Working Environment

➢ virtual terminals or consoles(tty1 - tty12)

➢ xterm – GUI terminal – pseudo terminal

(to be launched in GUI)

➢ switching between virtual terminals

(ctrl-alt-F1 <-> ctrl-alt_F2 <-> ctrl-alt-F3<-
> ......)

➢ both terminals present an interactive shell that
provides a CLI

➢ demo

Working Environment

➢ virtual terminals or consoles(tty1-tty12)

· by default, 6 virtual consoles are available

· may be increased or decreased by editing

the /etc/inittab(as root user and with expertise)

· tty1 – tty6 are the default consoles

· tty7 is dedicated for the GUI – X-Window Application

· tty10 is dedicated for the kernel messages

(messages are generated in the kernel-space

using printk() - core of the OS and modules)

➢ demo

Working Environment

- xterm – GUI terminal – pseudo terminal (to be launched in GUI)

- switching between VT – GUI (ctrl-alt-F1(2,3..6) <-> ctrl-alt_F7)

- xterm-based terminals can launched using the terminal icon found under applications

- demo

Shell prompt

- ➢ interactive shells provide a shell prompt

- ➢ normal user – shell prompt ends with $, typically

- ➢ super user   - shell prompt ends with #, typically

- ➢ user can enter/edit commands at the shell prompt accessing previous commands

- ➢ exit or ctrl-D – to exit the current interactive shell

- ➢ su <user-name> – switching between users

- ➢ demo

Unix Shell

➢ a shell is an active instance of a program (a process) that takes commands typed by the user and calls the OS(using system library routines) to run those commands

➢ a shell acts as a wrapper around the OS – hence, known by the term shell

➢ ideally, the system library must be known as the shell – well, may be or may be not !!!

Unix Shell

- shell is a special utility that plays several roles – it is a versatile utility

- command interpreter

- command editor

- job controller

- programming language interpreter

- CLI – command-line interface to the OS

Unix Shell – who needs it ?

➢ it is still the best interface to administer and monitor an Unix/Linux system

➢ system administrators and system-software developers must master this environment and use this environment

➢ well, that is what the Gurus say !!!

➢ as a student, use the shell offered by the virtual terminal – that is the best environment to learn

Unix Shell – who needs it ?

➢ no matter how good your GUI interface is, still you need the power of shell

➢ it can increase the speed and efficiency of your usage

➢ somethings cannot be done using GUI or at the best, can be done incorrectly – shell is the best bet for such work

➢ proof – Mac OS X 10.x and recent Linux distributions – used as Desktop Operating Systems

Different shells

- chsh -l

- cat /etc/shells

- in the above cases, you will see several shells - the absolute pathnames of the various shells will be displayed

- The above are supposed to be the shells supported on your system – they may or may not be installed on your system !!!

Different shells

➢ you can check the availability of a particular shell by typing the full pathname :

   e.g. /bin/bash or /bin/tcsh

Different shells

➤ you can change your shell temporarily by
  just typing the full pathname of the new
  shell:

   /bin/tcsh or /bin/csh

➤ you can also change your default shell
  permanently using :

   chsh -s /bin/tcsh

   logout and login again – you will see the new

   default shell is effective – *not recommended*

Different shells....

➢ why so many shells ?

➢ Bourne shell(oldest – AT&T)

/bin/sh – It was developed by Steve Bourne

➢ It does not have many features and hence, newer shells compatible to this have been created

➢ Default shell for AT&T System V Release 2

Different shells....

- ➢ Korn shell(newer - AT&T)

- ➢ Popular among SVR4 systems and SVR4 based variants – default shell in many commercial Unix systems – HP-UX and Solaris

- ➢ It is compatible with Bourne shell and has more features – like, aliasing, command history and many more

- ➢ It was proprietary for a very long time – not proprietary anymore

Different shells....

➢ C shell(developed at UC,Berkeley and released with BSD)Based on C programming language style

➢ It is not compatible with Bourne shell – uses different syntax and has several good features

➢ Widely available in Unix variants

➢ Korn shell was created to counter C shell

Different shells....

➢ Linux uses a variant of C shell – TENEX C shell(tcsh)

➢ In fact, csh in Linux is a soft-link to /bin/tcsh

➢ Supports additional features like – command line editing and command line completion

Different shells....

- Bash shell (Bourne-Again shell) has the feel of Bourne and Korn shells and incorporates features from C and Korn shells

- compatible with Bourne shell

- bash(/bin/bash) is the most popular shell in Linux and the default shell for users as well

- it is POSIX compliant with POSIX1003.2

- Bourne shell is open-source and released under GNU/GPL

Bash shell features

- compatible with Bourne shell

- job control

- history list

- command-line editing

- aliases

- functions

- arrow keys for command editing

Which shell to choose..

➢ Korn or Bash is good

➢ Pick-one (mostly, the native) and stick to it

➢ If you like C style syntax, use tcsh

➢ Gurus say that the best choice will be Bash
  or Korn – C shell has the problem of using
  its own syntax that is not compatible with
  Bourne, Korn and Bash

➢ We will only be looking at Bash, in Linux

➢ Explore more, if you need to !!!

Unix sessions

➢ an interactive Unix session involves a
  shell and the associated terminal

➢ login shell(VT) – login, interactive session

➢ non-login(xterm) – non-login, interactive session

➢ echo $0 – print the name of the current shell

➢ echo $SHELL - typically, prints the name of
  the default shell

standard streams

- standard input - by default, associated with the virtual-terminal or pseudo-terminal – 0

- standard output – same as, above – but, 1

- standard error – same as, above – but, 2

- 0,1 and 2 are the open file-descriptors of any process

- any interactive process, by default, has the above settings true

Redirecting standard streams

- file descriptors that normally define the standard streams may be redirected to point to other file or pipe streams

- shell is capable of setting up redirections for the utilities and applications launched from the shell command prompt

- typically, filters like, cat, wc, grep, sed,etc,. benefit from redirection – examples will be seen in the slides to be followed

accessing documentation

- man pages - provides brief info. - a good reference -does not teach – not a tutorial

- info pages – provides detailed information – in some cases, same as corresponding man page

- other resources – books/internet – there are some

- good references on Unix/Linux shell scripting

man pages

- there are several man sections – like chapters of a book

- man ls – will provide the man page from the first section – section 1

- man 1 ls – specifically from section1

- man 1 kill – specifically from section 1

- man 2 kill – specifically from section 2

- man 3 printf – specifically from section 3

- sections – 1-8 ; there are several sections

man pages....

➢ man -a kill – list pages from each section – one after the other

➢ you can easily move around a man page

➢ page-up/page-down or up/down arrows to scroll

➢ q to quit

➢ /search string ; n to repeat the search - forward

➢ ?search string : N to repeat the search - backward

info pages

- info ls

- more detailed information

- hyper-text based documentation

- q to quit

- nodes of information

- use page-up/page-down

- use u or n(move between nodes)

files – pathnames

- just a filename (file1)

- current-directory based (./file1)

- relative-based (dac1/file1)

- home-directory based (~/file1) – can be used with shell and shell scripting only

- do not use ~ in system calls or library calls

- absolute pathname (/home/dac1/file1)

- does it matter ? yes it does

key directories in the root file-system

- /bin

- /sbin

- /etc

- /boot

- /home/dac1, /home/lrde1...

- /usr

- /lib

key directories in the system...

- /root
- /tmp
- /proc
- /sys
- /var
- /usr/src

key directories explained

➤ File Hierarchy Standard(FHS) – sets the rules for layout of the Unix file system hierarchy and its contents

➤ Linux community understands the importance of standards, and all major distributions support the standard

➤ Full FHS will be available at www.pathname.com/fhs   - explore if needed !!!

key directories explained

➢ We will see some of the important aspects here

➢ Files/directories are treated as shareable or non-shareable

➢ Shareable - are files/directories that may be seen by other hosts(executable/documentation)

➢ Non-shareable – are file/directories that must not be seen by other hosts(configuration files/start-up files)

key directories explained

➤ Files are treated as static and variable

➤ Variable - are files that may be changed frequently(/var/log/messages or any user data-files)

➤ Static – rest of the files are normally unchanged or less frequently changed(utilities, tools and other user created binaries)

key directories explained

- /bin – essential binaries(ls,date,cp,mkdir,ps,...)

- /sbin – most essential system administration

   binaries(fsck,fdisk,init,...)

- /etc – most essential configuration files like inittab.fstab,...

- /lib – most essential libraries and kernel modules

   are located here(/lib/modules/<kernel-version>/)

- /boot – contains boot-loader files and kernel images

key directories explained

- /root – recommended default system administrator's home directory

- /home – contains home directories of individual users(/home/corporate)

- /tmp – contains temporary files that may be deleted at every system boot

- /opt – third-part software packages may be installed here

key directories explained

- /usr/bin – not so important utilities

- /usr/sbin – not so important system administration utilities

- /usr/lib – libraries that support add-on software packages

- /usr/share – shareable directory with documentation and other executables that may be shared with others hosts over NFS

key directories explained

- /usr/local/bin – locally provided utilities and binaries

- /usr/local/sbin – locally provided administration utilities

- /usr/local/doc – locally provided documentation

- /usr/src – may contain source-code of utilities/ kernel

a tour of Bourne again Shell(bash)

!!! get ready to try the following

   on your systems !!!

➢ per se,   shell, shell programming and shell scripting are not difficult

➢ yet, they demand a strong foundation in Unix/Linux system concepts and system programming skills

pattern matching using wild cards

➢ ls -l file*(use touch to create file1,file2,file3..)

  (any no of any characters – including none)

➢ ls -l file? (any one character)

➢ ls -l file[123]

➢ ls -l file[1-3]

➢ ls -l file[!1-3] or ls -l file[^1-3]

➢ shell interprets wild-card characters, the utilities

  do not interpret the wild-cards

Escaping pattern matching

➢ ls -l 'file*'

➢ ls -l "file*"

➢ ls -l file\[123\]

➢ escaping using single or double quotes

➢ escaping using backslash

➢ the escape characters suppress the pattern matching by the shell

➢ each escape character is useful in a different way

Escaping pattern matching

➢ Single-quotes – escapes everything within - meaning, shell does not do pattern matching

➢ Double-quotes – does not escape parameter substitution,command substitution and arithmetic substitution – meaning, shell does partial pattern matching

➢ Double-quoted string can be included in double-quotes

➢ Single-quoted string can be included in double-quotes

shell variables – user defined

➢ var1=0 (can be used without explicit declaration)

➢ var2=5

➢ echo $var1

➢ echo $var2

➢ var3=$var1+$var2 (treated as strings)

➢ echo $var3 (result is a string)

➢ user defined are just that – defined by users

➢ by default, variables are of string type

shell variables – user defined..

➤ declare -i var1=0 (explicitly declare as integer)

➤ declare -i var2=5

➤ echo $var1

➤ echo $var2

➤ var3=$var1+$var2 (still this will be a surprise)

➤ echo $var3

➤ ((var3=var1+var2)) or ((var3 = $var1+$var2))

➤ does it make any difference ? Try to fix the problem and understand how things work !!!

shell variables – user defined..

➢ exporting user defined variables

➢ create a sub-shell (just type bash)

➢ echo $var1(a copy is not provided to the sub-shell)

➢ echo $var2

➢ leave the sub-shell, run declare -x var1 var2
and again enter the sub-shell

➢ does it make any difference ? Yes – a copy of the
exported variables is given to the sub-shell

shell variables – user defined..

➢ by default, user-defined variables are not exported – you must explicitly export it

➢ unset – delete a variable – see below

➢ unset var1 var2

➢ echo $var1

➢ echo $var2

➢ only the specific shell is affected – other shells are unaffected

shell variables – predefined

- environment variables

- their usage is predefined by the shell

- normally exported

- PATH,PS1,PS2,PWD,HOME,SHELL,USER, TERM..

- let us look at PATH

- you will get to know more about others, if you need them; say, a particular application may need them

shell variables – special variables

➤ also known as special parameters

➤ their usage is predefined by the shell and the values are set during run-time to reflect the current state of the shell

➤ do not confuse them with environment variables

➤ ?, $, @, #, 0, 1, 2, 3, 4.........

➤ demo - echo $$ and echo $?

shell variables – special variables..

➢ special variable ? - used to store the exit status of
   the previous command or return value of a
   function execution

➢ more on other special variables when we discuss
   functions and scripting

➢ used for passing parameters to a shell script

➢ used for passing parameters to a shell function

➢ very useful in shell scripting

shell – command history

- use up/down arrows to see what you have in

   the history and use it without typing the commands again

- history - try the history command

- !no (!1000)

- no. is obtained from the history list - using history command

- length of the history maintained by the shell can be controlled by a predefined(environment) variable

command-line editing

- .bash_history – when you exit from a shell,it saves the list in this file

- when the shell is launched again, it is read from

  the history file – this cycle continues

- the name of this file is defined by HISTFILE

- for vi editing mode on command line

  set -o vi //enable vi editing mode

- input-mode vs command-mode

- default mode is input mode – refer to vi slides

command-line editing

- Move left one character – h

- Move right one character – l

- Move right one word – w(?)

- Move left one word -b(?)

- Beginning of next non-blank word – W

- Beginning of previous non-blank word - B

command-line editing

➤ Move to end of current word – e(?)

➤ Move to end of current non-blank word – E

➤ Move to beginning of line – O

➤ Move to first non-blank character in a line - ^

➤ Move to end of line - $

➤ Experiment with above – they are peculiar at times !!!!

command-line editing

- To get back to input mode :
- Insert text before current character – i
- Insert text after current character – a
- Insert text at the beginning of line – I
- Insert text at the end of line – A
- Overwrite current text – R
- Overwrite only current character - r

command-line editing

- ➢ Delete one character backwards – dh
- ➢ Delete one character forwards – dl
- ➢ Delete one word backwards – db
- ➢ Delete one word forwards – dw
- ➢ Delete one non-blank word backwards – dB
- ➢ Delete one non-blank word forwards – dW
- ➢ Delete to end of line – d$
- ➢ Deleting to beginning of line - d0

command-line editing

➢ Move backward one-line – k or -

➢ Move forward one-line – j or +

➢ /string – search backward for string

➢ ?string – search forward for string

➢ Repeat the search backwards – n

➢ Repeat the search forwards - N

shell – command substitution

- file_list=ls

- echo $file_list

- file_list=`ls` or file_list=$(ls)

- echo $file_list

- date_var=date -u

- echo $date_var

- date_var=`date -u` (back tick not single quotes)

shell – arithmetic substitution

➢ Var1=10 ; var2=20;

➢ echo $((var1+var2))

➢ echo $((var1+1))

➢ echo $((var1*2 + var2*2))

➢ echo ((var1*2 + var2*2))

➢ echo $((var++))

➢ echo $((++var))

shell – arithmetic substitution..

- ➢ addition
- ➢ subtraction
- ➢ multiplication
- ➢ division
- ➢ post-increment operation
- ➢ increment operation
- ➢ refer to the man bash for more details

shell – aliases

➢ create an alias for a standard command with frequently used options

➢ alias ll='ls -l' // use single-quotes

➢ alias my_ps='ps -e -o pid,ppid,uid,gid,tty,cmd'

➢ now type ll or my_ps and your job is done

➢ aliases are not exported

shell – aliases...

- just type alias on you system // what do you observe ?

- alias ls='ls -l' // use single-quotes without fail

- alias ps='ps -e -o pid,ppid,uid,gid,tty,cmd'

- now try if the above alias commands work !!

- unalias ll // to remove the alias

- try ll   // try the alias command again

shell – built-ins

➢ shell is a program like any other

  (may be a bit special ??)

➢ /bin/bash or /bin/tcsh

➢ shell has code built-in for certain commands

➢ they are known as built-ins

➢ faster to execute, but bloats the shell

➢ only essential ones are kept so !!!

shell – built-ins..

- cd

- export

- declare

- exit

- echo

- alias

- unalias .....the list quite long

shell – built-ins..

- ➢ just a give a thought whether cd or exit built-ins can be implemented as external commands
- ➢ aliases are given preference over built-ins

command separators

- command1 ; command2; command3

- command1

    command2

    command3

- the above two are the same (newline has two roles)

- using \ to escape the newline being interpreted as a command separator – useful when typing lengthy commands on command line or in scripts

command separators...

➢ ls -lR / | \

grep "file" | \

wc -l

➢ when the command is incomplete, the shell throws the secondary command prompt

command chaining

- command1 || command2

- command1 && command2

- you can add your logic as below

- ls -l file1 && rm -ri file1

- a more useful example is below :

    make && make modules_install && \

    make install //used in kernel recompilation

shell - functions

- function f1(){ echo "this is my first function";}

- echo $(f1) // command - substitution

- function f2(){ ((var2=var1+var2)); return $var2;}

- f2 //must execute like a command

- echo $? //return value is stored in ? special variable

- function f2(){ ((var2=var1+var2)); echo $var2; }

- f2 //what do you observe ?

shell - functions..

➤ functions can take parameters

➤ special variables 1,2,3,..... are used to handle function parameters also(scripts also need them)

➤ function f2(){ return $(($1+$2)); }; f2 3 5; echo $?

➤ if you use a function inside a script, the function parameters must be explicitly passed

➤ functions can be exported

shell - functions..

➤ functions can be written in shell start-up scripts

➤ functions take precedence over shell built-ins

➤ aliases take precedence over functions

➤ export -f f1 //export a function

➤ unset -f f1   //delete a function setting in a shell

➤ first step towards storing a set of commands

➤ in what way are they better than aliases ??

➤ fast compared to scripts – no sub-shell is needed

shell - redirection

- ls -lR / (use ctrl-C to terminate it)

- ls -lR / 1>file1 2>file2

- ls -lR / 1>file1 2>&1

- ls -lR / 2>&1 1>file1 (wrong)

- standard output is redirected to active file of file1

- standard error is redirected to active file of file2

- cat < /etc/passwd

- standard input is redirected to active file of /etc/passwd

shell - redirection..

- cal 7 2006

- cal 7 2006 > month.txt

- cal 8 2006 >> month.txt

- cal 17 2006 > month.txt

- cal 17 2006 >> month.txt 2>>errors

- >> redirection with the active of month.txt
  opened in write/append mode

shell - pipelines

➢ combine utilities to achieve more complex work

➢ cat /etc/passwd | less

➢ leads to grouping utilities

➢ leads to grouping processes doing the same job

➢ can be quite long

ls -lR /usr/include | grep "sigset_t" | less

➢ pipeline and filters combine to do complex work
which is the philosophy of Unix design

shell - pipelines..

➢ uses unnamed pipes

➢ concurrent processes are created

➢ uses redirection

➢ taken care by shell with the help of kernel
   (of course, how else ??)

➢ kernel uses process group, process group leader
   and process group id to support this concept

shell – flow-control structures

- ➢ for control structure

  for variable in list

  do

      command1; command2; ...... ;

      commandn;

  done

- ➢ differs from C language syntax

shell – flow-control structures..

- for control structure

  ```
  for x in hosts services

  do

      cat /etc/$x

   done
  ```

- list a set of system files under same directory

shell – flow-control structures..

- for control structure

  mkdir /root/backup

  for FILE in /etc/*.conf

  do

      echo "backing up $FILE..."

      cp $FILE backup/

  done

shell – flow-control structures..

➢ for control structure

for SERVICE in httpd ftpd

do

    /sbin/service $SERVICE status

done

shell – flow-control structures..

- ➢ for control structure – another type

   (more familiar)

   for ((a=1; a<=32; a++))

   do

      echo $a

   done

- ➢ similar to C language syntax

shell – flow-control structures..

- if control structure

  if control-command

  then

      command1; command2; ...;

      commandn;

  fi

shell – flow-control structures..

➤ if control structure

if who | grep -q dac1

then

   echo "dac1 has logged in"

fi

➤ the outcome of the command is used
   as a test condition by if

shell – flow-control structures..

➢ if control structure

  if test $A -gt 50   // test command evaluates

  then

     echo "too high"

  fi

  ➢ numeric comparison

  ➢ test is a command

shell – flow-control structures..

➢ if control structure

(does the same as test command)

if [$A -gt 50]   // [ ] command evaluates

then

   echo "too high"

fi

shell – flow-control structures..

> if control structure(another variation)

if control-command

then

    commands

 else

    commands

 fi

shell – flow-control structures..

- ➢ if control structure(another variation)

- ➢ if control-command

  then

      commands

  elif control-command

  then

      commands

  else

      commands

  fi

shell – flow-control structures..

➢ what can be used with test or [ ] commands ?

➢ -f file (file exists and is regular)

➢ -d file (file exists and is directory)

➢ -r file   (file exists and readable)

➢ -x file (file exists and executable)

➢ -w file (file exists and writable)

➢ check the man bash for more cases

shell – flow-control structures..

- what can be used in test or [ ] commands ?

- string1 == string2 (string comparison)

- string1 != string2 ("   "   )

- value1 -eq value2(numeric comparison)

- value1 -gt value2("   "   )

- value1 -lt value2("   "   )

- expression1 -a expression2   // and operator

- expression1 -o expression2 // or operator

shell – flow-control structures..

- ➤ while control structure

➤ while control-command

  do

      command1;command2;....;commandn;

  done

- ➤ control command can be test or [ ]

shell – flow-control structures..

- while control structure

- while [$(who | wc -l) -lt 100]

```
    do
        sleep 15;
    done
    echo "over 100 users are now logged in `date`"
```

shell – flow-control structures..

- while control structure

- A=0

- while [$A -lt 20]

  do

    ((A=A+1));

    echo $A

  done

shell – flow-control structures..

➢ while control structure

➢ A=0

➢ while [$A -lt 20]

   do

      ((A++));

      echo $A;

   done

shell – flow-control structures..

- while control structure

- A=0

- while [$A -lt 20]

  do

    ((++A));

    echo $A;

  done

shell – scripting

➢ at last, we are ready to write shell scripts !!!

➢ combine commands and programming constructs discussed earlier – write them into a file – the file is known as a shell script

➢ need an interpreter at the top of the script #!/bin/bash or #!/bin/tcsh

➢ A script file needs to be made script an executable

➢ explicitly – using chmod u+x file or chmod +x file

1

shell – scripting

1

- ➢ if no interpreter is mentioned at the top of the script then, the kernel uses the default depending on the system

- ➢ in Linux, it is /bin/bash

- ➢ other systems may have a different default shell

shell – scripting..

➢ you can do almost everything that you can

do on command-line(almost ??)

➢ can execute other scripts from a script

➢ can call exported functions and variables

➢ can source other scripts using source or .

➢ source ./script1.sh or . ./script1.sh

➢ sourcing is popular in start-up scripts and

also, can be used to include scripts from other
scripts

1

shell – scripting..

➢ scripts may be used for system management, installation or automation

1

➢ you can modify the shell start-up scripts to

manage your sytem

/etc/profile (system wide)

~/.bash_profile //separate for each user –

//login shell

~/.bashrc   //separate for each user –

//non-login shell

shell – scripting..

➢ /etc/bashrc (system wide)

➢ depending upon the way shells are invoked, different start-up scripts are invoked

➢ refer to man bash for more details on start-up scripts

shell – scripting..

- best way to learn is to practice

- practice changing start-up scripts

- practice writing your own scripts

- info bash – a very good guide

- Unix and Shell programming-Behrouz A.Forouzan and Richard F.Gilberg

- there are many such good texts

vi editor - basics

1

- ➢ vee-eye
- ➢ vi filename
- ➢ vi file1 file2
- ➢ modes – command,edit and ex mode(: mode)
- ➢ universally available
- ➢ several variants like vim, gvim....

vi editor - basics..

- ➢ edit mode

1

- ➢ i - insert before cursor

- ➢ I - insert at the start of the line

- ➢ a - append after the cursor

- ➢ A- append at the end of the line

- ➢ o - open a line after current line and insert

- ➢ O- open a line before the current line and insert

vi editor - basics..

➢ use one of edit mode commands to enter
   as per your convenience

➢ demo – try typing script

➢ use esc key to switch to command mode

➢ you can repeat the above as you wish

➢ R puts in you text replace mode – a special edit
   mode

vi editor - basics..

1

- ex(or :) mode – extended command mode
- entered by just pressing :
- :0 - beginning of the file
- :$ - to the end of the file
- :n - any specific line
- :w - save
- :wq – save and quit

vi editor - basics..

- ➤ ex(or :) mode - extended command mode

- ➤ :w! - force write(even if the file read-only)

- ➤ :x – save and exit (also, ZZ)

- ➤ :q - quit vi

- ➤ :q! - quit without saving

- ➤ :set nu – line numbering

- ➤ :set nonu – disable line numbering

vi editor - basics..

1

- ➤ ex(or :) mode

- ➤ :w filename – save as

- ➤ :e filename – edit a new file

- ➤ :spl – split vi into 2 windows

- ➤ :clo – close current window

- ➤ ctrl-w ctrl-w to move between windows

vi editor - basics..

➢ In ex(or :) mode

➢ /pattern/ to search – regular expression search

1

➢ use n to keep moving from one find to the other

➢ :s/reg-exp/replacement-string/

➢ :s/reg-exp/replacement-string/g

➢ :1,$/reg-exp/replacement-string/[g]

➢ :%s/reg-exp/replacement-string/[g] – global in a file

➢ [g] is an option – global in a line, not in a file

vi editor - basics..

➢ in command mode, try the following

➢ dd – delete current line

➢ 2dd(ndd) – delete 2 lines from current line

➢ yy – copy(yank) current line

➢ 2yy(nyy) – copy 2 lines from current line

➢ p – copy below current line

➢ P – copy above current line

vi editor - basics..

1

- ➢ in command mode, try the following :
- ➢ begin marking a block – ma(b,c..)
- ➢ end copying a block - y`a
- ➢ end deleting a block – d`a
- ➢ after this you can use p or P to paste

  the block of data at the appropriate place in the file
- ➢ this is a very useful command while coding

vi editor - basics..

➢ in command mode, try the following :

➢ using named buffers for copying and pasting

➢ start marking a block - ma

➢ end marking and copying a block – "ay`a

➢ after this you can use "ap or "aP to paste

   the block of data at the appropriate place in the
   file

➢ this is a very useful command while coding

vi editor - basics..

- in command mode, try the following :

- using named buffers for copying and pasting

1

- start marking a block - ma

- end marking and copying a block – "by`a

- after this you can use "bp or "bP to paste

  the block of data at the appropriate place in the
  file

- this is a very useful command while coding

vi editor - basics..

1

- ➢ in command mode, try the following :

- ➢ using named buffers for copying and pasting

- ➢ start marking a block - ma

- ➢ end marking and copying a block – "cy`a

- ➢ after this you can use "cp or "cP to paste
  the block of data at the appropriate place in the
  file

- ➢ this is a very useful command while coding

basic commands

1

- ➢ cp
- ➢ mv
- ➢ touch
- ➢ rm
- ➢ mkdir
- ➢ rmdir
- ➢ ln(without or with -s)

basic commands..

- find // find is very powerful – use the man find

- uname

- id

- which

- whereis

- type

- file

basic filters

- ➢ less
- ➢ grep
- ➢ sort
- ➢ cut
- ➢ sed
- ➢ wc
- ➢ bc

basic filters..

- less (we have seen before)

- sort

1
- cat /etc/passwd | sort

  - sort /etc/passwd

  - sort -r /etc/passwd

  - sort -f /etc/passwd

  - sort -t':' +0 -1 /etc/passwd (field separator is :)

  - sort -t':' -n +2 -3 /etc/passwd (numeric sorting)

  - we can change the delimiter - space and tab are default delimiters

basic filters..

➤ wc

➤ wc   /etc/passwd

➤ wc -l /etc/passwd

➤ wc -c /etc/passwd

➤ wc -w /etc/passwd

basic filters..

➢ cut (by default, field separator is a tab)

➢ cut -f1 /etc/passwd

➢ cut -f1 -d':' /etc/passwd (delimiter is :)

➢ cut -f1,2 -d':' /etc/passwd (cutting specific fields)

➢ ls -l /home/corporate | cut -c1-10 (cutting specific characters)

➢ can cut characters

➢ can cut fields

basic filters..

- bc - it is a calculator

- supports multiplication(*), division(/), modulus(%), and power(^)

- by default, base is 10

- scale – no of digits after the decimal in a floating-point number

- bc   -options   arguments

- echo 'expression' | bc

- echo 'scale=2; 3.0 / 2' | bc   (different syntax)

basic filters..

➢ for further info. , refer to - man bc

➢ break your head or pull your hair – depending
upon your convenience

1

echo

- echo command

- echo string

- echo 'string'   - what does this do ?

- echo "string" - what does this do ?

- echo var1

- echo $var1

- echo '$var1'

- echo "$var1"

echo

- echo $((2 * 2))

- echo   "$((2 * 2))"

1

- echo   '$((2 * 2))'

- echo   ((2 * 2))

- echo   "((2 * 2))"

- echo   '((2 * 2))'

- Get used to shell syntax and shell escaping

array variables

- ➢ declare -a array1
- ➢ array1=(1 2 3 4 5)
- ➢ echo $array[0]
- ➢ echo $array[1]
- ➢ echo ${array[1]}
- ➢ echo ${array[@]}
- ➢ echo ${#array[@]}

regular expressions

➢ not shell expansion/not understood by shell

➢ made up of atoms(characters) and operators(meta-characters)

➢ understood by grep,sed,vi,emacs.....

➢ *,^,$,.,character class – standard or basic regular expressions

➢ extended regular expressions are +,?,|, {repetitions}

➢ demo with grep

1

regular expressions

- ➢ grep -F 'root' /etc/passwd   //string only match
- ➢ grep '^root' /etc/passwd   //basic reg.exp. match
- ➢ grep -E 'bash$ | tcsh$' /etc/passwd //extended
                                    //reg.exp. match
- ➢ grep -e pattern file(s)
- ➢ grep is a powerful filter
- ➢ look once into /etc/profile – some good examples of grep are available

grep and regular expressions

➢ grep works on a line by line basis, applying the regular-expression on each line

1

➢ if a match occurs, the line is output to stdout

➢ grep is mainly used for searching file(s) for specific pattern(s)

grep - options

- ➢ grep -c   // count no. lines that contain a match

- ➢ grep   -i //   case insensitive matching

- ➢ grep   -l //   print list of files that contain a match

- ➢ grep   -n //   show line no. before the line

- ➢ grep   -s //   silent mode

- ➢ grep   -v //   inverse output – non-matching lines

- ➢ grep   -w //   match entirely, as a word

- ➢ grep   -x //   match entirely, as line only

grep - options

➢ grep   -f filename

   //filename can contain several reg.expressions

   //to be matched

➢ grep   -r   or   grep   -R

   // test files recursively in a directory

➢ use find for searching for files and grep for

   searching patterns in files

➢ combine them and you get a powerful utility

➢ find ~ -type f -exec grep -l 'reg.exp' {} \;

sed and regular expressions

- ➤ sed – stream editor

- ➤ can do what grep can do and much more

- ➤ can delete lines on the fly

- ➤ can search and replace

- ➤ cat /etc/passwd | sed '/^babu.*:/d' > passwd.tmp

- ➤ mv passwd.tmp /etc/passwd

- ➤ ls -l /root | sed 's/ */ /' or ls -l /root | sed 's/ */ /g'

sed

> sed – stream editor

> cat /etc/passwd | sed

> cat /etc/passwd | sed 'd'

> sed -e 'd' /etc/passwd

> cat /etc/passwd | sed 'p' | less

> cat /etc/passwd | sed -n 'p' | less

> cat /etc/passwd | sed -n 'p' | head -n 10

> cat /etc/passwd | sed -n 'p' | tail -n 10

1

sed

➢ cat /etc/passwd | sed '1d' | head -n 20

➢ cat /etc/passwd | sed   '1,5d' | head -n 20

1

➢ cat /etc/passwd | sed   '10,5' | head -n 20

➢ cat /etc/passwd | sed   '1,5!d' | head -n 20

➢ cat /etc/passwd | sed 's/root/normal/'

➢ cat /etc/passwd | sed 's/root/normal/g'

➢ cat /etc/passwd | sed 's/root/normal/gi'

➢ cat /etc/passwd | sed 's/\/bin\/bash$/\/bin\/zsh/'

Additional slides on scripting

1

- ➢ try echo $BASH_VERSION
- ➢ use ps -e | less to check the current processes
- ➢ try ctrl-c or ctrl-z or ctrl-\(^c or ^z or ^\)
- ➢ use ps -e | less to check again
- ➢ foreground and background processes
- ➢ commands : jobs,fg and bg to manage jobs
- ➢ use ps -ej and ps -e j – what do you observe ??

Additional slides on scripting

1

- ./prog   //launching a foreground job
- ./prog&   //launching a background job
- what is the difference ?
- use ctrl-c or ctrl-z or ctrl-\ to test !!!!
- use fg and bg commands with them

Additional slides on scripting

1

- ➢ ls -l filename
- ➢ ls -ld dirname
- ➢ ls -li filename
- ➢ ls -l dirname
- ➢ ls -lr dirname
- ➢ ls -li filename(dirname)
- ➢ ls -lR /home/dac(or any starting directory)

Additional slides on scripting

➢ use touch to create a file

1

➢ ls -lu filename

➢ ls -lc filename

➢ ls -l filename

➢ use cat filename then, repeat ls commands

➢ use echo "123" > filename; repeat ls commands

➢ use chmod +x filename; repeat ls commands

Additional slides on scripting

- try echo $SHELL

1

- try chsh /bin/tcsh – what do you observe ??

- type exit

- type logout

- try ctrl-d

Additional slides on scripting

1

- ➤ Try tty on a virtual terminal
- ➤ Try tty on a xterm
- ➤ What do observe ??
- ➤ Try echo "123" > /dev/tty1(terminal device name)
- ➤ Try echo "123" > /dev/pts/0(terminal device name)

Additional slides on scripting

- ➢ cut -d':' -f1 < /etc/passwd | sort

- ➢ cat /etc/passwd | cut -d':' -f1 | sort

- ➢ grep 'bash' /etc/passwd | \

    awk -F':' '{ print $7 }' | sort -u

- ➢ awk -F':' '/bash/{ print $7 } | sort -u

- ➢ awk is a powerful utility supporting scripting

    and requires per se a thorough study

Additional slides on scripting

➢ Background jobs and I/O

➢ Try cat&

➢ Type jobs

➢ Stopped by SIGTSTP

➢ Another use of signals

➢ Bring it to foreground using fg and try again

Additional slides on scripting

➢ Try cd /tmp followed by pwd

➢ Get back to the home directory using cd

➢ Try (cd /tmp) followed by pwd

➢ What do you observe ??

➢ ( ) creates a new sub-shell

➢ Just another option

Additional slides on scripting

1

- ➢ echo $HISTFILE
- ➢ echo $HISTSIZE
- ➢ echo $HISTFILESIZE
- ➢ cat history | less
- ➢ Ctrl-R (reverse search in history)
- ➢ Get used it by experience
- ➢ Using TAB for command completion

Additional slides on scripting

➢ history management

1

➢ fc -l

➢ fc -l grep

➢ fc grep

➢ fc   no. // no. obtained from history

Additional slides on scripting

➢ set -o

➢ set -o noglob

➢ set +o noglob

➢ set -o noclobber

➢ set +o noclobber

➢ set -o nounset or set +o nounset

➢ -o and +o are counter-intuitive

Additional slides on scripting

- var1=10
- echo '$var1'
- echo '$var1'
- echo "$var1"
- echo \$var1
- Strong(' ' and \) and weak(" ") escaping

1

Additional slides on scripting

- PS1

- PS2

- echo $PS1

- echo $PS2

- PS1='\s--\w--\u'

- PS1='\@--\w--\u'

- PS1='\@--\d--\w--\u'

- try and set you primary prompt to display date, time, current working directory and your user-id

Additional slides on scripting

- hash

- hashall option

- hash -p ls

- hash -p ps

- hash -d ls

- hash -d ps

- hash -r

- shell also uses caching techniques

Additional slides on scripting

- ➢ echo 'PS1=''[\u \w]''' >> ~.bash_profile
- ➢ what does the above command do ???

Additional slides on scripting

- ➢ Positional parameters in scripts

- ➢ $1,.....,$n

- ➢ $#

- ➢ $0

- ➢ $@

- ➢ $*

- ➢ IFS and $*

Additional slides on scripting

➢ Positional parameters in functions

1

➢ $1,.....,$n

➢ $#

➢ $0

➢ $@

➢ $*

➢ IFS and $*

Additional slides on scripting

1

- ➢ shift built-in and positional parameters
- ➢ Try to use shift in your assignments
- ➢ Variables in a script outside functions
- ➢ Variables in a script inside functions
- ➢ Global vs local variables
- ➢ local var1 var2 in a function – function scope only

Additional slides on scripting

- echo $9
- echo ${10}
- echo ${UID}___${USER}
- echo -e ${PATH//:/\\n}
- echo -e ${PATH//:/'\n'}
- Try them !!!

Additional slides on scripting

- ➢ Write a shell script that will modify IFS and print each component of PATH variable one on each line
- ➢ Also print details such as permissions, hard-link count, uid,gid and dir name
- ➢ Do not use the method given in the previous slide

Additional slides on scripting

1

- ➢ declare -x
- ➢ declare -i
- ➢ declare -f
- ➢ declare -a
- ➢ declare -r
- ➢ declare -F

Additional slides on scripting

- You may use printf instead of echo
- printf "hello world!!"
- printf "%s %s\n" hello world
- printf "%10s\n" hello
- printf "%-10s\n" hello
- read var1 var2 .. varN

Additional slides on scripting

- ➢ Write a shell script that will kill all the background jobs currently running in your session

  hint: job -p
- ➢ signal handling in scripts
- ➢ trap "echo 'caught SIGINT'" INT //handler
- ➢ trap " INT   //ignore
- ➢ trap ' ' INT   //default

Additional slides on scripting

- ➢ eval and shell interpretation

- ➢ listpage="ls | more"

- ➢ $listpage

- ➢ listpage="ls|more"

- ➢ $listpage

- ➢ In both cases, try eval $listpage

- ➢ eval is a special shell built-in – refer to man bash

Additional slides on scripting

1

- Write a script that takes two command line parameters – file name and no. of lines to printed
- The script sorts the lines and prints the first N lines as per the second parameter

    Hint: use eval and head(another filter)
- head -10 filename
- cat filename | head -20

Additional slides on scripting

- ➤ Try echo $$#
- ➤ Try echo '$'$#
- ➤ Try echo $$var1
- ➤ Try echo '$'$var1
- ➤ Try echo \$$var1
- ➤ Try echo \$${var1}

Additional slides on scripting

1

➢ Write a script that takes on his command line a command and its arguments and starts its as a background job with the standard o/p and standard error redirected to files which are mentioned on the command line as well

Assignments in scripting

- ➤ write a simple shell script that will take one or more file names as parameters and list their various time-stamps

- ➤ write a simple shell script that will take one or more file names as parameters and list the type of the file

- ➤ modify your start-up shell script to add /home/corporate/ bin to PATH environment variable

Assignments in scripting..

➢ write a shell script that will take a file-name as a

parameter and list all its hard-links in the system

➢ write a shell script that will list all the login shells

in the system and their process ids

Note: the above 2 assignments can be done off-line;

they will give you a taste of shell-scripting which

is very different from high-level languages