail -1")
4	<code>sed '\$!N;s;/n/ /' temp</code>	join pairs of lines side-by-side (like "paste")
5	<code>sed '\$!N; s/^(\.*\n)\n \1\$/; t; D' temp</code>	Delete all lines except duplicate lines (emulates "uniq -d").
6	<code>sed '1,10d' temp</code>	delete the first 10 lines of a file

February 5, 2016 | Proprietary and Confidential | >20 >

 Capgemini  
CONSULTING TECHNOLOGY OUTSOURCING

**Instructor Notes:**

## Summary

**➤ In vi editor:**

- esc key is used to change the mode.
- esc - \$ is used to move cursor at the end of the file.
- wq is used to write (save) and quit from the file.
- q! is used to quit without saving.

**➤ SED**

- Commands used to process the data.
  - Command line instruction
  - Script file based instruction

**Instructor Notes:****Review Questions**

- What command is used to copy the lines in vi editor?
- \_\_\_\_\_ command search for the pattern in vi editor in forward direction?
- What is the <control b> command used for?
- VI editor is stream Oriented?
  - True
  - False



## UNIX

Processes and Related Commands

February 1, 2010 | Proprietary and Confidential | 414



## Lesson Objectives

➤ **UNIX processes:**

- Parent and Child processes
- Process Status Command – ps
- Running processes in background mode
- Terminate process
- Process scheduling



6.1: UNIX Processes

## What is a Process?

➤ **Characteristics of processes:**

- Process is an instance of program in execution.
- Many processes can run at the same time.
- Processes are identified by the Process Identifier.
- PID is allocated by kernel.

February 4, 2016 | Proprietary and Confidential | +3+

 Capgemini  
INNOVATION PRACTICE

**UNIX Processes:**

When you execute a program on your UNIX system, the system creates a special environment for that program. This environment contains everything needed for the system to run the program as if no other program were running on the system.

**What is a Process?**

A process is an instance of a program in execution. When any executable file is executed, a process starts. It remains active while the program is executing. When the program terminates, the process dies. Generally the name of the process is the name of the executable.

Since Unix is a multi-tasking system, there can be many processes that run at the same time. A unique number called as the Process Identifier, PID, identifies each of these processes. The kernel allocates this PID, which is a number from 0 to 32767. It is the responsibility of the kernel to manage the processes – in terms of time allocated to process, its associated priorities and swapping etc.

6.1: UNIX Processes > 6.1.1: Parent and Child Processes

## Concepts

- On logging to a system, a process is set up due to execution of shell.
- Shell is the parent process for every other process setup due to the execution of commands.
- Every process, with the exception of PID 0 processes, has a parent process.
- Parent process waits for death of child process before resuming execution.

February 1, 2016 | Proprietary and Confidential | 44

 Capgemini  
INNOVATION PRACTICING

### Parent and Child Processes:

When a user logs on to the system, the kernel sets up a process – this is actually the process set up due to execution of sh command. This process remains active till the user logs off. The sh process is identified by the special variable \$\$. Any logging off and logging on again will result in an assignment of a different PID. The knowledge of this PID is required for controlling the activities of the terminal.

Any command written at the prompt is actually the standard input to the sh program. When a external command is run from the command line, the shell process spawns a new process for the command, which remains active till the command is active. The shell here is the parent process while the new process is the child process. The child process inherits the environment of the parent process. The child process can alter the operating environment that it has inherited, but the modified environment will not be available to the parent after the death of the child process.

Every process has a parent. The exception is the first process, with PID 0, which does not have a parent. This is set up when the system boots. It can be treated as analogous to the “root” in the file system.

Every process can have only one parent process. However, a process can spawn multiple processes. For example, when a pipeline is set up between 2 commands, the sh process would set up two processes for each of the commands.

On the completion of the child process, a signal is sent to the parent, and the control is reverted back to the parent process. However, if the parent process dies, the child processes automatically die.

6.1: UNIX Processes > 6.1.1: Parent and Child Processes

## Running a Command

**Is command: Steps for running a Unix command**

- The shell performs a fork. This creates a new process that the shell uses to run the ls program.
- The shell performs an exec of the ls program. This replaces the shell program and data with the program and data for ls and then starts running that new program.
- The ls program is loaded into the new process context, replacing the text and data of the shell.
- The ls program performs its task, listing the contents of the current directory .

February 1, 2016 | Proprietary and Confidential | 45+

 Capgemini  
INNOVATION PRACTICING

### Running a Command:

When you enter ls to look at the contents of your current working directory, UNIX does a series of things to create an environment for ls and run it.

Internally, Unix creates a process with the fork system call. This system call creates a copy of the process that invokes it. This child process is an image of the parent process, but with a new PID. The exec system call is then used – the parent process will overwrite the image of child with the copy of the program that is to be executed. After this, the parent executes the wait system call – parent will continue to wait till the death of the child. After this, parent can continue with its other activities.

Process is NOT set for all commands. The shell recognizes 2 types of commands – external and internal. External commands are commands like cat, ls etc or utilities. Shell scripts also come under the category of external commands. A process will be set up for the external commands. The internal commands are the shell's own built in statements, and commands like cd and echo etc. No process is set up for such commands.

It is necessary that some commands are built into the shell itself and no process is set up. That is because it is very difficult or sometimes impossible to implement some commands as external commands.

6.1: UNIX Processes > 6.1.2: Process Status Command - ps

## PS Command

➤ ps command displays characteristics of a process.

➤ Syntax:

ps [ option [ arguments ] ... ]

➤ Options:

- -f - full form
- -u - details of only users processes
- -a - all processes details
- -l — - detailed listing
- -e - system processes

ps

```
$ ps
  PID   TTY      TIME CMD
  599   ttys0  00:00:00 sh
  613   ttys0  00:00:00 ps
$ -
```

February 1, 2016 | Proprietary and Confidential | 6/4

 Capgemini

### PS Command:

The ps command can be used to display the characteristics of the processes. It has the knowledge of kernel built into it, using which it can read the process tables to get necessary information.

This command is one of the few Unix commands, which generates header information. It is also a highly variant command – the exact output obtained depends on the version of Unix as well as the hardware used.

When used without any options, ps command displays the following:

1. Process Id
2. The terminal with which the process is associated
3. Cumulative processor time of the process
4. The process name

6.1: UNIX Processes > 6.1.2: Process Status Command - ps

### Example

Output of ps -l command:

```
$ ps -l
F S  UID  PID  PPID  C PRI NI      ADDR  SZ  WCHAN  TTY  TIME C
MD 20 R  599  598  3 47 24 fbi1c8b0  60      -  tttyp0  00:00:00 s
h 20 0  201  625  599  1 48 24 fbi1ca08 164      -  tttyp0  00:00:00 p
s -
$ -
```

February 4, 2016 | Proprietary and Confidential | -7-

 Capgemini

**Process Status Command: Example:**

- **I** – It displays Long format
- **F** - Octal flags which are added together to give more information about the current status of a process(20 – process is loaded in primary memory: it has not been swapped out to disk)
- **S** - State of the process (0 – Process is running on a processor, R – Process is on run queue)
- **UID** - The Userid of the process owner ( login name is printed using -f option)
- **PID** - The process ID of the process (this number is needed to kill a process)
- **PPID** - The process ID of the parent process
- **C** - CPU usage by the process; combination of this value with nice value is used to calculate the priority
- **PRI** - The priority of the process (lower number mean lower priority)
- **NI** - The nice value of the process
- **ADDR** - The virtual address of the process entry in the process table

6.1: UNIX Processes > 6.1.3: Running Process in Background Mode

## Process in Background Mode

- **Processes can run in foreground or background mode.**
  - Only one process can run in foreground mode but multiple processes can run in background mode.
  - The processes, which do not require user intervention can run in background mode, e.g. sort, find.
  - To run a process in background, use & operator
    - + `$sort -o emp.lst emp.lst &`
- **nohup (no hangup) - permits execution of process even if user has logged off.**
  - `$nohup sort emp.lst &` (sends output to nohup.out)

February 1, 2016 | Proprietary and Confidential | 18

 Capgemini  
PERPETUAL INNOVATION EXPERIENCE

### Process in Background Mode:

It is possible to have only one job working in the foreground. But the other jobs can be made to run at the background. Background execution is a useful feature if more important jobs are to be run in the foreground and less important ones relegated to the background.

The & operator is provided by the shell to run a process in the background. On invoking a command terminated with &, shell immediately returns a PID for this number; and the shell is ready to accept another command even though the previous command is not terminated yet. It will be used as follows:

```
$ sort -o emp.lst emp.lst &
```

Even if a command is run in the background, it is possible that the output as well as error messages are still coming to the standard output. Hence, care should be taken to suitably redirect this output.

However, too many jobs should not be run in the background as significant deterioration of CPU performance can occur.

6.1: UNIX Processes > 6.1.4: Terminate Processes

## Kill Command

- Kill Command- Used to terminate a process
- Syntax :
  - kill [-signumber] pid ...
- Example:
  - \$kill 1005 (default signal 15) - kills job with pid 1005
  - \$kill -9 1005 - sure killing of job
  - \$kill 0 - kills all background process

February 1, 2016 Proprietary and Confidential 18

 Capgemini  
PERIODIC TABLE OF KNOWLEDGE

### The Kill Command:

It is possible to send signals to processes. When a process receives a signal, it can ignore it, terminate or do something else. Signals in Unix are identified by a number – each signal notifying that an event has occurred.

Other syntax

```
kill -s signame pid ...
kill -l [ exit_status ]
kill [-signame] pid ...
```

A signal number 15 is used by default by the kill command to terminate a process. The kill command takes one or more PID numbers as its arguments and facilitates premature termination of these processes. It is used as follows (assumed that process numbered 117 is being killed):

```
$ kill 117
```

It is possible that some programs simply ignore this command and continue normal execution. In that case, a “sure kill”, with signal number 9, has to be employed.

```
$ kill -9 117
```

In order to kill all processes except the login shell, an argument of 0 is passed to the kill command.

Of course, one can kill only one's own processes. Besides, some system processes cannot be killed at all.

6.2: Process Scheduling > 6.2.1: Overview of Process Scheduling

## Details

**Scheduling Policy:**

- time-sharing technique
- Several processes are allowed to run "concurrently," which means that the CPU time is roughly divided into "slices," one for each runnable process.
- The scheduling policy is also based on process priority
- In UNIX, process priority is dynamic.

February 1, 2010 | Proprietary and Confidential | 4 of 10 | Capgemini

 Capgemini  
INNOVATION THROUGH INTEGRATION

6.2: Process Scheduling > 6.2.1: Overview of Process Scheduling

## Continued...

➤ Processes are traditionally classified as "I/O-bound" or "CPU-bound."

- **I/O-bound Processes:**  
Make heavy use of I/O devices and spend much time waiting for I/O operations to complete.
- **CPU-bound Processes:**  
Are number-crunching applications that require a lot of CPU time.

February 1, 2016 | Proprietary and Confidential | 19 of 19

 Capgemini  
INNOVATION PREDICTIVE PERFORMANCE

### Overview of Process Scheduling:

When speaking about scheduling, processes are traditionally classified as "I/O-bound" or "CPU-bound."

- **I/O-bound Processes:** They make heavy use of I/O devices and spend much time waiting for I/O operations to complete.
- **CPU-bound Processes:** They are number-crunching applications that require a lot of CPU time.

## Continued...

## ➤ Processes can also be classified as:

- **Interactive processes:**  
These interact constantly with their users, and therefore spend a lot of time waiting for key presses and mouse operations.
- **Batch processes:**  
These do not need user interaction, and hence they often run in the background.
- **Real-time processes:**
  - Should never be blocked by lower-priority processes.
  - Should have a short response time.

February 5, 2016

Proprietary and Confidential

+12 /

**Overview of Process Scheduling (contd.):**

An alternative classification distinguishes three classes of processes:

**Interactive processes**

These interact constantly with their users, and therefore spend a lot of time waiting for key presses and mouse operations. When input is received, the process must be woken up quickly, or the user will find the system to be unresponsive. Typically, the average delay must fall between 50 and 150 ms. The variance of such delay must also be bounded, or the user will find the system to be erratic. Typical interactive programs are command shells, text editors, and graphical applications.

**Batch processes**

These do not need user interaction, and hence they often run in the background. Since such processes do not need to be very responsive, they are often penalized by the scheduler. Typical batch programs are programming language compilers, database search engines, and scientific computations.

**Real-time processes:**

These have very strong scheduling requirements. Such processes should never be blocked by lower-priority processes, they should have a short response time and, most important, such response time should have a minimum variance. Typical real-time programs are video and sound applications, robot controllers, and programs that collect data from physical sensors.

The two classifications we just offered are somewhat independent. For instance, a batch process can be either I/O-bound (e.g., a database server) or CPU-bound (e.g., an image-rendering program). While in UNIX real-time programs are explicitly recognized as such by the scheduling algorithm, there is no way to distinguish between interactive and batch programs. In order to offer a good response time to interactive applications, UNIX (like all Unix kernels) implicitly favors I/O-bound processes over CPU-bound ones.

6.2: Process scheduling > 6.2.1: Overview of Process Scheduling

## nice and wait command

- nice - runs a program with modified scheduling priority.
- Syntax:  
`nice [OPTION] [COMMAND [ARG]...]`
  - \$ nice cat chap?? | nice wc -l > wclist &

- Wait - waits for child process to complete.
- Syntax:  
`wait [ process id... ]`
  - \$wait 138 - waits for background job with pid 138

February 1, 2016 | Proprietary and Confidential | 13 / 14

 Capgemini  
INNOVATION THROUGH INSPIRATION

### nice and wait Command:

All processes on Unix are usually executed with equal priority. In order to reduce the priority of a job, the command needs to be prefixed with nice. By default, nice reduces the priority of any process by 10 units. The amount of reduction can also be specified as an argument (value from 0 to 19) to the nice command. Incase of commands in a pipeline, to reduce the priority of all commands in the pipeline, nice needs to be used in each element in the pipeline.

```
$ nice cat chap?? | nice wc -l > wclist &
```

The wait command can be used to wait for the completion of a background process. This is a built-in shell command – no process is spawned for this command. It sends the shell into a wait state so that it can acknowledge the death of child processes.

## cron

- A system daemon which performs a specific task at regular intervals
- The command and schedule information is kept in the directory /var/spool/cron/crontabs or in /usr/spool/cron/crontabs.
- Each user has a crontab file. cron wakes up periodically and executes any job that are scheduled for that minute.
- Only users who are listed in /etc/cron.allow or not listed in cron.deny can make an entry in the crontab.
- **Crontab <filename> -> used to make an entry in the crontab file.**
  - where the file contains the commands to execute

MIN	HOUR	DOM	MOY	DOW	COMMAND
(0-59)	(0-23)	(1-31)	(1-12)	(0-6)	—
* 0	18	* * *	*	/home/gather	



You can view the contents of the global cron file by logging in as **root** and typing:

**# crontab -l**

The cron process executes the jobs it contains at the times that are specified. Each "cron job" is composed of six fields. Fields one to five contain clock information which specify when the command (given in the sixth field) should be executed.

**Minute Hour Day Month Day Task**

**Minute** = Minute of the hour, 00 to 59. \* Will indicate every minute (details later)

**Hour** = Hour of the day in 24-hour format, 00 to 23. \* Will indicate every hour (details later)

**Day** = Day of the month, 1 to 31. \* Will indicate every day (details later)

**Month** = Month of the year, 1 to 12. \* Will indicate every month (details later)

**Day** = Day of the week, 3 chars - sun, mon, tue, or numeric (0=sun, 1=mon etc).... \* Will indicate every day (details later)

**Task** = The command you want to execute

Note: each of the above must be separated by at least 1 space.

## Summary

- Unix processes
- Process related commands
  - ps
  - nohup
  - wait
  - kill
  - nice
- Background processes



## Review Questions

➤ **Complete The Following :**

- A unique number called the \_\_\_\_\_ identifies each process.
- Processes using heavy i/o are called as \_\_\_\_\_



➤ **True / False**

- A signal number of 9 is used, by default, by the kill command to terminate a process.
- You can kill any process, including the system process, using the kill command.

## UNIX

AWK Programming

February 1, 2010 Proprietary and Confidential 414



## Lesson Objectives

- **At the end of the session you is able to understand:**
- How to write simple awk scripts.



7.1: Advanced Filter - awk

## Introduction

➤ **AWK**

- Based on pattern matching and performing action.
- We have seen how grep uses pattern.
- Limitations of the grep family are:
  - No options to identify and work with fields.
  - Output formatting, computations etc. is not possible.
  - Extremely difficult to specify patterns or regular expression always.
- AWK overcomes all these drawbacks.

February 6, 2016 | Proprietary and Confidential | - 3 -

 Capgemini  
CONSULTING TECHNOLOGY OUTSOURCING

The **awk** command, named after its authors Aho, Weinberger and Kernighan, is one of the most powerful utilities for text manipulation. It combines features of many other filters. It can access, transform and format individual fields in a record – it is also called as a report writer. It can accept regular expressions for pattern matching, has “C” type programming constructs, variables and in-built functions. In fact, awk is nearly as powerful as any other programming language. However, awk programs are generally slow, if any alternative commands are available to do the same use them rather than using AWK.

7.1: Advanced Filter - awk

## Introduction

➤ **AWK**

- Named after Aho, Weinberger, Kernigham.
- As powerful as any programming language.
- Can access, transform and format individual fields in records.

February 6, 2016 | Proprietary and Confidential | - 4 -

 Capgemini  
CONSULTING TECHNOLOGY OUTSOURCING

Filters accept some data as input, perform some manipulation on it and produce some output. Some simple filters have been discussed in the previous chapters. This chapter discusses advanced filter `-awk`.

7.1: Advanced Filter - awk

## Contents

➤ **Syntax:**

- awk <options> ‘line specifier {action}’ <files>
- Example:
  - awk '{print \$0 }' emp.data
- This program prints the entire line from the file *emp.data*.
- \$0 refers to the entire line from the file *emp.data*.

February 6, 2016 | Proprietary and Confidential | + 5 +

 Capgemini  
CONSULTING TECHNOLOGY OUTSOURCING

### Advanced Filter awk.

The general syntax of the awk command is:

awk <options> ‘line specifier {action}’ <file(s)>

#### Simple awk Filtering

Following is an example of a simple awk command:

It prints all lines from file *books.lst* in which pattern ‘Computer’ is found.

```
$ awk '/Computer/ {print}' books.lst
```

#### Output:

1001 Learning Unix	Computers	01/01/1998	575
1003 XML Unleashed	Computers	20/02/2000	398
1004 Unix Device Drivers	Computers	09/08/1995	650
1007 Unix Shell Programming	Computers	03/02/1993	536

7.1: Advanced Filter - awk

## AWK variables

➤ **Variable List:**

- **\$0:** Contains contents of the full current record.
- **\$1..\$n:** Holds contents of individual fields in the current record.
- **NF:** Contains number of fields in the input record.
- **NR:** Contains number of record processed so far.
- **FS:** Holds the field separator. Default is space or tab.
- **OFS:** Holds the output field separator.
- **RS:** Holds the input record separator. Default is a new line.
- **FILENAME:** Holds the input file name.
- **ARGC:** Holds the number of Arguments on Command line
- **ARGV:** The list of arguments

February 5, 2016 | Proprietary and Confidential | - 5 -

 Capgemini  
CONSULTING TECHNOLOGY OUTSOURCING

7.1: Advanced Filter - awk

## Example

- **awk '{ print \$1 \$2 \$3 }' emp.data**
  - This prints the *first*, *second* and *third* column from file *emp.data*.
  
- **awk '{ print }' emp.data**
  - Prints all lines (all the fields) from file *emp.data*.

7.2 AWK variables

## Overview

- Line specifier and action option are optional, either of them needs to be specified.
- If line specifier is not specified, it indicates that all lines are to be selected.
- {action} omitted, indicates print (default).
- Fields are identified by special variable \$1, \$2, ... ;
- Default delimiter is a contiguous string of spaces.
- Explicit delimiter can be specified using -F option
  - Example: awk -F " " 'sales/{print \$3, \$4}' emp.lst
- Regular expression of egrep can be used to specify the pattern.

February 4, 2016 | Proprietary and Confidential | 8 / 8

 Capgemini  
CONSULTING TECHNOLOGY OUTSOURCING

The line specifier uses a context address to specify the lines that need to be taken up for processing in the action section. If the line specifier is missing, then the action is applicable to all lines of the file.

In the action part, statement {print} indicates that selected lines are to be printed. The statement {print} is equivalent to {print \$0} - \$0 being the variable for the entire line. The awk command is also capable of breaking each line into fields – each field is identified as \$1, \$2 etc.

For the purpose of identification of fields, awk uses a contiguous string of spaces as field delimiter. However, it is possible to use a different delimiter. This is specified using the -F option in awk.

```
$ awk -F"|" '/Computer/ {print $2,$5}' books.lst
Output:
Learning Unix      575
XML Unleashed     398
Unix Device Drivers 650
Unix Shell Programming 536
```

For pattern matching, awk uses regular expressions of egrep variety.

```
$ awk -F"|" '/XML|Unix/ {print $2,$5}' books.lst
Output:
Learning Unix      575
XML Unleashed     398
Unix Device Drivers 650
XML Applications  630
Unix Shell Programming 536
```

It is possible to specify line numbers in file using the inbuilt NR variable. Also, awk can use the C-like printf statement to format the output.

```
$ awk -F"|" '$1=="1002" {printf "%2d,%-20s",NR,$2}' books.lst
Output:
2,Moby Dick
```

The -f option of awk is useful if you wish to store the line specifier or action in a separate file.

7.2 AWK variables

## Examples

- **awk '\$3 > 0 { print \$1, \$2 \* \$3 }' emp.data**
  - Checks for \$3 (third field) value. If it is greater than 0, then it prints the first column and the multiplication of the second and the third columns.

February 1, 2016 Proprietary and Confidential | + 9 +



The logical operators || (or) and && (and) are used by the awk command to combine conditions in the line specifier. A relational operator can be used in the line specifier, also in the action component.

The operators == (equal) and != (not equal) can handle only fixed length strings and not regular expressions. To match regular expressions, ~ (match) and !~ (negate) are used. The characters ^ and \$ can be used for looking for a pattern in the beginning and end of field.

To work with numbers, operators like < (less than), > (greater than), <= (less than or equal), >= (greater than or equal), == (equal) and != (not equal) can be used.

It is possible to perform computations on numbers using C like arithmetic operators (+, -, \*, /, %, ++, --, += etc).

**§ awk -F"\|"/'Unix/ && \$5 < 600 {printf "%s,%d\n",\$2,\$5}' books.lst**

**Output:**

```
Learning Unix      575
Unix Shell Programming ,536
$ awk -F"\|"/'$2=="Learning Unix"/' books.lst
$ awk -F"\|"/'$2~/Learning Unix/' books.lst
1001|Learning Unix      |Computers   |01/01/1998| 575
$ awk -F"\|"/'$5<500 {
> cnt=cnt+1
> printf "%d %s\n",cnt,$2}' books.lst
1 Moby Dick
2 XML Unleashed
```

7.2 AWK variables

## Examples

➤ Line numbers can be selected using NR built-in variable.

- awk -F "!" "NR ==3, NR ==6 {print NR, \$0}" emp.lst
- awk '{ print NF, \$1, NR }' emp.data
- awk '\$3 == 0' emp.data
- awk '{ print NR, \$0 }' emp.data
- awk '\$1 == "Susie"' emp.data

February 6, 2016 Proprietary and Confidential + 10 +

 Capgemini  
CONSULTING TECHNOLOGY OUTSOURCING

**awk -F "!" "NR ==3, NR ==6 {print NR, \$0}" emp.lst**

In this example, NR represents record number. It prints records from *third record* to *sixth record*.

-F is use to specify field separator.

**awk '{ print NF, \$1, NR }' emp.data**

This prints number of fields, contents of field 1 and record number for all records.

**awk '\$3 == 0' emp.data**

It prints all lines in which the value in the third field is 0.

**awk '{ print NR, \$0 }' emp.data**

It will print record number and record for all records

**awk '\$1 == "Susie"' emp.data**

It prints all lines in which the value in the first field is Susie.

7.3 Logical and Relational operators

## Logical and Relational operator

- Logical Operator &&, ||

```
$awk -F "|" '$3 == "director" || $3 == "chairman" {printf "%-20s", $2}' emp.lst
```

- Relational Operators : <, <=, ==, !=, >=, >, ~, !~

```
$awk -F "|" '$6 > 7500 {printf "%20s", $2}' emp.lst
$awk -F "|" '$3 == "director" || $6 > 7500 {print $0}' emp.lst
```

February 1, 2016 Proprietary and Confidential | 15 |

 Capgemini  
CONSULTING TECHNOLOGY OUTSOURCING

The logical operators || (or) and && (and) are used by the awk command to combine conditions in the line specifier. A relational operator can be used in the line specifier, also in the action component.

The operators == (equal) and != (not equal) can handle only fixed length strings and not regular expressions. To match regular expressions, ~ (match) and !~ (negate) are used. The characters ^ and \$ can be used for looking for a pattern in the beginning and end of field.

For working with numbers, operators like < (less than), > (greater than), <= (less than or equal), >= (greater than or equal), == (equal) and != (not equal) can be used.

It is possible to perform computations on numbers using C like arithmetic operators (+, -, \*, /, %, ++, --, += etc).

Print all lines with Unix pattern in the line and value of 5 th field should be < 600:

```
$ awk -F"|" '/Unix/ && $5 < 600 {printf "%s,%d\n", $2, $5}' books.lst
```

#### Output

```
Learning Unix ,575
Unix Shell Programming ,536
$ awk -F"|" '$2 == "Learning Unix"' books.lst
$ awk -F"|" '$2 ~ /Learning Unix/' books.lst
1001|Learning Unix |Computers |01/01/1998| 575
$ awk -F"|" '$5 < 500 {
> cnt=cnt+1
> print $0
1 Moby Dick
2 XML Unleashed
```

Example of awk command using relational, logical expressions and computations.

7.3 Logical and Relational operators

## Logical and Relational operators

- **== tries to find perfect match.**
  - String may have trailing spaces.
  - To overcome this you can use “-” and “!-” (match and negate of match) with the regular expression.
  - Example:

```
$awk -F "!" '$2~/director/| !$2~/g.m/{printf $0}' emp.lst  
$3 ~/^g.m/ # Beginning with g.m
```

February 5, 2016

Proprietary and Confidential

&lt; 12 &gt;



7.4 Arithmetic operators

## Arithmetic operators

➤ Computation on numbers is done:

- +, -, \*, /, % operator available.
- No type-declaration for variables.
- Variables are initialized to zero.

```
$awk -F "!" '$3 == "director" || $6>7500 {  
    >kount = kount+1  
    >printf "%3d%-20s\n", kount,$2}' emp.lst
```

➤ Can also use C constructs like kount++, Kount+=sum etc.

February 5, 2016

Proprietary and Confidential

+13 +



7.5 awk command  
**awk command**

➤ **-f option**

- awk program can be written in a separate file and used in awk.
  - Example:

```
$awk -F “|” -f emp.awk emp.lst
```

- Single quoted contents are written in this file without quotes.

```
$cat emp.awk
```

```
$3 == “Director” {print $2,$4,$5}  
$
```

7.6: BEGIN and END Section

## BEGIN and END Section

➤ **BEGIN and END Section:**

- Format: (i) BEGIN {action} (ii) END {action}

```
awk<options> 'BEGIN {action}
line specifier {action}
END {action}' <files>
```

February 6, 2016 | Proprietary and Confidential | + 15 +

 Capgemini  
CONSULTING TECHNOLOGY OUTSOURCING

### BEGIN and END Sections

In case, something is to be printed before processing the first line begins, BEGIN section can be used. Similarly, to print at the end of processing, END section can be used. These sections are optional. When present, they are delimited by the body of the awk program.

Normally if you want to print any header lines or want to set field separator then use BEGIN section

And if you want to display total or any summarized information at the end of the report then use END section

## 7.6 BEGIN and END Section

## Example

```
$cat emp.awk
BEGIN { printf "\n\tEmployee details \n\n"
}
$6>7500{
# increment sr. no. and sum salary
kount++; tot+=$6
printf "%d%-20s%d\n", kount, $2, $6
}
END { printf "\n The Avg. Sal. Is %6d\n", tot/kount
}
$awk -F "|" -f emp.awk emp.lst
```

February 6, 2016

Proprietary and Confidential

+ 16 +



```
$ cat awk1.awk
BEGIN {
printf "\n\t Unix Book Details \n\n"
}
$2~/Unix/ {
cnt++; tot+=$2
printf "%d %-15s\n",cnt,$2
}
END {
printf "\n Total Cost is %d\n",tot
}
$ awk -F "|" -f awk1.awk books.lst
Output:
        Unix Book Details
1 Learning Unix
2 Unix Device Drivers
3 Unix Shell Programming
Total Cost is 1761
```

## 7.6 BEGIN and END Section

## Example

## Example 1:

```
awk '
BEGIN
{
    print "NAME RATE HOURS";
    print ""
}
{
    print
}
' emp.data
```

## Example 2:

```
awk '
END
{
    print NR, "employees"
}
' emp.data
O/P : 6 employees
```

February 6, 2016

Proprietary and Confidential

+17+

**Example 1:**

In this example It will perform action enclosed in BEGIN section (It prints headings) and then it will perform print action on all lines in the file emp.dat (because no line specifier is mentioned)

**Example 2 :**

In this example BEGIN section is not given. The main action part is also empty hence no action is performed on all the records and then it will execute End section which prints last record number which will give you total number of records.

7.7 Positional parameters and shell variable

## Positional parameters and shell variable

- Requires entire awk command to be in the shell script.
- Differentiate positional parameter and field identifier
  - Positional parameter should be single-quoted in an awk program.
  - Example: \$3 > '\$1'

February 6, 2016 Proprietary and Confidential + 18 +



### Positional Parameters and Shell Variables

It is possible to store an entire awk command into a file as a shell script, and pass parameters as arguments to the shell script. The shell will identify these arguments as \$1, \$2 etc based on the order of their occurrence on the command line.

Within awk, since \$1, \$2 etc. indicate fields of data file, it is necessary to distinguish between the positional parameters and field identifiers. This is done by using single quotes for the positional parameters used by awk.

```
$ cat awk2.awk
awk -F"\t" $5>$1' {
cnt++
printf "%d %s %d\n",cnt,$2,$5
END {
printf "\n No. of books costing more than specified amount is:%d\n",cnt
}' books.ls1
$ awk2.awk 600
Output:
1 Unix Device Drivers    650
2 Complete Works:Sherlock H 1290
3 XML Applications    630
No. of books costing more than specified amount is:3
$ awk2.awk 1000
1 Complete Works:Sherlock H 1290
No. of books costing more than specified amount is:1
(Example of awk command using positional parameter.)
```

7.8 Built in functions

## Numeric Functions

- **int(x)**
  - Returns integer value of x.
- **sqrt(x)**
  - Returns square root of x.
- **index(s1,s2)**
  - Returns the position of string s2 in s1.
- **length()**
  - Returns length of the argument.

February 5, 2016 | Proprietary and Confidential | + 19 +

 Capgemini  
CONSULTING TECHNOLOGY OUTSOURCING

7.8 Built in functions

## String

- **substr(s1,s2,s3)**
  - Returns portion of string of length - s3, starting position s2 in string s1.
- **split(s,a)**
  - Split the string s into array a.
  - Optionally it returns the number of fields.
    - Example:

```
awk -F":" 'split{$5,arr,":";print "20arr[3]arr[2]arr[1]}' emp.lst
```

- Splits 5th Field (date) into an array and prints in form "YYYYMMDD".

```
awk -F":" 'substr($5,7,2) > 45 && substr($5,7,2) < 52' emp.lst
```

- Retrieves those born between 1946 and 1951.

February 6, 2016 | Proprietary and Confidential | >20 <

 Capgemini  
CONSULTING TECHNOLOGY OUTSOURCING

`awk -F":" 'length > 1024' emp.lst`

Retrieves all lines > 1024.

`awk -F":" 'length($2) > 11' emp.lst`

Retrieves all lines whose characters in 2nd col < 11.

`awk -F":" 'substr($5,7,2) > 45 && substr($5,7,2) < 52' emp.lst`

Retrieves those born between 1946 and 1951.

7.9: Control statements

## Control statements

```
if (<condition>)
{
    <Block of statements>
}
else
{
    <Block of statements>
}
```

```
While (Condition)
{
    <Block of statements>
}
for (<expr1>;<cond1>;<expr2>)
{
    <Block of
    statements>
}
```

7.9: Control statements

### Example

```
awk '  
{  
    if (NF != 3) {  
        print $0, "number of fields is not equal to 3"  
    if ($2 < 3.35) {  
        print $0, "rate is below minimum wage" }  
    if ($2 > 10) {  
        print $0, "rate exceeds $ 10 per hour" }  
    if ($3 < 0) {  
        print $0, "negative hours worked" }  
    if ($3 > 60) {  
        print $0, "too many hours worked" }  
    }  
' emp.data
```

February 5, 2016

Proprietary and Confidential

&gt; 23 &lt;



7.10 Pattern Matching

## Pattern matching

- Print all lines with pattern Susie:  

```
awk ' /Susie/ ' emp.data
```
- Print all lines in which 4th field matches with pattern Asia:  

```
awk' { if ($4 ~ /Asia/) { print } } ' countries.data
```

February 6, 2016 | Proprietary and Confidential | +23+

 Capgemini  
CONSULTING TECHNOLOGY OUTSOURCING

To print all lines in which 4 th field does not matches with pattern Asia:

```
awk'  
{  
if ($4 !~ /Asia/)  
{  
print  
}  
}  
'countries.data
```

## 7.10 Pattern Matching

## Example

# Print any line that matches exactly  
three digits

```
awk '  
{  
if (/^([0-9][0-9][0-9]$)/  
{  
print  
}  
}  
' numbers.data
```

# print any line that consists  
only digits

```
awk '  
{  
if (/^([0-9]+$/)  
{  
print  
}  
}  
' numbers.data
```

February 5, 2016 | Proprietary and Confidential | -24-



## Summary

- AWK is based on pattern matching and performing action.
- Commands enclosed in BEGIN section gets executed first.
- Then, it performs main action part on all records.
- Commands enclosed in END section gets executed.



February 6, 2016 | Proprietary and Confidential | +25+



Add the notes here.

## Review Questions

- In ----- section report header can be printed
- ----- section helps to print totals
- Print \$0 prints whole record
  - TRUE
  - FALSE



February 6, 2016 Proprietary and Confidential • 29 •



Add the notes here.

## UNIX

### Shell Programming

February 1, 2010 | Proprietary and Confidential | 414



## Lesson Objectives

➤ At the end of the session you will be able to understand:

- Shell variable
- Environment variables
- Shell script commands
- Command substitution
- Command line argument
- Conditional statements
- Iterative statements



8.1: Shell Variables

## Introduction

- **System Variables**
  - Set during:
    - Boot
    - Login
- **.profile:**
  - Script executed at login.
  - Alters operating environment of a user.
- **\$set**
  - Displays a list of system variables.

February 1, 2006 Proprietary and Confidential | - 3 -



### System Variables

There are several variables set by the system - some during booting and some after logging in. These are called the system variables, and they determine the environment one is working in. The user can also alter their values. The set statement can be used to display list of system variables.

```
$ set
HOME=/usr1/deshpavn
HUSHLOGIN=FALSE
HZ=100
IFS=
LOGNAME=deshpavn
MAIL=/usr/spool/mail/deshpavn
MAILCHECK=600
MF_ADMIN=adm.cat@Unix
MSG_MAIL=1
MS_PROFILE=1
OPTIND=1
PATH=/bin:/usr/bin:/usr1/deshpavn/bin:
PS1=$
PS2=>
SHELL=/bin/sh
TERM=ansi
TZ=IST-5:30
```

8.2: Environmental Variables

## Standard shell variables

➤ **Shell Variables**

- PATH : Contains the search path string.
- HOME : Specifies full path names for user login directory.
- TERM : Holds terminal specification information
- LOGNAME : Holds the user login name.
- PS1 : Stores the primary prompt string.
- PS2 : Specifies the secondary prompt string.

February 6, 2016 | Proprietary and Confidential | - 4 -

 Capgemini  
CONSULTING TECHNOLOGY OUTSOURCING

### Output of set command

Significance of some of these variables is explained below:

**PATH Variable:** Determines the list of directories (in order of precedence) that need to be scanned while you look for an executable command.

Path can be modified as:

```
$ PATH=$PATH:/usr/user1/progs
```

This causes the /usr/user1/progs path to get added to the existing PATH list.

**HOME Variable:** This controls the login or Home directory for the user.

**IFS Variable:** It contains a string of characters that can be used as separators on command line.

**PS1 and PS2 Variables:** These determine the primary and secondary prompt.

8.2: Environmental Variables

## Scripts executed automatically

- **.profile script**
  - shell script that gets executed by the shell when the user logs on
  - Used by Bourne shell
- **.cshrc ,.login**
  - Used by C Shell users
  - .login and is read when the user logs in.
  - .cshrc and is read whenever a new C shell is created
- **.logout script**
  - .logout file can also be created for commands to be executed when you log out.

February 6, 2016 | Proprietary and Confidential | + 5 +

 Capgemini  
CONSULTING TECHNOLOGY OUTSOURCING

### **.profile script**

The **.profile** script is a shell script that gets executed by the shell when the user logs on. It contains settings for the operating environment of the user, and it remains in effect throughout the login session. Using this file, it is possible to customize operating environment.

### **.cshrc ,.login and .logout script**

For the Bourne shell, the system reads the **.profile** file and executes the commands found there. C Shell users, however, have two files to read and execute. One is called **.login** and is read when the user logs in. The second is called **.cshrc** and is read whenever a new C shell is created, including the login shell. A **.logout** file can also be created for commands to be executed when you log out.

8.3: Shell script Commands

## Example

➤ Simple Shell Script: Accept Name & Display Message

**hello.sh**

```
echo "Good Morning!"  
echo "Enter your name?"  
read name  
echo "HELLO $name How are you?"
```

➤ To execute the shell script  
\$sh hello.sh

➤ To debug the shell script use -x option  
\$sh -x hello.sh

February 6, 2016 | Proprietary and Confidential | 6/6

 Capgemini  
CONSULTING TECHNOLOGY OUTSOURCING

In above program, the `read` command accepts input from the user and stores it in `name` variable.

To display the variable value, you need to precede the variable name with a `$` sign:

```
echo "HELLO $name How are you?"
```

8.4: Arithmetic Operations

## Details

```
echo "Enter first Number"
read no1
echo "Enter second Number"
read no2
res=`expr $no1 + $no2`
echo "The result is $res"
```

➤ In the above example, instead of `expr` we can use `let`.

- Syntax:
  - `let expressions` or `((expressions))`
- In above script `res=`expr $no1 + $no2`` can be replaced by  
`let res=no1+no2`

February 6, 2016 | Proprietary and Confidential | - 7 -

 Capgemini  
CONSULTING TECHNOLOGY OUTSOURCING

The above program accepts two numbers and displays their sum as a result. Instead of the `expr` command, we can use the `let` command.

**Example:**

Add one to variable i. Using `expr` statement:

- `i=`expr $i + 1``

Add one to variable i. Using `let` statement:

- `let i=i+1` If no spaces in expression
- `let "i=i+1"` enclose expression in “...” if expression includes spaces
- `(( i=i+1 ))`

`Expr` is generally used but `let` is more user-friendly. It is used in `Bash` and `Korn shell`

8.5: Command Substitution

### Details

- Command is enclosed in backquotes (`).
- Shell executes the command first.
  - Enclosed command text is replaced by the command output.
- Display output of the date command using echo:

```
$echo The date today is `date'  
The date today is Fri 27 00:12:55 EST 1990
```
- Issue echo and date commands sequentially:

```
$echo The date today is; date
```

February 6, 2016 | Proprietary and Confidential | - 8 -

 Capgemini  
CONSULTING TECHNOLOGY OUTSOURCING

**\$echo The date today is `date`**

In this command date is a command which is enclosed in backquotes and hence will get replaced by its output and then echo command will display message

**\$echo The date today is; date**

In above command echo and date commands are separated by ; hence will get executed sequentially.

8.5: Command Substitution

### Example

- Following instructions print pwd as a string:

```
➤ var=pwd
echo $var
Output: pwd
```
- Following instructions execute PWD shell command and display the present working directory:

```
var='pwd'
echo $var
Output: /usr/despavan
```

February 6, 2016 | Proprietary and Confidential | - 9 -

 Capgemini  
CONSULTING TECHNOLOGY OUTSOURCING

In the first example pwd is a string which is assigned to var variable. Hence o/p of echo \$var will be pwd

But in second example 'pwd' string is assigned to var variable  
Hence echo \$var command will be  
echo 'pwd'

Since pwd is enclosed in backquotes it will get replaced by present working directory. echo will display name of current working directory.

8.6: Command Line Arguments

## Details

- Specify arguments along with the name of the shell program on the command line called as **command line argument**.
- Arguments are assigned to special variables \$1, \$2 etc called as **positional parameters**.
- **special parameters**
  - \$0 – Gives the name of the executed command
  - \$\* - Gives the complete set of positional parameters
  - \$# - Gives the number of arguments
  - \$\$ - Gives the PID of the current shell
  - \$! – Gives the PID of the last background job
  - \$? – Gives the exit status of the last command
  - \${@} - Similar to \${\*}, but generally used with strings in looping constructs

February 6, 2016 Proprietary and Confidential | + 10 +

 Capgemini  
CONSULTING TECHNOLOGY OUTSOURCING

You can pass values to shell programs while you execute shell scripts. These values entered through command line are called as **command line arguments**.

### Parameters Related to Command Line Arguments

When you specify argument along with the name of the shell procedure, they are assigned into parameters \$1, \$2 etc. They are called as **positional parameters**. There are also some other **special parameters** you can use. Some of them are:

- \$0 – Gives the name of the executed command
- \${\*} - Gives the complete set of positional parameters
- \${#} - Gives the number of arguments
- \$\$ - Gives the PID of the current shell
- \${!} – Gives the PID of the last background job
- \${?} – Gives the exit status of the last command
- \${@} - Similar to \${\*}, but generally used with strings in looping constructs

8.6: Command Line Arguments

## Details

➤ Arguments are assigned to special variables (positional parameters).

- \$1 - First parameter , \$2 - Second parameter,....
- Example:

```
echo Program: $0
echo Number of arguments are $#
echo arguments are $*
grep "$1" $2
echo "\n End of Script"
```

– Run script:

\$ scr1.sh "Unix" books.lst

-\$1 is UNIX , \$2 –books.lst

 Capgemini  
CONSULTING TECHNOLOGY OUTSOURCING

In above example  
\$ scr1.sh "Unix" books.lst - The output only has lines with UNIX as substring from book.lst file .

Program: scr1.sh

Number of arguments are 2.

Arguments are Unix books.lst.

1001|Learning Unix |Computers |01/01/1998| 575  
1004|Unix Device Drivers |Computers |09/08/1995| 650  
1007|Unix Shell Programming |Computers |03/02/1993| 536

End of Script.

8.7: Conditional Execution

## Details

➤ **Logical Operators && and ||:**

- && operator delimits two commands. Second command is executed only if the first succeeds.
- || operator delimits two commands. Second command is executed only if the first fails.
- Example:

```
$grep `director` emp.lst && echo "pattern found"  
$grep `manager` emp.lst || echo "pattern not found"
```

February 6, 2016 Proprietary and Confidential ~ 12 ~

 Capgemini  
CONSULTING TECHNOLOGY OUTSOURCING

### Conditional Execution using && and ||

The shell provides && and || operators to control the execution of a command depending on the success or failure of previous command. In case of &&, the second command executes only if the first has succeeded. Similarly, || will ensure that the second command is executed only if the first has failed.

The following command displays “Found!” only if the XML pattern is found in the books.lst file at least once.

```
$ grep "XML" books.lst && echo "Found!"  
1003|XML Unleashed |Computers |20/02/2000| 398  
1006|XML Applications |Fiction |09/08/2000| 630  
Found!
```

The following command displays “Not Found ...”. If grep does not find the “WAP” pattern in the books.lst file.

```
$ grep "WAP" books.lst || echo "Not Found..."  
"Not Found..."
```

8.8: If Statement Format

## Details

- Syntax

(i) if <condition is true>  
then  
    <execute commands>  
else  
    <execute commands>  
fi  
(ii) if <condition is true>  
then  
    <execute commands>  
fi

- Example

```
if grep “^$1” /etc/passwd 2>/dev/null
then
    echo “pattern found”
else
    echo “pattern not found”
fi
```

February 6, 2016 | Proprietary and Confidential | +13 +

 Capgemini  
CONSULTING TECHNOLOGY OUTSOURCING

In UNIX **/dev/null** or **the null device** is a special file that discards all data written to it.

The null device is typically used to dispose the unwanted output stream of a process.

In given example, if **grep** returns any error and you wish to discard error messages, use **/dev/null** device.

8.9: If Statement Format

## if Statement

– Syntax:

```
(iii) if <condition is true>
    then
        <execute commands>
    elif <condition is true>
    then
        <execute commands>
    <...>
    else
        <execute commands>
    fi
```

– Example

```
if test $# -eq 0; then
    echo "wrong usage " > /dev/tty
elif test $# -eq 2 ; then
    grep "$1" $2 || echo "$1 not
        found in $2" > /dev/tty
else
    echo "you didn't enter 2
        arguments"
fi
```

The logo for Capgemini, featuring a blue cloud-like shape and the company name in a stylized font.

February 6, 2016 | Proprietary and Confidential | + 14 +

In the example, `test` command is used to specify condition. The shell script checks for two command line arguments. If the number of arguments is zero, then the output is:

**Wrong Usage**

If it is two, then the first argument is used as a pattern and the second one is used as the file name to search in the `grep` command.

If the pattern is found, then the output of the `grep` command is displayed. Otherwise, the output of the `echo` command is displayed.

If the number of arguments are not two, then the output is as follows: "you didn't enter 2 arguments".

8.9: test Statement

## Relational Operator for numbers

➤ **Specify condition either using test or [ condition ]**

- Example: `test $1 -eq $2` same as [ `$1 -eq $2` ]

➤ **Relational Operator for Numbers:**

- `eq:` Equal to
- `ne:` Not equal to
- `gt:` Greater than
- `ge:` Greater than or equal to
- `lt:` Less than
- `le:` Less than or equal to

## 8.9: test Statement

## Relational Operator for strings and logical operators

#### ➤ String operators used by test:

- |           |                                       |
|-----------|---------------------------------------|
| -n str    | True, if str not a null string        |
| -z str    | True, if str is a null string         |
| -S1 = S2  | True, if S1 = S2                      |
| -S1 != S2 | True, if S1 ≠ S2                      |
| -str      | True, if str is assigned and not null |

## ➤ Logical Operators

- a.AND.
  - o.OR.
  - ! Not

February 5, 2016 | Proprietary and Confidential | 10 |



9.9: test Statement

## File related operators

### ➤ File related operators used by test command

- |             |   |
|-------------|---|
| – -f <file> | True, if file exists and it is regular file   |
| – -d <file> | True, if file exist and it is directory file  |
| – -r <file> | True, if file exist and it is readable file   |
| – -w <file> | True, if file exist and it is writable file   |
| – -x <file> | True, if file exist and it is executable file |
| – -s <file> | True, if file exist and it's size > 0         |
| – -e <file> | True, if file exist                           |

8.9: test Statement

## Example

➤ Check whether user has entered a filename or not:

— Example:

```
echo "Enter File Name:\c"
read fn
if [ -z "$fn" ]
then
  echo "You have not entered file name"
fi
```

February 6, 2016

Proprietary and Confidential

+18+



In the given example `-z` checks whether `$fn` is empty or not. If users do not enter the file name, then the output is as follows:

“You have not entered file name“.

8.9: test Statement

## Example

➤ Example:

```
if test $x -eq $y  
≡ if [ $x -eq $y ]
```

➤ Example:

```
If [ ! -f fname ]  
then  
    echo "file does not exists"  
fi
```

February 6, 2016 | Proprietary and Confidential | + 19 +

 Capgemini  
CONSULTING TECHNOLOGY OUTSOURCING

```
if test $x -eq $y  
≡ if [ $x -eq $y ]
```

In above command both the conditions are the same. You can use the “[“ bracket to check the condition in place of the test command.

`test $x -eq $y` returns true if the values of variables x and y are equal. You can write the same condition as `[ $x -eq $y ]`. Here, instead of test command we use “[“ (square bracket).

```
If [ ! -f fname ]
```

You can also write this condition as:  
`test !-f fname`

8.9: test Statement

### Example

```
echo "Enter the source file name : \c"
read source
#check for the existence of the source file
if test -s "$source" #file exists & size is > 0
then
    if test ! -r "$source"
    then
        echo "Source file is not readable"
        exit
    fi
else
    echo "Source file not present"
    exit
fi
```

February 6, 2016

Proprietary and Confidential

&gt;20&gt;



The above example checks whether a given source file exists and displays appropriate messages.

8.10: Case Statement

## Case command

– Syntax:

```
case <expression> in
  <pattern 1> ) <execute
    commands>;;
  <pattern 2> ) <execute
    commands>;;
  <...>
  <...>
esac
```

– Example:

```
echo “\n Enter Option : |c”
read choice
case $choice in
  1) ls -l;;
  2) ps -f;;
  3) date;;
  4) who;;
  5) exit;;
esac
```

February 6, 2016 Proprietary and Confidential | 21 |

 Capgemini  
CONSULTING TECHNOLOGY OUTSOURCING

In a **case** statement you can also use commands enclosed in *backquotes*. The given example executes command **'date | cut -d " -f1'** which returns only the day part. The output is used to execute the appropriate case.

**Example:**

```
case `date | cut -d " -f1` in
  Mon ) <commands>;;
  Tue ) <commands>;;
  :
esac
```

**Example:**

#display the options to the user

```
echo “1. Date and time
      2. Directory listing”
echo “3. Users information
      4. Current directory”
echo “Enter choice (1,2,3,4):|c”
read choice
case $choice in
  1) date;;
  2) ls -l;;
  3) who;;
  4) pwd;;
*) echo wrong choice;;
esac
#end of script
```

8.10: Case Statement

### Example

```
echo "do you wish to continue?"  
read ans  
Case "$ans" in  
[yY][eE][sS];;  
[nN][oO]) exit ;;  
*) "invalid option";;  
esac
```

February 6, 2016

Proprietary and Confidential

&gt;23&lt;



In the above example, the first case matches with “yes” or “YES”. Similarly, the second case matches with “no” or “NO”.

8.11: While loop Statement

## Syntax and Example

— Syntax:

```
while <condition is true>
do
  <execute statements>
done
```

e.g.

```
while [ $x -gt 3 ]
do
  ps -a
  sleep 5
done
```

```
while true
do
  ps -a
  sleep 5
done
```

February 6, 2016 | Proprietary and Confidential | +23+

 Capgemini  
CONSULTING TECHNOLOGY OUTSOURCING

**Example:** Script to edit, compile and execute a program.

```
while true
Do
  cc $1
  case $? In
    0) echo "Compilation Successful"
       echo "Executing a.out"
       a.out ; exit ;;
    *) echo "Compilation Error"
       echo "Press <Enter> to edit"
       read pause
       vi $1 ;;
  Esac
done
```

8.11: Examples

## Example : While

```
#using while loop
num=1
while [ $num -le 10 ]
do
    echo $num
    num=`expr $num + 1`
done
#end of script
```

February 1, 2016 | Proprietary and Confidential | +24+

 Capgemini  
CONSULTING TECHNOLOGY OUTSOURCING

In the above example, the loop executes till the condition is true. This is till the value of the variable num is < 10.

8.12: Break &amp; Continue Statement

## break and continue statement

➤ **Continue:**

- Suspends statement execution following it.
- Switches control to the top of loop for the next iteration.

➤ **Break:**

- Causes control to break out of the loop.

February 5, 2016

Proprietary and Confidential

+25+



8.12: Break & Continue Statement

### Example

```
while echo "designation : |c"
    do
        read desig
        case "$desig" in
            [0-9]) if grep "^$desig" emp.lst >/dev/null
                    then
```

February 6, 2016 Proprietary and Confidential | 25 |

 Capgemini  
CONSULTING TECHNOLOGY OUTSOURCING

In above example, the `while` loop is an unending loop as `echo "designation : |c"` statement (which is put as a condition in the `while` loop) always returns an exit status of success (condition becomes true).

Hence, it is more efficient if you write the following as a single statement:  
`while true`  
`echo "designation : |c"`

In the above program if you enter a designation as a two digit number, it matches with case [0-9][0-9]. If the designation found in the file break statement is executed, control comes out of the loop and the program halts.  
Otherwise, the default case is executed. Continue statement transfers the control at the beginning of the loop.

8.13: until loop

## Syntax

- Complement of **while** statement.
- Loop body executes repeatedly as long as the condition remains **false**.
  - Example:

```
until false
do
    ps -a
    sleep 5
done
```

February 6, 2016 | Proprietary and Confidential | +27+

 Capgemini  
CONSULTING TECHNOLOGY OUTSOURCING

The syntax is as follows:

```
until condition
do
    commands
Done
```

This loop is a complement of the *while* loop. In the *while* loop statements are repeated till the condition is *true*. But in an *until* loop, statements inside loop are repeated till the condition is *false*. As soon as the condition becomes *true*, the iteration stops.

In the above example given until loop is infinite loop.

8.14: For Statement

## for statement

— Syntax:

```
for variable in list
do
    <execute
    commands>
done
```

— Eg:

```
for x in 1 2 3
do
    echo "The value of x is $x"
done
```

```
for var in $PATH $HOME $MAIL
do
    echo "$var"
done
```

```
for file in *.c
do
    cc $file
done
```

February 1, 2016 | Proprietary and Confidential | ~18 ~

 Capgemini  
CONSULTING TECHNOLOGY OUTSOURCING

**Example 1:**

In this example, for loop executes three times because three numbers are there in the list . In every iteration x is assigned 1, 2 and 3 respectively.

**Example 2:**

In this example also, for loop executes 3 times. In each iteration, var takes values from system variables in the list \$PATH, \$HOME and \$MAIL respectively.

**Example 3:**

In this example, the for loop iterates equal to the number of files with extension c in the current working directory. This is because \*.c is replaced with a list of all files with extension c in the current working directory.

Some more examples are:

```
for i in 1 2 3 4 5 6 7 8 9 0
do
    echo $i
done
```

### Example : for

```
for file in chap20 chap21 chap22 chap23;  
do  
cp $file ${file}.bak  
echo $file copied to ${file}.bak  
done
```

```
for file in 'catclist'.....
```

```
for file in *.htm *.html;  
do  
# do something  
done
```

```
for pattern in "$@"; do  
grep "$pattern" emp.lst || echo "$pattern not found"  
done
```

8.14 For Statement

## Details

— Syntax:

```
for (( expr1; expr2; expr3
      ))
  do
..... repeat all
statements between
do and done until
expr2 is TRUE
done
```

e.g.

```
for(( i = 0 ; i <= 5; i++ ))
do
echo "Welcome $i times"
done
```

February 6, 2016 Proprietary and Confidential | 30 |

 Capgemini  
CONSULTING TECHNOLOGY OUTSOURCING

In above example, syntax before the first iteration, `expr1` is evaluated. This is usually used to initialize variables for the loop. All statements between `do` and `done` are executed repeatedly until the value of `expr2` is true.

After each iteration of the loop, `expr3` is evaluated. This is usually used to increment a loop counter.

The output of the given example is:

```
Welcome 0 times
Welcome 1 times
Welcome 2 times
Welcome 3 times
Welcome 4 times
Welcome 5 times
```

8.15: Examples

## Example : Until

```
#script to create a employee file
ans="y"
until [ $ans = "N" -o $ans = "n" ]
do
    echo "Enter the name :\c"
    read name
    echo "Enter the grade :\c"
    read grade
    echo "Enter the basic :\c"
    read basic
    echo $name: $grade : $basic >>emp
    echo "Want to continue (Y/N) :\c"
    read ans
done
#end of script
```

February 6, 2016 Proprietary and Confidential - 31 -



In above example the loop executes till the condition is false. This is as soon as the user enters "N" or "n" for `ans`, the condition is true and the loop stops iteration.

Some more examples of shell script are:

**Script to accept five numbers and display their sum:**

```
echo the parameters passed are : $1, $2, $3, $4, $5
echo the name of script is : $0
echo the number of parameters passed are : $#
#calculate the sum
sum=`expr $1 + $2 + $3 + $4 + $5`
echo the sum is $sum
#end of script
```

Invoke this script as follows:

`$sh disp_sum 10 12 13 14 15`

The above command is to be followed by 5 different number as shown.

8.16: Shell functions

## Functions in Shell Script

- Use shell functions to modularize the script.
- These are also called as script module
- Normally defined at the beginning of the script.
- Syntax (Function Definition):

```
functionname(){  
    commands  
}
```
- Example: Function to create a directory and change directories:
- Use mkcd mydir to call the function. mydir is used as \$1 in the function.

```
mkcd()  
{  
    mkdir $1    --$1 is the argument we pass while calling function  
    cd $1  
}
```

February 6, 2016 | Proprietary and Confidential | ~32~

 Capgemini  
CONSULTING TECHNOLOGY OUTSOURCING

You can also call the shell function *script module* as it makes a whole script section available under a single name. Normally, shell functions are defined at the beginning of the script. Or several functions can be stored in a file and read whenever they are needed. Files are stored in the *bin* directory. Function name can be any combination from the regular character string.

8.16 : Shell functions

## Using return statement

➤ **Used to come out of a function from within.**

- If called without an argument, function return value is the same as exit status of the last command executed within the function
- If called with an argument it returns the argument specified.
- Example:

```
functret()  
{  
    command1  
    if .....  
    then  
        return 1  
    else  
        return 0  
    fi  
    Command2  
}
```

February 6, 2016

Proprietary and Confidential

- 33 -



8.16 : Shell functions

## Using return statement

```
Myfunction(){  
echo "$*"  
echo "The number should be between 1 and 20"  
read num  
if [ $num -le 1 ]-a [ $num -ge 20 ]  
    return 1;  
else  
    return 0;  
fi  
echo "You will never reach to this line"}  
  
echo "Calling the function Myfunction"  
if Myfunction "Enter the number"  
then  
    echo "The number is within range"  
else  
    echo the number is out of range"  
}
```

February 6, 2016 Proprietary and Confidential • 34 •

 Capgemini  
CONSULTING TECHNOLOGY OUTSOURCING

In the above example, *Myfunction* is called in the *if* statement with message “enter the number”. This message is passed as three arguments.

In *Myfunction*, the first line is *echo \$\**.

Hence, it display message “Enter the number”.

*Read num* accepts the number.

If the number is between 1 and 20, the function returns 1, otherwise it returns 0.

If *Myfunction* returns 1, then the output is:

**The number is within range.**

Otherwise the output should be as follows:

**The number is out of range.**

8.16 : Arrays

## Using arrays

- Contains a collection of values accessible by individuals or groups
  - Subscript of array element indicates their position in the array.
    - `arrayname[subscript]`
- First element is stored at subscript 0.
  - Assign a value in `flowers` array at the first position.
    - `Flowers[0]=Rose`
- Assign values in an array with a single command:
  - `$ set -A Flowers Rose Lotus`
- Access individual array elements
  - `${arrayname[subscript]}`

8.16 : Arrays

## Using arrays

➤ To print values from array we can use while loop

```
flowers[0]=Rose
flowers[1]=Lotus
flowers[2]=Mogra
i=0
while [ $i -lt 3 ]
do
echo ${flowers[$i]}
i=`expr $i+1`
done
```

➤ Access all elements:

```
 ${array_name[*]}
 ${array_name[@]}
```

February 6, 2016 Proprietary and Confidential | 38 |

 Capgemini  
CONSULTING TECHNOLOGY OUTSOURCING

You can display all elements from the array using \* or @ symbol:  
Num[0]—"Zero"  
Num[1]—"One"  
Num[2]—"Two"  
Num[3]—"Three"  
echo "First Method: \${NAME[\*]}"  
echo "Second Method: \${NAME[@]}"

## Summary

- **.profile:**
  - Script executed during login time.
- **Command enclosed in backquotes (`):**
  - Shell executes the command first
  - Enclosed command text is replaced by the command the output.
- **Test:**
  - Command used to check the condition in an if statement.
- **Different loop statements in Unix are:**
  - For
  - While
  - Until



## Review Questions

### ➤ Complete The Following

- ----- command can be replaced by test command.
- ----- condition checks whether two strings are equal or not.
- ----- loop terminates as soon as condition becomes true.

### ➤ TRUE OR FALSE

- PS1 stores primary cursor string:



## UNIX

Source Code Control system

February 1, 2010 | Proprietary and Confidential | 414



## Lesson Objectives

➤ SVN



February 6, 2016

Proprietary and Confidential

- 2 -



9.1: SVN - Subversion

## What is SVN?

➤ **Subversion (SVN):**

- Software versioning and revision control system.
- Tool that help in to control various program versions.
- Only incremental changes to the program are stored. Multiple copies are not stored.
- Particularly useful when programs are enhanced but the original version is still needed.

February 6, 2016 Proprietary and Confidential - 3 -

 Capgemini  
CONSULTING TECHNOLOGY OUTSOURCING

A version control system (or revision control system) is a system that tracks incremental versions (or revisions) of files and, in some cases, directories over time. Of course, merely tracking the various versions of a user's (or group of users') files and directories isn't very interesting in itself. What makes a version control system useful is the fact that it allows user to explore the changes which resulted in each of those versions and facilitates the arbitrary recall of the same.

9.1: SVN - Subversion

## Repository

- Repository is the central store of that system's data.
- The repository usually stores information in the form of a file system tree ( a hierarchy of files and directories)
- Any number of clients connect to the repository, and then read or write to these files.

A typical client/server

February 6, 2016 Proprietary and Confidential - 4 -

 Capgemini  
CONSULTING TECHNOLOGY OUTSOURCING

At the core of the version control system is a repository, which is the central store of that system's data. The repository usually stores information in the form of a files system tree -- a hierarchy of files and directories. Any number of clients connect to the repository, and then read or write to these files. By writing data, a client makes the information available to others; by reading data, the client receives information from others.

What makes the repository special is that as the files in the repository are changed, the repository remembers each version of those files.

When a client reads data from the repository, it normally sees only the latest version of the files system tree. But what makes a version control client interesting is that it also has the ability to request previous states of the files system from the repository. A version control client can ask historical questions such as What did this directory contain last Wednesday? and Who was the last person to change this file, and what changes did he make? These are the sorts of questions that are at the heart of any version control system.

**Note :** Repository creation is a one time activity which happens on server and usually done by system administrator.

9.1: SVN - Subversion

## Initial Project Setup

- If the project is not yet started , then create the basic directories required for project directly on repository.
- If the project is already in place and the directory structure has been already defined then import the directory structure (along with the all latest files ) to the SVN repository.
- After importing now delete local copy as files are now successfully stored in SVN repository.
- Command for creating , Removing Directory and Checking log

```
svn mkdir file:///repository_path/directory_name -m "Message"  
svn rm file:///repository_path/directory_name -m "Message"  
svn log file:///repository_path
```

February 1, 2016 Proprietary and Confidential | + 5 +

 Capgemini  
CONSULTING TECHNOLOGY OUTSOURCING

**Note :** Initial Project Setup creation is a one time activity which can be done by administrator or user .

Following is the command to import the entire project from client to server

```
svn import user_directory_path file:///repository_path -m 'Message'
```

9.1: SVN - Checkout

## The Working Copy (Check-Out)

- An ordinary directory tree on local system, containing a collection of files is called **working copy**.
- Owner's private work area ( edit , compile, change any files/code)
  - Complete Checkout
    - svn checkout file:///repository\_path
  - Partial Checkout
    - svn checkout file:///repository\_path/Directory\_Path
- Changes in working copy needs explicit commit to update the changed version in repository

February 1, 2016 | Proprietary and Confidential | 6/6

 Capgemini  
CONSULTING TECHNOLOGY OUTSOURCING

9.1: SVN – Lock, Modify, Unlock

## The Problem of File-Sharing

➤ The challenge is :

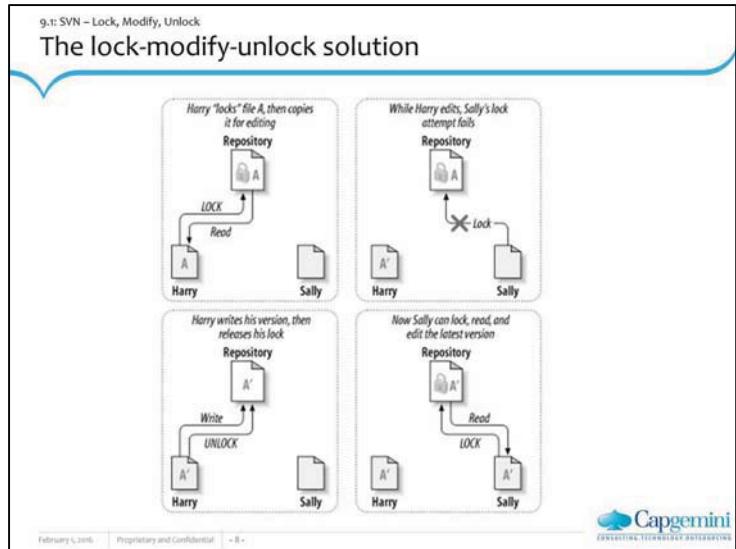
- How we can make sure that only one person at a time should modify the file

February 6, 2016 Proprietary and Confidential | + 7 +

 Capgemini  
CONSULTING TECHNOLOGY OUTSOURCING

All version control systems have to solve the same fundamental problem: how will the system allow users to share information, but prevent them from accidentally stepping on each other's feet? It's all too easy for users to accidentally overwrite each other's changes in the repository.

Suppose we have two co-workers, Harry and Sally. They each decide to edit the same repository file at the same time. If Harry saves his changes to the repository first, then it's possible that (a few moments later) Sally could accidentally overwrite them with her own new version of the file. While Harry's version of the file won't be lost forever (because the system remembers every change), any changes Harry made won't be present in Sally's newer version of the file, because she never saw Harry's changes to begin with. Harry's work is still effectively lost or at least missing from the latest version of the file and probably by accident. This is definitely a situation we want to avoid!



#### The Lock-Modify-Unlock Solution

Many version control systems use a lock-modify-unlock model to address the problem of many authors clobbering each other's work. In this model, the repository allows only one person to change a file at a time. This exclusivity policy is managed using locks. Harry must lock a file before he can begin making changes to it. If Harry has locked a file, then Sally cannot also lock it, and therefore cannot make any changes to that file. All she can do is read the file, and wait for Harry to finish his changes and release his lock. After Harry unlocks the file, Sally can take her turn by locking and editing the file.

The problem with the lock-modify-unlock model is that it's a bit restrictive, and often becomes a roadblock for users:

- Locking may cause administrative problems. Sometimes Harry will lock a file and then forget about it. Meanwhile, because Sally is still waiting to edit the file, her hands are tied.
- Locking may cause unnecessary serialization. What if Harry is editing the beginning of a text file, and Sally simply wants to edit the end of the same file? These changes don't overlap at all. They could easily edit the file simultaneously.

The solution of this problem is "Copy-Modify-Merge"

9.t: SVN – Few More Commands

## SVN-More Command

➤ **Addition of file to repository**

```
svn add File_Path/File_Name File_Path/File_Name
```

➤ **Committing the changes**

```
svn commit -m 'Message'
```

➤ **Locking the file**

```
svn lock File_Path/File_Name File -m "Message"
```

➤ **Un-Locking the file**

```
svn unlock File_Path/File_Name
```

February 5, 2016

Proprietary and Confidential

+ 9 \*



## Summary

➤ SVN



February 6, 2016

Proprietary and Confidential

+ 10 +



### Review Questions

- In \_\_\_\_\_ multiple copies are not stored only changes are stored.
- List any 3 commands used in SVN?



## Reference Books

- **The Unix Programming Environment**
  - Kerningham & Pike, Prentice Hall
- **Unix System V.4 Concepts and Applications**
  - Sumitabha Das, Tata McGraw-Hill
- **Advanced Unix Programmer's Guide**
  - Stephen Prata, BPB
- **Introducing Unix System V**
  - Vijay Mukhi, Tata McGrawHill



# UNIX

(Version: 1.5)

Lab Book

---

Copyright © 2011 IGATE Corporation (a part of Capgemini Group). All rights reserved.

No part of this publication shall be reproduced in any way, including but not limited to photocopy, photographic, magnetic, or other record, without the prior written permission of IGATE Corporation (a part of Capgemini Group).

IGATE Corporation (a part of Capgemini Group) considers information included in this document to be confidential and proprietary.

---

## Document Revision History

---

Date	Revision No.	Author	Summary of Changes
	1	Veena Deshpande	New course creation
30-Sept-2009	2	Kishori Khadilkar	Revamped as per new template
20-June-2011	3	Rathnajothi Perumalsamy	Revamped as per Integrated syllabus
30-Sep -2013	4	Amit Sali	Lab for SVN(subversion) is added.
9-Mar-2015	5	Vishal Pachpute	Lab for AWK is added.

---

## Table of Contents

---

<i>Document Revision History</i> .....	2
<i>Table of Contents</i> .....	3
<i>Getting Started</i> .....	4
<i>Overview</i> .....	4
<i>Setup Checklist</i> .....	4
<i>Instructions</i> .....	4
<i>Learning More (Bibliography if applicable)</i> .....	4
<i>Lab 1. Connecting to the Unix Server</i> .....	5
1.1: <i>Connecting to the Unix Server</i> .....	5
1.2: <i>Logging out of the system</i> .....	5
<i>Lab 2. Unix Basic Command</i> .....	6
2.1 <i>Executing basic commands</i> : .....	6
<i>Lab 3. UNIX File System &amp; Permissions</i> .....	9
3.1: <i>Viewing the File System and Granting/Removing Permissions</i> .....	9
( <i>Note: Create required files if doesn't exists</i> .) .....	9
<i>Lab 4. Simple and Advance Filetrs</i> .....	10
4.1: <i>Using Pipes and Filters</i> : .....	10
<i>Lab 5. Vi Editor</i> .....	13
5.1: <i>Working wth Vi Editor</i> .....	13
<i>Lab 6. SED Commands</i> .....	14
6.1: <i>Using SED Commands</i> .....	14
<i>Lab 7. Process Related Commands</i> .....	15
7.1: <i>Using Process-Related Commands</i> .....	15
<i>Lab 8. AWK</i> .....	16
8.1: <i>Writing awk-Scripts</i> .....	16
<i>EmpName:EmpId:Subject:ObtMarks:TotMarks:Result</i> .....	16
<i>Lab 9. Shell Script</i> .....	19
9.1: <i>Writing Shell-Scripts</i> .....	19
<i>Lab 10. Source Code Control System</i> .....	22
11.1: <i>Repository Creation</i> .....	22
11.2: <i>Initial Project Setup</i> .....	22
11.3: <i>Creating the working copy of project (Check out)</i> .....	23
11.4: <i>Modifying, updating, creating new file in the project</i> .....	24
11.5: <i>The lock-modify-unlock Cycle</i> .....	25
11.6: <TODO> <i>Working on SVN</i> .....	25

---

## Getting Started

---

### Overview

This lab book is a guided tour for learning Unix. It comprises 'To Do' assignments. Follow the steps provided and work out the 'To Do' assignments.

### Setup Checklist

Here is what is expected on your machine in order for the lab to work

### Minimum System Requirements

- Intel Pentium 90 or higher (P166 recommended)
- Microsoft Windows 95, 98, or NT 4.0, 2k, XP.
- Memory: 32MB of RAM (64MB or more recommended)

### Please ensure that the following is done:

- A text editor like Notepad is installed.
- Participants should be able to connect to UNIX server through telnet (IP address : 192.168.224.34)

### Instructions

- For all coding standards refer Appendix A. All lab assignments should refer coding standards.
- Create a directory by your name in drive <drive>. In this directory, create a subdirectory html\_assgn. For each lab exercise create a directory as lab <lab number>

### Learning More (Bibliography if applicable)

- UNIX Concepts and Application by Sumitabha Das
- The Unix Programming Environment", by Kernighan and Pike.
- UNIX Primer Plus, Third Edition. Don Martin, Stephen Prata, Mitchell Waite, Michael Wessler, and Dan Wilson
- Advanced Unix : a programmer's guide / Stephen Prata

## Lab 1. Connecting to the Unix Server

---

Goals	<ul style="list-style-type: none"><li>• Learn to connect to the Unix server</li><li>• Learn to log out of the Unix server</li></ul>
Time	5 min

### 1.1: Connecting to the Unix Server

**Step 1:** Enter your login name and password to login to the UNIX system.

### 1.2: Logging out of the system

**Step 1:** Type the exit command at \$ prompt or else, press ctrl and d together to log out.

## Lab 2. Unix Basic Command

---

Goals	<ul style="list-style-type: none"> <li>• Learn to use basic Unix commands</li> </ul>
Time	100 min
Lab Setup	Telnet with Unix Server

### 2:1 Executing basic commands:

1. To display the current working directory, the command is:  
**pwd**  
The output is as follows.  
/home/trg1
2. Display the path to and name of your HOME directory.
3. Display the login name using which you have logged into the system
4. Display the hidden files of your current directory.
5. List the names of all the files in your home directory.
6. Using the long listing format to display the files in your directory.
7. List the files beginning with chap followed by any number or any lower case alphabet. (Example, it should display all files whose names are like chap1, chap2, chap3 ....., chapa,ahapb,chapc,.....)
8. Give appropriate command to create a directory called C\_prog under your home directory. (Note: Check the directory using ls)
9. Create the following directories under your home directory. (Note: Check using ls)  
newdir  
newdirectory
10. List the names of all the files, including the contents of the sub directories under your home directory.
11. Remove the directory called newdirectory from your working directory.
12. Create a directory called temp under your home directory.
13. Remove the directory called newdir under your home directory and verify the above with the help of the directory listing command.
14. Create another directory directorynew under the temp directory.
15. Change the directory to your home directory.
16. From your home directory, change the directory to directorynew using relative and absolute path.
17. Remove the directory called c\_prog, which is in your home directory.
18. Change to the directory /etc and display the files present in it.

19. List the names of all the files that begin with a dot in the /usr/bin directory.
20. Create a file first.unix with the following contents.

Hi! Good Morning everybody.  
Welcome to the First exercise on UNIX.  
Hope you enjoy doing the assignments.
21. Copy the file first.unix in your home directory to first.unics.

(Note: checked using ls, first.unix file also should exist along with first.unics)
22. List the contents of first.unix and first.unics with a single command.
23. Create a new directory under the temp directory.
24. From your home directory, copy all the files to the directory created under the temp sub directory.
25. Move the file first.unix to the directory temp as second.unix
26. Remove the file called first.unics from the home directory.
27. Change your directory to temp and issue the command rm \*. What do you observe?
28. Move all files whose names end with a, c and o to the HOME directory.
29. Copy all files that end with a 'UNIX' to the temp directory.
30. Issuing a single command, remove all the files from the directory temp and the directory itself.
31. Try commands cp and mv with invalid number of arguments and note the results.
32. Use the cat command to create a file friends, with the following data:

Madhu	6966456	09/07/68
Jamil	2345215	08/09/67
Ajay	5546785	01/04/66
Mano	7820022	09/07/68
David	8281292	09/09/60
Simmi	7864563	12/12/70
Navin	2224311	30/05/68

The fields should be separated by a tab.
33. Display contents of the file friends.
34. Copy contents of friends to newfriend without using the cp command.
35. Display contents of the file friends and newfriends in a single command.
36. Find all users currently working on the system and store the output in a file named as users.
37. Append contents of friends file to the file, users.
38. Display current system date and time and record your observations. How is the time displayed?

39. Display calendar for the month and year of your birth.
40. Try following commands and record your observations.

```
date "+ %"
date "+%m"
date "+%D"
date "+%/%Training Activity"
date "+%Training Activity"
date "+%r"
```

## Lab 3. UNIX File System & Permissions

Goals	<ul style="list-style-type: none"><li>• Learn to grant and to remove permissions and to view the file system</li></ul>
Time	15 min
Lab Setup	Telnet with Unix Server

### 3.1: Viewing the File System and Granting/Removing Permissions

(Note: Create required files if doesn't exists.)

1. Give the execute permission for the user for a file chap1
2. Give the execute permission for user, group and others for a file add.c
3. Remove the execute permission from user, give read permission to group and others for a file aa.c
4. Give execute permission for users for a.c, kk.c, nato and myfile using single command
5. Change the directory to root directory. Check the system directories, like bin, etc, usr etc

---

## Lab 4. Simple and Advance Filetrs

---

Goals	<ul style="list-style-type: none"> <li>• Learn to use Pipes &amp; Filters in UNIX</li> </ul>
Time	100 min
Lab Setup	Telnet with Unix Server

### 4.1: Using Pipes and Filters:

- 1: Redirect the content of the help document ls, into a file called as lsdoc.
- 2: Display the content of the lsdoc page wise.
- 3: Display only the first 4 lines of the lsdoc file.
- 4: Display only the last 7 lines of the file lsdoc.
- 5: Remove the file lsdoc.
- 6: There will be B'day celebration from the friends file, find how many B'day parties will be held. If two of the friends have the B'date on the same day, then we will be having one party on that day.
- 7: Display the lines starting with Ma, in the file friends.
- 8: Display the lines starting with Ma, ending with i or ending with id, in the file friends.
- 9: Print all the files and the directory files from the current directory across all the sub directories, along with its path
- 10: Print only the Directory files.
- 11: Display the files starting with chap, along with its path.
- 12: Sort the file friends in ascending order of names.
- 13: Display the contents of the file friends in uppercase letters.
- 14: Store the contents of your home directory in a file called dir.
- 15: From the above file dir, display the file permissions and the name of the file only.
- 16: From the same dir file, store only the file names in a file called files.

- 17: From the same dir file, store only the permissions of files in a file called perms.
- 18: From the same dir file, store only the file sizes in a file called sizes.
- 19: Display the file names, sizes and permissions from your directory in that order.
- 20: Display the number of users working on the system.
- 21: Find out the smallest file in your directory.
- 22: Display the total number of lines present in the file friends.
- 23: Create the following fixed record format files (with "|" delimiter between fields) with the structure given below, and populate them with relevant data use these files to solve following questions  
emp.lst: Empid(4),Name(18),Designation(9),Dept(10),Date of Birth(8),Salary(5)  
dept.lst : Dept.Code(2),Name(10),Head of Dept's id(4)  
desig.lst: Designation Abbr.(2), Name (9)
  1. Find the record lengths of each file.
  2. Display only the date of birth and salary of the last employee record.
  3. Extract only employee names and designations. (Use column specifications). Save output as cfile1.
  4. Extract Emp.id, dept, dob and salary. (Use field specifications). Save output as cfile2.
  5. Fix the files cfile1 and cfile2 laterally, along with the delimiter.
  6. Sort the emp.lst file in reverse order of Emp. Names.
  7. Sort the emp.lst file on the salary field, and store the result in file srtf.
  8. Sort the emp.lst file on designation followed by name.
  9. Sort the emp.lst file on the year of birth.
  10. Find out the various designations in the employee file. Eliminate duplicate listing of designations.
  11. Find the non-repeated designation in the employee file.
  12. Find the number of employees with various designations in the employee file.
  13. Create a listing of the years in which employees were born in, along with number of employees born in that year.
  14. Use nl command to create a code table for designations to include designation code (Start with dept. code 100, and subsequently 105, 110 ...).

- 24: PCS has its offices at Pune, TTC and Mumbai. The employees' data is stored separately for each office. Create appropriate files (with same record structure as in previous assignment) and populate with relevant data.
1. List details about an employee 'Manu Sharma' in the Mumbai office.
  2. List only the Emp.Id. And Dept. of Manu Sharma.
  3. List details of all managers in all offices. (O/P should not contain file names.).
  4. Find the number of S.E. in each office.
  5. List only the Line Numbers and Employee names of employees in 'H/W' in Pune file.
  6. Obtain a listing of all employees other than those in 'HR' in the Mumbai file and save contents in a file 'nonhr'.
  7. Find the name and designation of the youngest person who is not a manager.
  8. Display only the filename(s) in which details of employee by the name 'Seema Sharma' can be found.
  9. Locate the lines containing saxena and saksena in the Mumbai office.
  10. Find the number of managers who earn between 50000 and 99999 in the Pune office.
  11. List names of employees whose id is in the range 2000 – 2999: in Pune Office; in all offices.
  12. Locate people having same month of birth as current month in Pune office.
  13. List details of all employees other than those of HR and Admin in file F1.
  14. Locate for all Dwivedi, Trivedi, Chaturvedi in Pune file.
  15. Obtain a list of people in HR, Admin and Recr. depts. sorted in reverse order of the dept.

**Stretched assignments:**

- 25: Write a command sequence that prints out date information in this order: time, day of week, day number, month, year:  
13:44:42 IST Sun 16 Sept 1994
- 26: Write a command sequence that prints the names of the files in the current directory in the descending order of number of links
- 27: Write a command sequence that prints only names of files in current working directory in alphabetical order
- 28: Write a command sequence to print names and sizes of all the files in current working directory in order of size
- 29: Determine the latest file updated by the user
-

---

## Lab 5. Vi Editor

---

Goals	Work with Vi Editor in Unix
Time	30 min
Lab Setup	Telnet with Unix Server

### 5.1: Working wth Vi Editor

1. Create a file using Vi. Enter the following text:  
*A network is a group of computers that can communicate with each other, share resources, and access remote hosts or other networks. Netware is a computer network operating system designed to connect, manage, and maintain a network and its services. Some of the network services are Netware Directory Services (NDS), file system, printing and security.*
  - a. Change the word "Netware" in the second line to "Novell Netware".
  - b. Insert the text "(such as hard disks and printers)" after "share resources" in the first line.
  - c. Append the following text to the file:  
"Managing NDS is a fundamental administrator role because NDS provides a single point for accessing and managing most network resources."
2. Create the data files, used in the previous lab sessions using vi editor.

---

## Lab 6. SED Commands

---

Goals	Learn to use SED Commands in Unix
Time	15 min
Lab Setup	Telnet with Unix Server

### 6.1: Using SED Commands

1. Create a file "Employee.dat" with text as follows.

```
James    76382  PACE  Chennai
John     34228  GRIT  Hyderabad
Peter    22321  GE    Bangalore
Albert   32342  GRIT  Pune
Mathew   23222  PACE  Mumbai
Richard  23232  ACS   Pune
```

- a) Write a sed command to print only the lines starting at line 2 and ending with the letters "Pune"
  - b) Write a sed command that will display the top 5 lines from the file
  - c) Write a sed command that will substitute the word "Chennai" for "Pune" used in all instance of the word
  - d) Write a sed command that will replace occurrence of the character e with the string UNIX in all lines. (Use -e option)
  - e) Write a sed command to delete blank lines
  - f) Write a sed command to delete lines from 3 to 5
- 2: Create a new file "PACE.dat which has only the lines that contain the word "PACE" from Employee.dat

## Lab 7. Process Related Commands

---

Goals	Learn to use process-related commands in Unix
Time	15 min
Lab Setup	Telnet with Unix Server

### 7.1: Using Process-Related Commands

1. Find out the PID of the processes that are activated by you
2. Find out the information about all the processes that are currently active
3. Start a different process in the background. Find out the status of the background process using the PID of the same.

---

## Lab 8. AWK

---

Goals	Learn to write awk scripts
Time	90 min
Lab Setup	Telnet with Unix Server

### 8.1: Writing awk-Scripts

1. Consider the results are stored in following format:

EmpID	Name	Subject	Marks(/50)
E001	Nilesh	Unix	30
E002	Nilesh	DSA	20

Like these you have 10 records ( 5 of DSA and 5 of Unix)

Calculate the avg score secured in Unix and DSA and the first 2 topers in Unix and DSA each.

2. Write a script to get the report of the users logged on to the System in the following formats. (Records should be sorted on logging time.)

Header must include company name and Date

Records in the format

Username Logged-in-time Terminal

Tailor should include total number of the users logged in.

3. Consider a text file containing the records (colon separated fields) in the format:  
**EmpName:EmpId:Subject:ObtMarks:TotMarks:Result**

Write a script to get the result of “UNIX” Subject in the format (Considering the data file has TotMarks=50 for UNIX)

EmpName:ObtMark:MarksOutof35

The header of the report must contain total marks and the tailor must specify the percentage result for that subject.

Also generate another summary result containing total number of participants appeared, total number of participants passed, and Name of the participants ranked 1st IIInd, IIInd with their total score.

4 Consider the Arizona roaster as an input data file for this exercise. Using any combination of the text-processing utilities listed below, write a one-line shell command that performs each of the following tasks

- (a) reports the number of players on the roster
- (b) displays the roster in order by jersey number
- (c) displays the roster in alphabetical order by surname
- (d) displays the heaviest five players
- (e) displays all players who attended Wisconsin

The Roaster is given below:-

JersyNo	Name,surname	linebackers	Weight	Date	Experience	Country
20	Anderson, Damien	RB 5'10"	212	07/17/1979	3	Northwestern
30	Ayanbadejo, Oba	FB 6'02"	235	03/05/1975	5	San Diego
92	Berry,Bert	DE6'03"	250	08/15/1975	7	NotreDame
81	Boldin,Anquan	WR6'01"	215	10/03/1980	2	FloridaState
91	Bryant,Wendell	DT6'04"	303	09/12/1980	3	Wisconsin
35	Carter,Dyshod	DB5'10"	197	06/18/1978	2	Kansas State
52	Fisher,Levar	OLB6'01"	235	07/02/1979	3	NorthCarolina
11	Fitzgerald,Larry	WR6'04"	229	08/31/1983	10	Pittsburgh
63	Garcia,Frank	G6'02"	302	01/28/1972	10	Washington
64	Grace,Steven	C6'03"	295	02/13/1979	3	Arizona
84	Hamilton,Lawrence	WR6'03"	205	08/31/1980	2	StephenF.Austin

5. Create an emp\_mast and dept\_mast files containing following details,

empno:name:job:deptno  
deptno:deptname  
empno:basic:hra:conveyance:medical:prof. tax: PF:TDS

Generate the pay slip as shown below by passing the empno as parameter to the awk script.

Empno	1001
Deptno	10
Job	MGR
Currency	INR

---

Earnings

---

BASIC	10,000.00
HRA	4,000.00
CONVEYANCE	9,600.00
MEDICAL	5,000.00

---

Total Earnings : 28,000.00

---

Deductions

---

PF	1200.00
PROF_TAX	200.00
TDS	1600.00

---

Net Pay : 25000.00

---

Note : Check empno before passing to awk script.

1. Generate the report as given below by passing the deptno to awk script

Deptno : 10 Dname : Admin

---

Empno	Name	Job
1001	Ketan	MGR
1002	Sachin	CLERK

---

Note : Check deptno before passing to awk script

## Lab 9. Shell Script

Goals	Learn to write simple shell scripts
Time	150 min
Lab Setup	Telnet with Unix Server

### 9.1: Writing Shell-Scripts

1. Display the Primary and Secondary prompt. Change the primary prompt to your name: temporarily
- 2: As soon as you login, the prompt should be changed to your name: also the name of the home directory should be automatically displayed.
- 3: Check the content of the Environmental variable SHELL.
- 4: Try the below exercise and check the output.

**Note:** Type every line and press enter, do not type the entire code in a vi editor.

```
$continent="Africa"
$echo "$continent"
-----→ Africa
$sh
$echo "$continent"
-----→ No Response
$continent="Asia"
$echo "$continent"
-----→ Asia
$ctrl + d
$echo "$continent"
-----→ Africa
$sh
$echo "$continent"
-----→ No Response
$ctrl + d
```

- 5: Try the below exercise and check the output. (Export variables)

**Note:** Type every line and press enter, do not type the entire code in a vi editor.

```
$continent="Africa"
export continent
```

```
$echo "$continent"
-----→ Africa
$sh
$echo "$continent"
-----→ Africa
$continent="Asia"
$echo "$continent"
-----→ Asia
$ctrl + d
$echo "$continent"
-----→ Africa
```

- 6: Write a shell script that takes the user name as input and reports whether he / she has logged in or not.
- 7: Write a shell script to display the file name and its contents of all the files that is there in the current directory.
- 8: Write a shell script, which will take a file name as argument and check whether the file exists and display its access permissions for user.
- 9: Pass three numbers as command line arguments and display the largest number in the given three numbers.
- 10: Write a shell script which will accept a pattern and a file name. The pattern will be searched in the file provided. Display appropriate messages and perform necessary validations on file.
- 11: To create a menu program for a) creating a file, b) Creating a directory, c) copying a file, d) moving a file. (use functions)
  - a. If the file exists already give the appropriate message
  - b. If the dir exists already give the appropriate error message
  - c. Source file should exist if not give a message, It should have read permission if not another message, Destination file either there or not, if not there then create it and copy it. If there, then ask whether to overwrite or not, if yes then overwrite it or else give a message file exists already and not overwritten.
- 12: Write a function yesno() to display question to user and accept answer as y/n. If answer to the question is y the function should return 0 otherwise 1.  
Use yesno functions for asking different questions. Question will be passed as parameter to the function.  
Accept filename from user check whether it is file or directory. Use yesno() function to display question do you really want to delete file? If the ans is y, then delete the file or directory.

- 13: Write a shell script to store names of four employees and check whether those employees are currently logged in or not. Display appropriate message.
- 14: Accept the user's first and last name and the echo the entire name along with some suitable comment.
- 15: List all files that have been modified today.
- 16: Display long listing of only the regular files in the current directory.
- 17: Display details of all files in the 2 "paths" accepted from user. The display should be screen by screen.
- 18: Let the script display its name and its PID.
- 19: Get the concatenated o/p of 2 files into a third file: Take 3 command line arguments: The first argument is the name of a destination file, and the other two arguments are names of files whose contents are to be placed in the destination file.

**Stretched Assignments:**

- 20: Write a menu driven shell program to:
  - a. Display calendar of current month
  - b. Search for a pattern in all the files/subdirectories from current directory.
  - c. Count the no. of directories / sub directories in current directory
- 21: Display day of week for a given date. (ddmmmyyyy)  
If day is Monday, display message "Monday Blues"  
Friday display message "yeh! It's week end."  
Similarly display different messages for each day of the week.
- 22: Display the contents of all .lst files in the current directory.
- 23: Design a simple calculator, which will add/subtract/multiply/divide 2 numbers.  
eg. cal 10 20 + will give o/p as 30.
- 24: For a student file with the following fields, rollno, name, marks, Generate 2 files 'Pass' and 'Fail' containing records of student who have passed or failed. Also count the number of students who have passed or failed.
- 25: Accept a date string from terminal and display employees born after the input date.

## Lab 10. Source Code Control System

Goals	<ul style="list-style-type: none"> <li>• Learn the concept of version management</li> <li>• Understand the steps involved in version management using SVN as a tool</li> </ul>
Time	60 min
Lab Setup	Linux Operating System and SVN client

### 11.1: Repository Creation

This is a Linux Administrator task and assumes that he has created the repository at location `"/home/demo_project_repository"`.

### 11.2: Initial Project Setup

#### Option 1: Using System administrator/ Normal User (If the project is newly started)

Creating a folder

```
[root@pace ~]# svn mkdir
file:///home/demo_project_repository/Analysis -m "Creating
Project Analysis directory"
```

Committed revision 1.

Deleting a folder

```
[root@pace ~]# svn rm
file:///home/demo_project_repository/Analysis -m "Removing
Project Analysis directory"
```

Committed revision 2.

Checking log file

```
[root@pace ~]# svn log file:///home/demo_project_repository
-----
r2 | root | 2013-09-30 09:47:17 +0530 (Mon, 30 Sep 2013) | 1 line
Removing Project Analysis directory
-----
r1 | root | 2013-09-30 09:46:49 +0530 (Mon, 30 Sep 2013) | 1 line
Creating Project Analysis directory
-----
```

#### Option 2: Using client (if the directory structure of project is already defined)

**Note: - This activity is call project import**

Initial project directory structure at client side

```
[testuser1@pace project]$ tree -a
.
|-- Analysis
|   |-- ReadMe.txt
|   '-- Software_Requirement_Specification.docx
|-- Coding
|-- Design
|   |-- High_Level_Design_1.2.docx
|   '-- Low_Level_Design_1.2.docx
|-- Query_Tracking_Sheet
|-- Testing
|   '-- test_case-lv1.0.xls
`-- Timesheet
```

Importing the Initial Project to Repository

```
[testuser1@pace project]$ svn import /home/testuser1/project
file:///home/demo_project_repository -m 'Initial import'
Adding          /home/testuser1/project/Query_Tracking_Sheet
Adding          /home/testuser1/project/Coding
Adding          /home/testuser1/project/Analysis
Adding          /home/testuser1/project/Analysis/Software_Requirement_Specificat
ion.docx
Adding          /home/testuser1/project/Analysis/ReadMe.txt
Adding          /home/testuser1/project/Timesheet
Adding          /home/testuser1/project/Design
Adding          /home/testuser1/project/Design/High_Level_Design_1.2.docx
Adding          /home/testuser1/project/Design/Low_Level_Design_1.2.docx
Adding          /home/testuser1/project/Testing
Adding          /home/testuser1/project/Testing/test_case-
lv1.0.xls

Committed revision 3.
```

The project was imported to the SVN, now delete the local directories

**11.3: Creating the working copy of project (Check out)**

Login to the client, go to the folder where working copy is to be created and fire the following commands

```
[testuser2@pace project]$ ls -a
. .
[testuser2@pace project]$ svn checkout
file:///home/demo_project_repository
```

```

A   demo_project_repository/Query_Tracking_Sheet
A   demo_project_repository/Analysis
A
demo_project_repository/Analysis/Software.Requirement.Specification.docx
A   demo_project_repository/Analysis/ReadMe.txt
A   demo_project_repository/Coding
A   demo_project_repository/Timesheet
A   demo_project_repository/Design
A   demo_project_repository/Design/High_Level_Design_1.2.docx
A   demo_project_repository/Design/Low_Level_Design_1.2.docx
A   demo_project_repository/Testing
A   demo_project_repository/Testing/test_case-lv1.0.xls
Checked out revision 3.
[testuser2@pace project]$ tree ./demo_project_repository
./demo_project_repository
|-- Analysis
|   |-- ReadMe.txt
|   `-- Software.Requirement.Specification.docx
|-- Coding
|-- Design
|   |-- High_Level_Design_1.2.docx
|   `-- Low_Level_Design_1.2.docx
|-- Query_Tracking_Sheet
|-- Testing
|   `-- test_case-lv1.0.xls
`-- Timesheet

```

#### 11.4: Modifying, updating, creating new file in the project

Login to the client, go to the folder where working copy is stored, do the changes and fire the following commands

```

[testuser2@pace demo_project_repository]$ cd Analysis
[testuser2@pace Analysis]$ touch SignOff.doc
[testuser2@pace Analysis]$ touch Open_Item.doc
[testuser2@pace Analysis]$ vi ReadMe.txt
[testuser2@pace Analysis]$ svn status
?     SignOff.doc
?     Open_Item.doc
M     ReadMe.txt
[testuser2@pace Analysis]$ cd ..
[testuser2@pace demo_project_repository]$ svn add
Analysis/SignOff.doc Analysis/Open_Item.doc
A         Analysis/SignOff.doc
A         Analysis/Open_Item.doc
[testuser2@pace demo_project_repository]$ svn commit -m 'User
Confirmation Document Added'
Adding     Analysis/Open_Item.doc
Adding     Analysis/ReadMe.txt
Adding     Analysis/SignOff.doc
Transmitting file data ...
Committed revision 4.

```

### 11.5: The lock-modify-unlock Cycle

Login to the client with **user1**, go to the folder where working copy is stored, lock the file on which **user1** want to work

```
[testuser1@pace demo_project_repository]$ svn lock
Analysis/ReadMe.txt -m "Locking a File for Change"
'ReadMe.txt' locked by user 'testuser1'.
```

Login to the client with **user2**, go to the folder where working copy is stored, try to change the same file which has been locked by user1.

```
[testuser2@pace demo_project_repository]$ cd Analysis
[testuser2@pace Analysis]$ vi ReadMe.txt
[testuser2@pace Analysis]$ svn commit -m 'User Confirmation
Document Added'
Sending      Analysis/ReadMe.txt
Transmitting file data .svn: Commit failed (details follow):
svn: User testuser2 does not own lock on path
'/Analysis/ReadMe.txt' (currently locked by testuser1)
```

Now user1 can either perform the changes in the file and commit it (This will automatically release the lock) or just unlock the file so that user2 will start working on it.

```
[testuser1@pace demo_project_repository]$ svn unlock
Analysis/ReadMe.txt
'ReadMe.txt' unlocked.
```

Now user2 can perform the changes and commit it.

```
[testuser2@pace Analysis]$ svn commit -m 'User Confirmation
Document Added'
Sending      Analysis/ReadMe.txt
Transmitting file data .
```

### 11.6: <TODO> Working on SVN

- 1) Create four folders in SVN repository with following structure

```
| -- YourUserName
|   |-- Research
|   |-- Development
|   |-- RND
```

- 2) Remove the directory RND from the repository
- 3) Add two files code.txt under folder Research and process.doc under folder Development
- 4) Lock file code.txt , change the contents of the file and update the changes to SVN server