Operating Systems

- User’s interface to the computer : ease and flexibility
- hides hardware and device peculiarities
- manages system’s resources : main memory, processor, devices
- permits concurrent usage improving utilization of resources
- does account keeping
- controls access to resources

Operating Systems

- provides high-level concepts (e.g., ‘file’) for storage of data and programs
- standards in OS : for portability of user applications(‘open’ systems)
- UNIX, Windows NT, and OS/2 widely used; many other ‘proprietary’ OS like IBM’s MVS
- manages system software such as compilers, assemblers, loaders
- supports utilities (like editors)

Automobiles vs OS

Bill Gates (Microsoft) :If GM had kept up with the technology like the computer industry, we would be driving $ 25 cars going 1000 miles / gallon

Welch (GM) : If GM had developed technology like Microsoft, we would be driving cars that would

- **crash twice a day**
- **only one person could use the car, unless you bought car95 or carNT but then you have to buy more seats**
- **…………...**

UNIX OS

- Initial development at Bell Labs in 1969 by Ken Thompson, Dennis Ritchie & others
- Parallel effort in Univ. of Berkeley
- Open Software Foundation (OSF) formed for its standardization. Current release is SVR4 (system V Release 4)
- many vendor versions : Solaris , AIX, SCO Unix
- public domain version Linux - redhat

UNIX OS

Why use UNIX OS ?

- **It runs on a wide variety of computers as compared to most other operating systems.**
- **It is the dominant OS on workstations and minicomputers. It is also used in supercomputers.**
- **It was designed with care and with a clear goal. It is elegant and modern.**
- **It is the first OS most part of which was written in C (high level language)**

UNIX OS

- started as an academic effort (in the univs & Labs) - the source code is available and readable.
- Linux, the public domain version of this OS, is very popular, freely available and being continuously enhanced.
- Most importantly, many crucial operating system design principles are well illustrated by it.
- designed as multi-user, multi-tasking system with a broad range of utility programs for editing, networking, etc.

Examples of Languages

- **Fortran Program**

```
C  Program which checks given 2 numbers
C  whether the second is a factor of first
        program  check_factor
         integer num, fact, rem
         print *, “ give first number : ”
         read *, num
         print *, “ give second number : ”
         read *, fact
         rem = num - fact * (num / fact)
         if ( rem .eq. 0 ) then
            print *, fact, “ is a factor of “, num
        else
         print *, fact, “ isn’t a factor of”, num
        endif
        end check_factor
```

Examples of Languages

- Assembly language program

```
call do_io
call do_io
call do_io
call do_io
mov  num, r0
idiv   r0, fact
imul  r0, fact
mov num, r1
sub   r1, r0
mov r1, rem
cmp $0, rem
jne  .L2
    …...
```

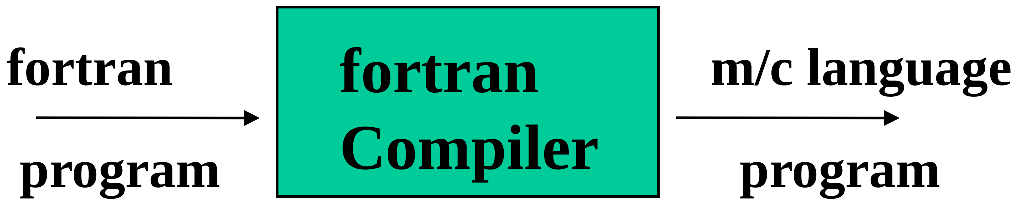
Examples of Languages

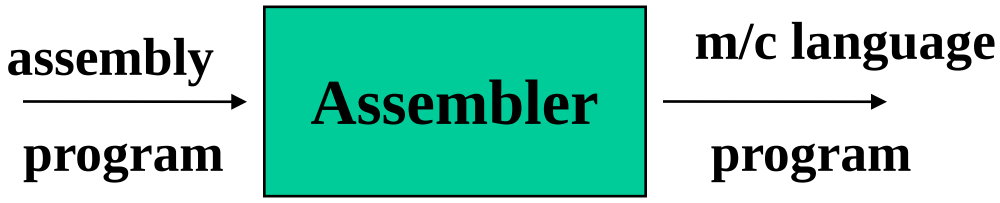
- Machine Language program

0001 1111	0011 1111	0100 0101	0100 1100
0100 0110	0000 0001	1010 1111	1000 0111
0101 1001	0100 1101	0000 1000	0101 1111

$\longleftrightarrow$

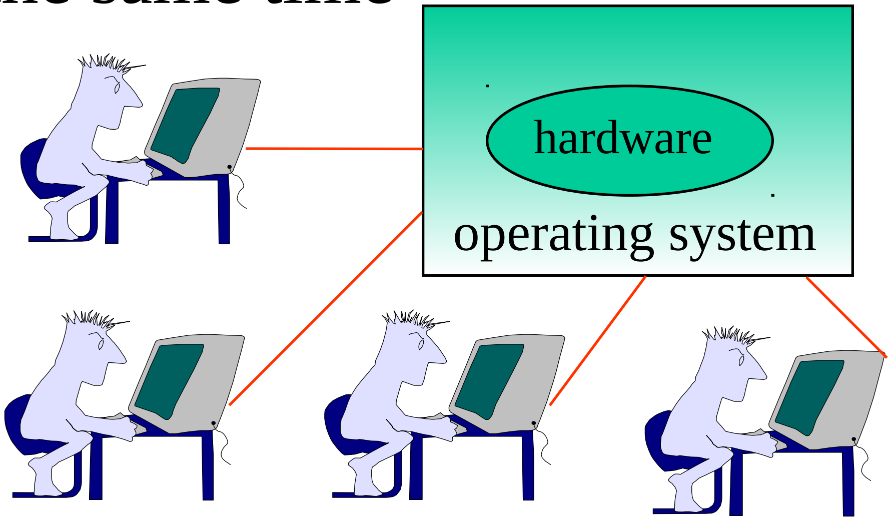
byte





Multi-user and Multi-tasking

- multiple users can login and work at the same time



os activities : maintaining accounts ;

scheduling the single cpu

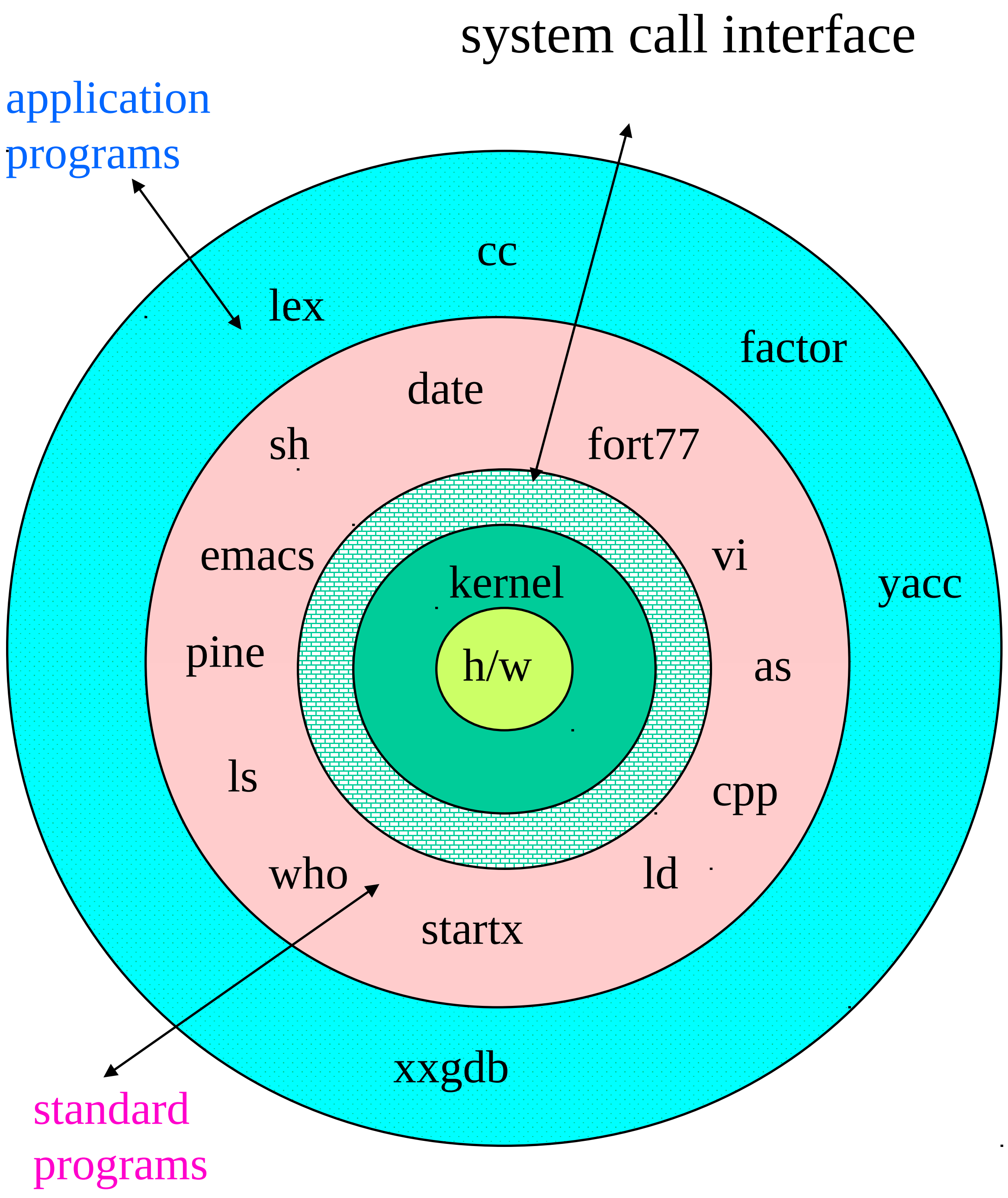
managing memory

managing disk space

protection of resources

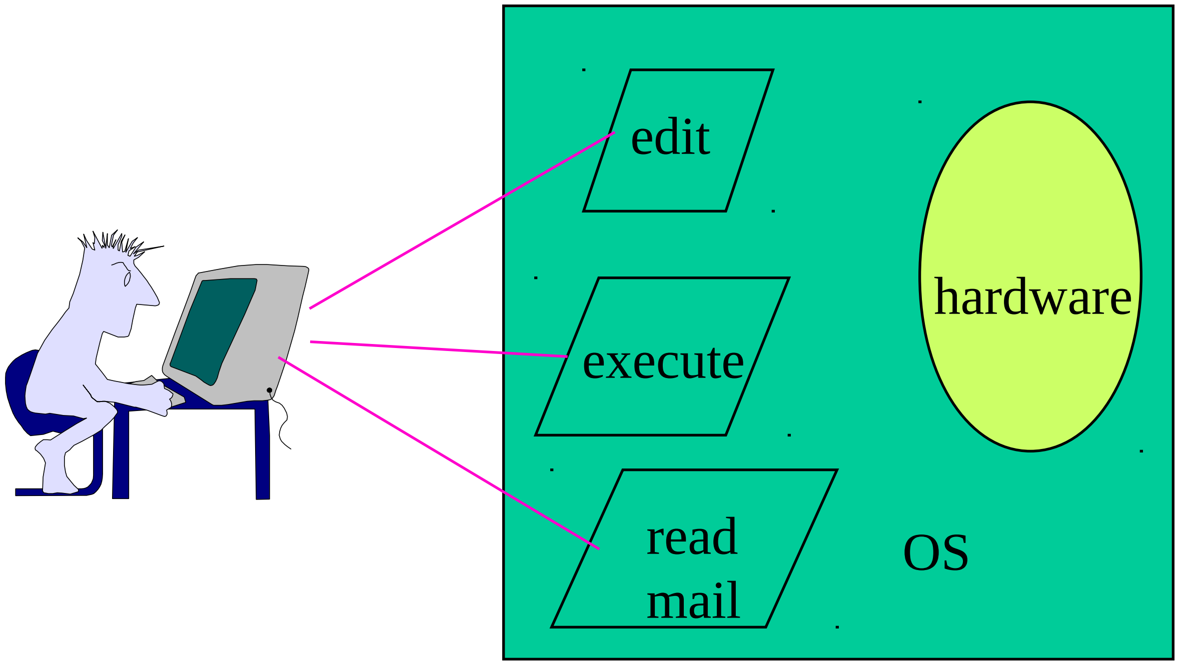
improving throughput

Architecture of Unix



Multi-user and Multi-tasking

- single user can initiate multiple tasks / processes at the same time



- program versus process :

program is a static entity ; resides in file

process is a program under execution - requires resources ; needs to interact with the OS

single program can be multiple process

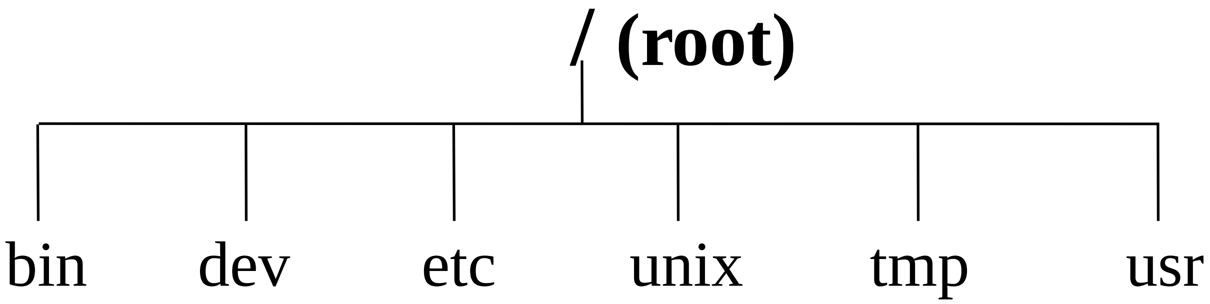
Unix

- The basic concepts supported by UNIX are files and processes.

UNIX File System is characterized by the following features

- a consistent treatment of file data, no file types
- a hierarchical organization of the file structure
- ability to perform basic operations on files
- permits dynamic growth of files
- treatment of peripheral devices as special files
- protection of file data

Pre-defined directories

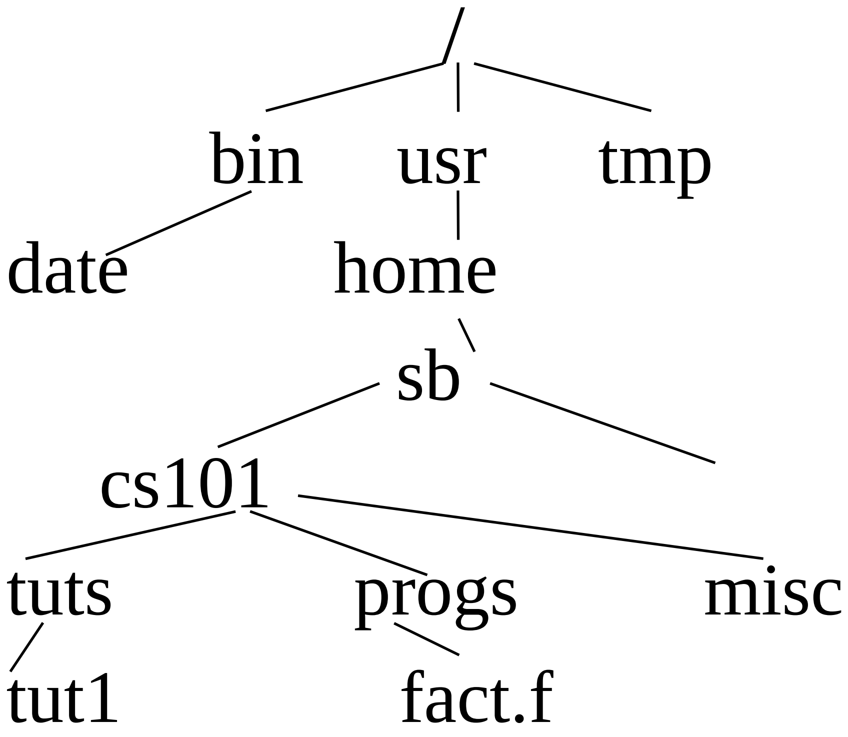


dev : device files (terminal,

printer, disk, …)
etc : data and script files
used by unix and users
usr : every user has a
directory in this place
tmp : for temporary files
bin : executable files (many
common commands)
unix : executable form of os

Directory Hierarchy

- unix file system is an inverted tree with / as the root
- all files reside in this tree; every node of the tree has an unique path starting from the root



- home directory : directory of an user after logging in
- current directory : the directory where an user is at this point
- relative path-name : a pathname relative to the current directory;

symbol **.** for current diry

. . for parent diry

- commands related to directories
  - create : **mkdir** - delete : **rmdir**
  - list : **ls** - change-dir : **cd**
  - current-dir : **pwd**
  - change-permissions : **chmod**

Unix File System

Attributes of a file :

- File /Group Owner : Indicates the individual owner, group owner ; Superuser has access rights to all
- File size : count in bytes
- File Type : Regular, directory, character or block special or fifo (pipes).
- File Access Permissions: There are three classes : owner,group owner and other users. Reasonable control over setting access rights.

Unix File System

- Each class has distinct access rights of the form r w x indicating read/write/execute which are set separately.
- For directories, execute implies right to search its contents.
- File Access Times : Time the file was last modified, it was last accessed and the time the inode was last modified.
- Link Count : This field indicates the number of names a file has in the directory hierarchy.

Session with Linux

- Logging into the system

You see the following on your screen -

```
login : sb <enter>
```

Enter your password

```
Password
```

```
No mail for you        (messages)
$                      (system prompt)
```

- user can start a session with unix now
- only users registered with the system can get through this stage
- file /etc/passwd contains relevant data for each user - encrypted password
- run a program - passwd - to change your password as often as you wish

Linux : directory commands

```
$ pwd
   /usr/home/sb
$ mkdir cs431
$ mkdir personal
$ ls
  cs431  personal
$ cd cs431
$pwd
/usr/home/sb/cs431
$ mkdir tuts
$ mkdir progs
$ mkdir misc
$ ls
  misc progs  tuts
$
```

```
$ cd tuts
$ vi tut1
$ cd ..
$ pwd
 ----------------
$ cd progs
$ vi fact.c
$ ls
 ----------------
$ ls ..
 ----------------
$ ls ../../..
 ----------------
$
```

Directory commands

- User starts a session from a fixed place in file system tree; home directory (starting current directory)
  - cd (change directory) permits one to go an arbitrary place in the tree
  - cd pathname
  - pathname starting with the root (/) is an absolute pathname
  - else it is taken as relative to the current directory
  - cd (missing pathname allows one to get back to home directory)
- pwd mkdir rmdir ls

  are the other common commands related to usage of directories

Directory permissions

- Can we go to any directory ?

Depends upon the access rights

ls directory_name : displays the contents of a directory in columns sorted in lexicographic order

ls -l directory_name : displays more information

```
$ ls -l  cs431
drwxr-xr-x  2 sb  fac Jan 29  15.00 tuts
drwxr-xr-x  2 sb  fac Jan29  15.00 progs
drwxr-xr-x  2 sb  fac Jan 29  15.00 misc
$
```

type perms links owner group size (in bytes) time-stamp name

Session with Linux

- **Default permissions with mkdir, vi etc.**
  - **for a directory : d rwx r-x r-x**
  - **for a file : - rw - r- - r- -**

□ □ □ □

type owner group others

  - **command chmod to change permissions**

chmod [ugoa][+ -=][rwx …][…] name

  - **set progs directory with rw- r-- r--**

```
$ chmod ugo-x progs
  ( current directory is cs431)
$ ls -l
$ cd  progs
   progs : permission denied
```

Session with Linux

- **Set progs directory perms to**

```
-wx --x --x
$ chmod a-r progs
$ ls  -l
$ ls -l progs     ( denies - no permission )
$ cd progs        ( what happens ? )
$ vi fact.c        ( the existence of fact.c ?)
```

- **Implication of write permissions**
  - **default perms of a created directory prevents others from inserting a file/diry in that place**
  - **os prevents its directories from being messed around by other users**
  - **only the owner can change the default access rights**

Directory contents

Summary : access rights of a directory

- **examining the contents (r);**
- **putting files/dirs in a given diry (w);**
- **going to a diry (x)**
- **owner sets and controls such accesses**

- **Contents of a directory :**
  - **2 special entries**
  - **1 entry for each entity (file / dir) contained in the directory**

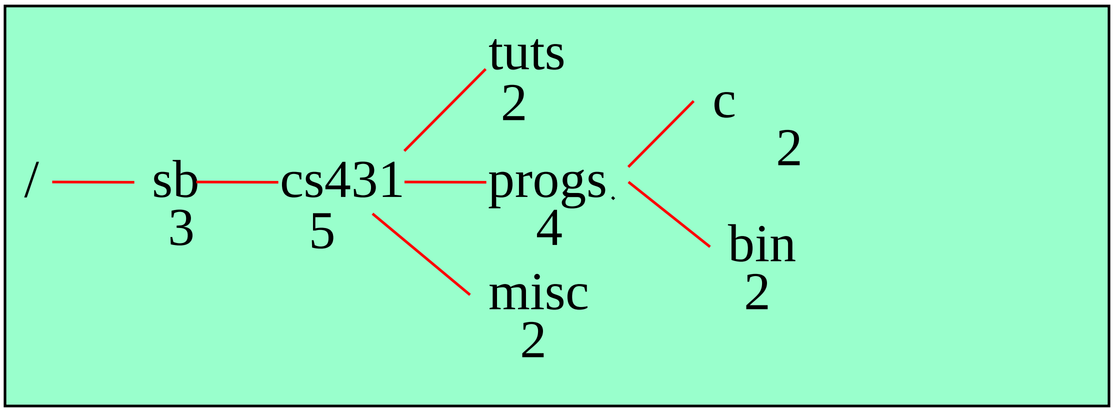
$ ls -al (displays the actual contents)

•	1243	This directory
••	234	Parent directory
tuts	391	
progs	45	
misc	675	

Name	inode-number
14 bytes	2 bytes

About Directories

- **Link Count of directories**



- **link-count = no. of child directories**

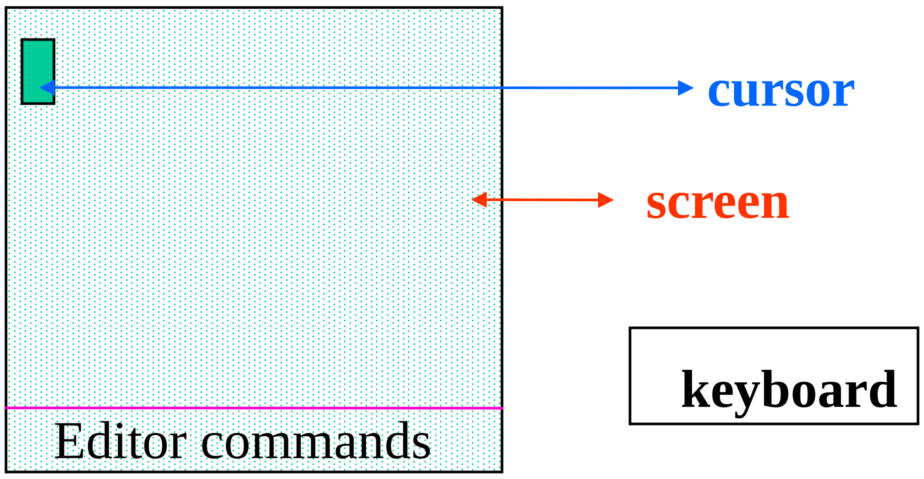
  + 1 (parent) + 1 (itself)

  what about root ? Same formula holds since the parent of root is root itself
- **a directory may be deleted only when all files and subdirectories contained in it are deleted; (link-count =2)**
- **file is deleted when its link-count drops to 0**

About Directories

- File commands
  - file creation : in several ways
  - using an editor to create / edit a file

    $ vi <filename>
- Editor views a file as a matrix of characters : notion of lines and characters / words within a line



File commands

- **common editor features are**
  - **insertion / deletion of text :**

    character, word and line level
  - **cut and paste text**
  - **search, replace text**
  - **do spell check**
  - **run os commands from within**
- **vi has 3 modes -**

  insert edit miscellaneous
- **file related commands are**

  cp (copy) rm (remove) mv (move)

  diff chmod
- **executable files : an executable file has its x bit(s) set**

Linux Commands

- all linux commands that we can execute after logging in are files that are executable

```
$ cd /bin
$ ls
    bash      date   man    rmdir
    cat       echo   mkdir  sh
    chmod     grep   pwd    sort
    cp        ls     rm     vi      …...
```

in the directory /usr/bin

```
   cal   fort77  passwd  pine  wc  who
```

/usr/X11R6/bin may contain

```
   startx   xman    xxgdb   …...
```

Linux Commands

- most of these files have perms -

  -rwx r-x r-x 1 root root size …..

  they are executable by all
- using some commands

```
$ date
$ cal 7 1998
$ man cal
$ which cal
$ cp /bin/ls .
$ ls -l
$ which ls
$ echo $PATH
```

Linux Shell

After logging in run ps

```
$ ps
```

A program (bash) is running without being invoked by us; this is the “shell” program

```
$ vi /etc/passwd
```

login-name:encrypted-password:uid:group-id:misc:login-dir:shell

- login program reads this file to authenticate the user identity and also starts the shell program

```
$ exit       ( terminates shell)
```

Linux Shell

- shell is the user interface of the OS

  this program interprets the commands entered by an user
- Features of Shell as a command interpreter
- shorthand for filenames

  certain characters are special

  *   matches 0 or more character
       in filenames

  [abc]   matches any single
  [a-c]   character from abc
             in filenames

Linux Shell

```
$ ls  /bin/s*
$ ls -l /usr/bin/[pr]*
```

– echo is a program that echoes its arguments

```
$ echo welcome to LT
```

– comparing ls and echo

```
$ ls /                    $ echo /
$ ls                      $ echo
$ ls *                    $ echo *
$ ls ‘*’                  $ echo ‘*’
```

role of shell is exposed

Linux Shell

- Grouping of commands

  several commands may be fired in the same line using semicolon as a separator between 2 commands

```
$ date ; who ; echo ; cal
```

  commands run in sequence, not simultaneously

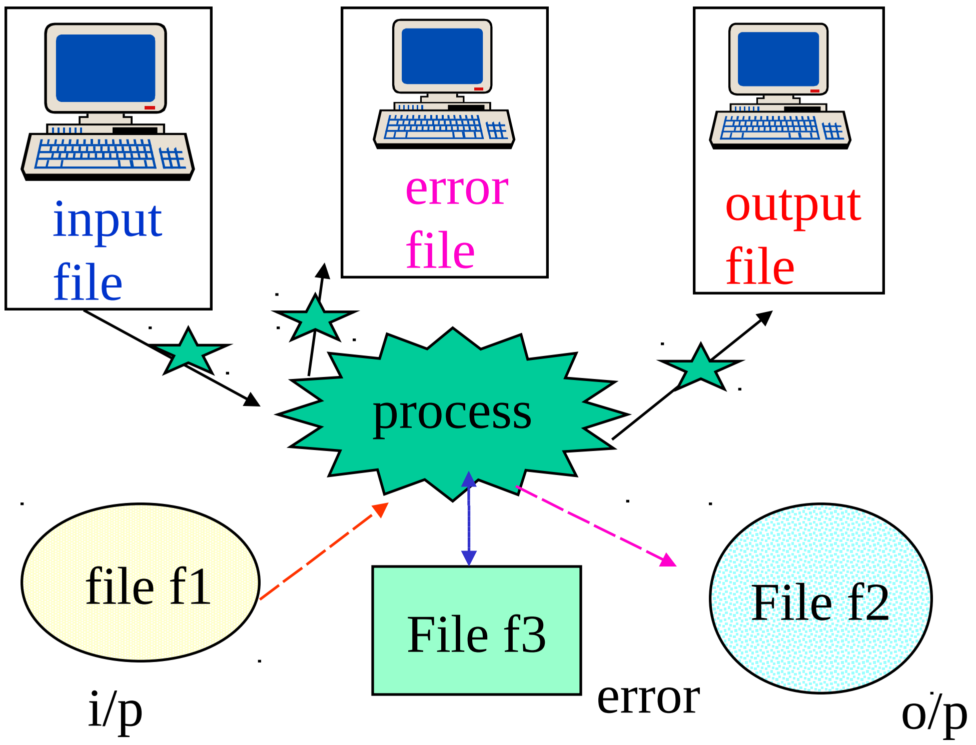
- Redirection of input / output

  every process has 3 associated files

  - 1 for receiving inputs
  - 1 for writing the outputs
  - 1 for indicating errors

  devices are also files in linux

Linux shell



f1, f2, f3 are regular files

- **shell uses the symbols**

<	**for input redirection**
>	**for output redirection**
>>	**for appending output**
2>	**for error redirection**

Linux shell

```
$ ls - al  > contents
$ ls  >> contents
$ wc - l <  contents
$ wc - l  <  contents > count
```

- a program may access several files
- only those programs which do all their i/o using the three standard files are amenable to such redirection
- when the o/p of one command is to be given as i/p to another, another feature is more handy

Linux shell

- connect commands using pipe
  - symbol '|' stands for pipe

    ```
    $ ls -al | more
    ```

    output of ls is passed down to more; more's o/p is on stdout
  - the commands connected through a pipe are active simultaneously
  - what happens if ls is slow and more is very fast ?

    ```
    $who | sort
    $ ls -al /bin | wc -l
    $ ls -al /bin | grep ^...s | wc -l
    ```

Linux Shell

- **Summary of useful commands**

sort -[rnf….] <input>

sorts its input by line I ASCII order;

f : fold upper/lower case

r : reverse normal order

n : in numeric order

d : ignore characters except letters,

digits and blanks

grep pattern filenames

searches filenames (or stdin) and prints each line that contains an instance of the pattern

The patterns are specified in regular expression notation; certain characters are given special meaning by grep

Linux shell

grep regular expressions

symbols	meaning
c	non-special character matches itself
\c	ignore special meaning of c
^	beginning of line
$	end of line
.	any single character
[…]	any of the characters in …; ranges like a-z are ok
r*	zero or more occurrences of r
r1r2	r1 followed by r2

wc -[l] <input>

counts lines, words and characters in the input

Linux shell

- **Using the shell**

```
</home/sb/shell> cal
 Calendar for August 1998
</home/sb/shell> which cal
     /bin/cal
</home/sb/shell> cp /bin/cal mycal
</home/sb/shell> ls -l mycal
</home/sb/shell> ls -l /bin/cal
```

note the difference in file attributes of the original file and its copy

```
</home/sb/shell> mycal
 command not found

</home/sb/shell>  /home/sb/shell/mycal
 Calendar for August 1998

</home/sb/shell>  ./mycal     ( works)
```

Linux shell

- PATH : shell searches for a command when a relative pathname is specified.
- This search is guided by the contents of a special variable PATH of the shell

```
 </home/sb/shell> echo $PATH
  /bin:/usr/bin:/usr/local/bin:/home/sb
```

The pathnames are separated by ‘:’ these are searched in the order listed. The command mycal is not found because the path /home/sb/shell is not listed in PATH

```
</home/sb/shell> PATH=$PATH:.
</home/sb/shell> echo $PATH
 </home/sb/shell> mycal
     calendar for August 1998
```

make change permanent by editing the file .bash_profile

Linux shell

- **Shell variables**

The shell has many predefined variables - PATH HOME PWD MAIL PS1 PS2 ….

A variable is set by assigning values to it : <variable>=…..

A variable's values is obtained by referring to it preceded by a '$' symbol

```
</home/sb/shell> echo $PWD
</home/sb/shell> echo $PS1
</home/sb/shell> echo $PS2
</home/sb/shell> echo $HOME
</home/sb/shell> PS1=\<$PWD\>
```

Try setting some of these variables to your choice and see the effect

Order of Execution

In order to understand the order in which the shell executes a list of commands, we start with a few relevant terms and their explanations

- **File system : it is a collection which has a certain structure ; similar to a set of sets - elements of the set may be sets themselves**
- **Commands : programs that perform various kind of actions -**

  - query the information in the file system such as ls ; test; du; quota; etc.

  - change the structure of the file system such as mkdir; rmdir ; cp; mv; chmod; rm;etc.

  - general purpose problem solving

Order of Execution

- **Syntax and Semantics of commands**

  The terms are used in their usual sense as in English. Syntax refers to the grammatical structure of a sentence. Semantics refers to the meaning of a sentence.
- **Example : The following sentence :**

  The grass ate the cow.

  is syntactically correct but not so semantically in its normal usage.
- **The syntax of commands is given by :**

  command_name options arguments

  A complete syntactic description of commands would also include the redirection symbols and associated file names.
- **ls -xdz /bin   is syntactically correct**

Order of Execution

- **This command is syntactically correct as far as the shell is concerned since it checks for the existence of options by looking for the character '-'.**
- **The option xdz is extracted by shell and passed to ls which on execution detects it as an invalid option and displays error message. Such an error is better classified as a semantic error**
- **Creating more commands**
- **In order to construct newer commands from the existing ones, shell gives us a few operators. These operators have the symbols '|' '&&' '||' ';' and '&'**
- **The operators along with their syntax and semantics are given below.**

Order of execution

Exit status : Every command on termination returns a value to the shell which is the called its “exit status”. Usually it returns 0 (for true) on successful execution and 1 (for false) otherwise. Terms in [ ] are optional.

1. Operator ; usage p1 ; [p2]

p1 and p2 (optional) are commands

After termination of command p1, shell

initiates execution of p2. Exit status of p1; p2 is the exit status of the last command.

2. Operator &   usage : p1 & [p2]

The shell initiates the execution of p1 and returns immediately, without waiting for its completion, to execute the next command. The process p1 is said to run in the background. For p1 & the shell returns a value 0.

Order of execution

- **Operator && usage : p1 && p2**

  The shell executes the command p1. If exit status of p1 is true then command p2 is also executed and the exit status of p1 && p2 is exit status of p2. If exit status of p1 is false then p2 is not executed and the exit status of p1 && p2 is the exit status of p1.

- **Operator | | usage : p1 | | p2**

  The shell executes the command p1. If exit status of p1 is false then command p2 is also executed and the exit status of p1 | | p2 is exit status of p2. If exit status of p1 is true then p2 is not executed and the exit status of p1 | | p2 is the exit status of p1.

  Note that the exit status for both is the exit status of the last command executed.

Order of execution

- **Operator | usage : p1 | p2**

  The commands p1 and p2 are both initiated for execution. The output produced by p1 is redirected to become the input for p2. The exit status of p1 | p2 is the exit status of the last command.

Finding Order of Execution :

Given a list of commands such as -

p1 *op* p2 *op* p3 *op* ……….. *op* pn *uop*

where *op* is any of the 5 operators and *uop* is '&' or ';', the problem is to find

i) the order in which the commands are executed and

ii) the operands used by the various operators.

Example : Given the list p1 && p2 | p3

which of the following is correct ?

Order of execution

- Option1 : (p1 && p2) | p3

  1. Execute p1 && p2

  2. Execute | with left operand p1 && p2 and with right operand p3

- Option2 : p1 && (p2 | p3)

  1. Execute p1 ; if false terminate

  2. If p1 true the execute p2 | p3

- The two options indicate the different possibilities for finding the operands of && and |
- Both the interpretations are correct but give different results. The list is said to be ambiguous in such cases.
- Programming languages specify properties of operators to remove such ambiguities.
- Parentheses can help in disambiguation

Order of Execution

- Shell specifies a precedence relation among operators for disambiguation.

( )	highest
|	to
&& ||	lowest in
; &	precedence order

  The operators && and || have the same precedence but lower as than that of |. The operators & and ; again have same precedence but are lower than all.

- The general rule used by the shell can be described as :

  1. Commands are executed in left to right execution order (the operators are picked up from the list as they appear in the list from left to right.

  2. The precedence relation is used to select the operands for an operator.

Order of Execution

- **We have seen that using parentheses, one can rewrite a list such that the operators are clearly grouped with their operands.**

  Computational Procedure

  The operators in the list are examined in the order they occur in a left to right scan of the list in each of the steps given below.

  Step 1 : For each occurrence of the highest precedence operator in a list, i.e., “|”, parenthesize the operator along with its operands, one on its left and the other on the right. A parenthesized entity is taken as a single command. The operands of all pipes in the list will be identified at the end of this step.

Order of Execution

Step 2 : Repeat step1 for the operators in the next higher precedence level, i.e. for the operators && and | |, in the left to right order.

Step 3 : Repeat step1 for the operators in the next higher precedence level, i.e. for the operators “&” and “;”.

The output of this procedure is to fully parenthesize a list.

- Example : Consider the following list

  p1 | | p2 | p3 | p4 && p5 & p6 | | p7

- In step1, first the pipe p1 | p2 is grouped

  p1 | | (p2 | p3) | p4 && p5 & p6 | | p7

  then the next pipe is parenthesized

  p1 | |((p2 | p3) | p4) && p5 & p6 | | p7

- In step2, the | | involving p1 on the left is parenthesized resulting in
- (p1 | |((p2 | p3) | p4)) && p5 & p6 | | p7

Order of Execution

(p1 | |((p2 | p3) | p4)) && p5 & p6 | | p7

Then && with p5 on its right is encountered in step 2 leading to

((p1 | |((p2 | p3) | p4)) && p5) & p6 | | p7

The | | operator with p6 on its left and p7 on its right is then handled in step 2.

((p1 | |((p2 | p3) | p4)) && p5) & (p6 | | p7)

- **In step 3, the only operator found is the unary “&” leading to**

(((p1 | |((p2 | p3) | p4)) && p5) &) (p6 | | p7)

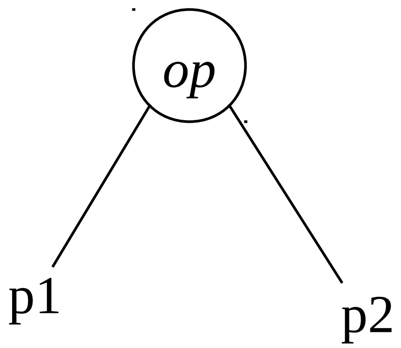
- **The procedure now terminates and the list has been fully parenthesized.**

Order of Execution

Interpretation from Parenthesized List :

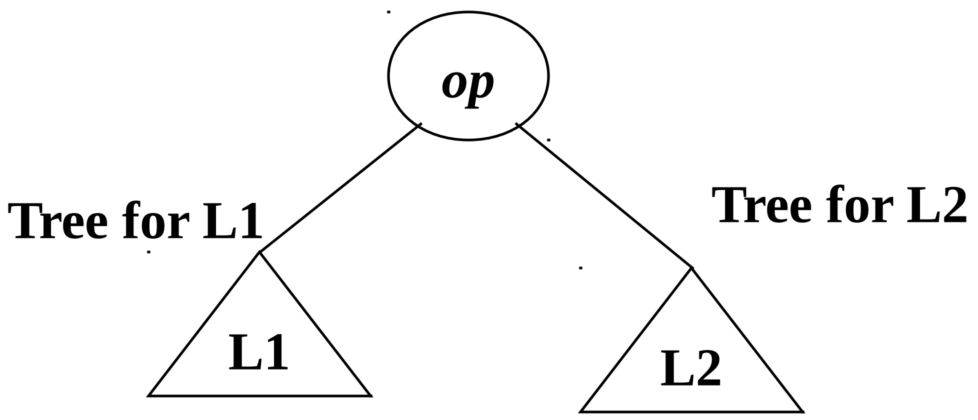
- **The parenthesized list obtained by using the computational procedure contains all the information concerning the execution order of the commands in the list.**
- **An alternative representation called a "parse tree" is more convenient for the purpose of interpretation.**
- **This construction is explained below:**

  Given a list of the form p1 *op* p2 , it can also be written in a tree form as :



Order of Execution

The tree rooted at *op* represents the list p1 *op* p2. In general given a list of the form : L1 *op* L2, where L1 and L2 are lists themselves, the corresponding tree would be as shown below :



- The Parse tree construction can be carried out each time a parenthesizing operation is performed in the steps of the procedure mentioned earlier.
- This is explained for the example given earlier.

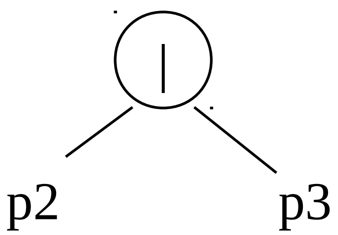
Order of Execution

- **Example :**

  p1 | | p2 | p3 | p4 && p5 & p6 | | p7
- **Step 1 : Parenthesizing first | gives**

  p1 | | (p2 | p3) | p4 && p5 & p6 | | p7

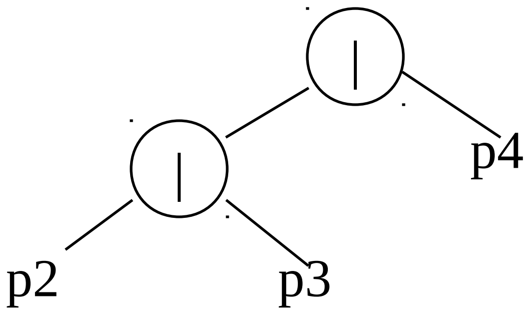
  Corresponding parse tree is



- **Step 1 : Parenthesizing second | gives**

  p1 | |((p2 | p3) | p4) && p5 & p6 | | p7

  Corresponding parse tree is

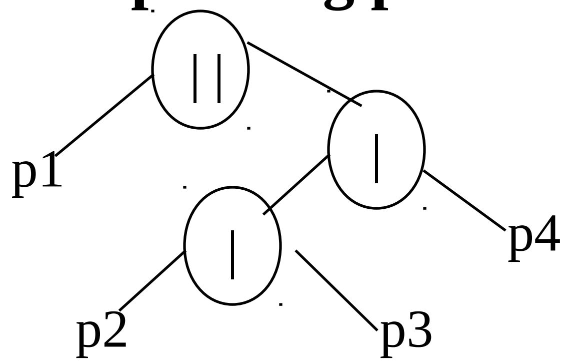


Order of Execution

- **Step 2 : Parenthesizing first | | gives**

  (p1 | | ((p2 | p3) | p4)) && p5 & p6 | | p7

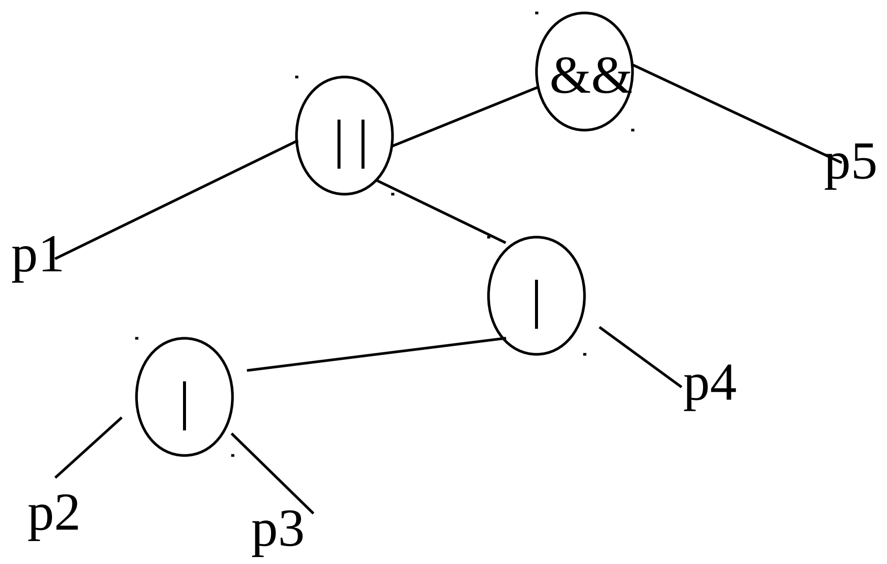
  Corresponding parse tree is



- **Parenthesizing && gives**

  ((p1 | | ((p2 | p3) | p4)) && p5) & p6 | | p7

  Corresponding parse tree is

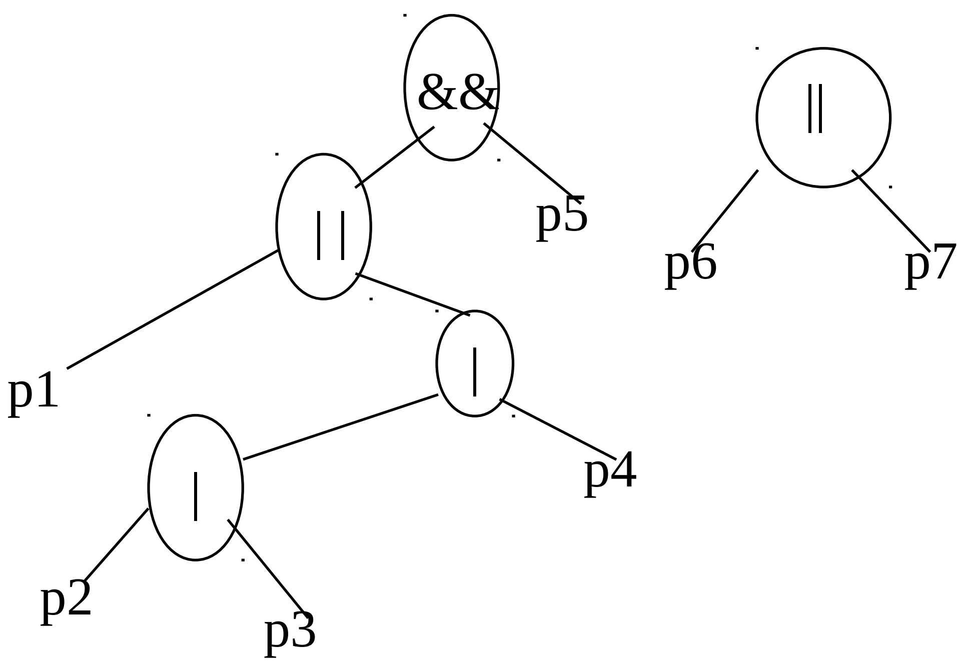


Order of Execution

- **Parenthesizing | | gives**

  ((p1 | | ((p2 | p3) | p4)) && p5) & (p6 | | p7)

  Corresponding parse tree is

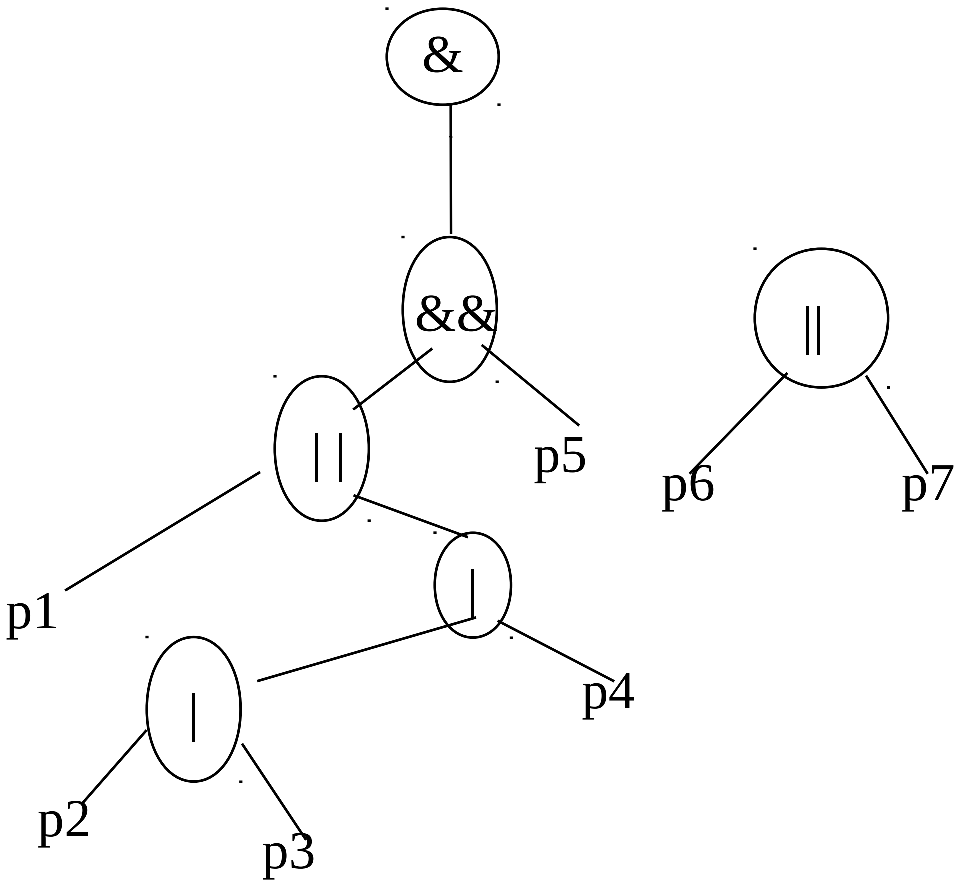


- **Finally parenthesizing & in step 3 gives**

  (((p1 | | ((p2 | p3) | p4)) && p5) &) (p6 | | p7)

Order of Execution

- **Corresponding parse trees are shown below. The tree rooted at & is processed by the shell before the tree rooted at ||.**



Order of Execution

The shell processes the parse tree starting from the root. The action taken depends upon the operator and its operands.

- If the operator is & then its only child, whether a single command or a list of commands (a subtree) is initiated for execution in the background. The child subtree of & is of course processed to determine in which order the commands are executed as explained below.
- For a binary operator, the shell examines the left child first and then the right child as dictated by the semantics of the operator.
- Example Parse Trees : The order of execution is determined as follows :

Order of Execution

1. The tree rooted at & is picked up and its child, the subtree rooted at && is executed in the background.
2. The shell picks up the other tree rooted at || and examines it.

   2.1 The command p6 is executed.

   2.2 If p6 returns 0, the execution of the

   subtree rooted at || is complete

   else the command p7 is executed

   and then the execution of || is over.

In the following we explain how the shell processes the subtree rooted at && which is under execution in the background.

Order of Execution

1.1 The && operator requires its left operand to be executed first. Since the left child is not a direct command, the shell proceeds to process the left subtree rooted at ||.

1.2 The || operator has command p1 as its left operand - so p1 is executed

1.2.1 If p1 returns 0, then the execution of subtree rooted at || is complete.

1.2.2 If p1 returns 1, then the right subtree of ||, which is rooted at | is processed.

- The left subtree of this tree is again a subtree rooted at |. The subtree rooted at | with operands p2 and p3 is now processed.

Order of Execution

- This causes the execution of p2 and p3 along with the required input-output connection between p2 and p3.
- The shell now proceeds to process the right operand of the subtree rooted at |, which is p4. This causes the execution of p4 and and the output of p3 to be redirected to the input of p4.

The subtree rooted at || is now completely processed.

1.3 The left operand of the subtree rooted at && has been processed. Depending on the exit status of the last executed command in 1.2 above, p5 is prepared for execution or ignored.

This completes the execution of the entire tree rooted at &&.

Order of Execution

- **It must be clear that the parse tree explicates the order in which the commands are picked up by the shell for execution.**
- **The parse tree lists only the various possibilities. The actual commands run during execution depends on the exit status of the commands involved and the operators connecting these.**
- **If the exit status of every command is known in advance then the commands that actually run and their order can also be determined.**
- **Example : Consider the same list**

  p1 | | p2 | p3 | p4 && p5 & p6 | | p7

  and assume that p1, p3, p5 and p7 return exit status 1 (false) while the rest, i.e., p2, p4 and p6 return status 0 (true).

Order of Execution

- **The commands that are executed along with their order is given below :**

  In the foreground :

  command p6

  In the background :

  command p1

  commands p2 , p3 and p4 (piped i/o)

  command p5

- **The commands in the background are executed in the sequence given; however they may run simultaneously with the command(s) in the foreground.**
- **The sequence shown above is unique under the given conditions because this is the only possible interpretation of the associated parse tree . This happened because the exit status of each command was made known in advance.**

Shell Programs

- **The power of Linux (or Unix) can be best appreciated by writing our own utilities using those supported by the OS**
- **Problem statement : list only sub-directories (not files) of a given directory**

```
$ ls -l > out
$ grep ‘^d’ out
$ wc -l out
```

- **These three commands when executed in the sequence given within a directory gives the listing of all its sub-directories only**
- **By storing this commands in a file, say , lsdirs, and executing the same we get the same effect as if this command was made available by the system.**

Shell Program

- Writing a shell program
  - a sequence of shell commands;
  - may define/refer to shell variables
  - can use control constructs of shell

```
$ cat lsdirs
```

```
# shell program to list sub-dir
 ls -l  | grep ‘^d’
echo -n “ No. of directories in $PWD =“
 ls -l | grep ‘^d’ |  wc -l
```

```
$ chmod +x lsdirs
$ lsdirs
```

- must cd to a dir and then use lsdirs
- why can’t specify diry pathname as an argument as we do for ls command ?

usage : lsdirs <diry_pathname>

Shell Program

- Before one writes a shell program, a proper design of the solution to be problem has to be worked out. One aspect of this is the user interface.
- User interface :

  - lsdirs invoked with no argument

    which directory should be listed ?

    /home/you   current directory

    /   any other

    Since system supported, ls, lists the current directory; we also decide to do the same.

  - lsdirs invoked with one argument

    the argument must be a directory path name : should it be absolute or relative ?

    Once again based on working of “ls”, we permit both pathnames for lsdirs

Shell Program

As a consequence the following forms for invoking lsdirs are all legal

```
lsdirs /bin              lsdirs  .
lsdirs                   lsdirs ../../home
lsdirs tuts              lsdirs /home/sb/cs101
```

assuming the directories exist.

- lsdirs with multiple arguments

we know that "ls" supports this feature; we have a choice : let us decide not to support lsdirs with >=2 arguments.

- **Designing a solution**

  Our solution should match the user interface worked out above :

  - lsdirs may be invoked with either no arguments or exactly one argument

  - lsdirs as it exists works fine with no arguments.

Shell Program

- It does not work fine when supplied an argument. The error is in the fact that the “ls -l” is invoked without any argument inside lsdirs.
- The argument passed to lsdirs at the time of invocation needs to be used while executing “ls -l” inside it.

Suppose lsdirs in invoked with

```
$ lsdirs /usr/home/sb
```

then we want the first command in lsdirs

to be : ls -l /usr/home/sb | grep ‘^d’

instead of ls -l | grep ‘^d’

- The basic problem is that how does a shell program know (at the time of writing the program) information about a) how many arguments and b) what arguments are supplied to this program at execution time ?

Shell Program

- The shell provides help to solve the problem stated above.

Shell processes a command and stores certain information about the command.

Consider the following command :

```
$ ls -l -t  dir1  dir2  dir3  dir4 dir5
```

The shell creates the following variables after processing the command line :

-t	-l	dir1	dir2	dir3	dir4	dir5	**$***
$1	**$2**	**$3**	**$4**	**$5**	**$6**	**$7**	

7	**$#**	**0/1**	**$?**	ls	**$0**

$* is a variable containing all the arguments as shown above.

Shell Program

- The individual arguments are stored in the variables $1, $2, …, $9; variable $0 holds the command name and variable $# stores the number of arguments passed to a program.
- These entities are called $-variables. The shell sets the $-variables while processing a command. An executing program may refer to these variables.
- The $ variables are also known as Positional Parameters. The first 9 positional parameters, may be used directly by referring to their names, $1 to $9. However 10 th parameter onwards are not referred by $10, $11….
- $# : count of parameters

  $* : list of all parameters

  $? : the exit status of last command

  $1 to $9 : First nine parameters

Shell Program

- **write and test the use of $ variables by writing a program “args”**

```
 $ cat args
# This programs prints its arguments   echo 'command name: $0’
 echo "$0"
 echo ' first argument: $1’ ; echo "$1”
 echo 'second argument: $2’ ; echo "$2”
 echo ' no. of arguments :’ ; echo “$#”
 echo ' list all arguments :’ ; echo “$*”
$ chmod +x args
$ args  -t  /usr/bin  123  /bin  -wsv
```

$0 shows the command name while the first two arguments are shown by $1 and $2 as -t and /usr/bin respectively.

$# is 5 and $* displays in order all the 5 arguments.

Shell Programs

- **We know a shell program can find out its name, parameters and use the same**
- **Rewrite the lsdirs program to work in the desired form**

```
# shell program to list sub-dir, version1
 ls -l  $1 | grep '^d'
echo -n " No. of directories in $1 ="
ls -l  $1 | grep '^d' | wc -l
```

- **This version of lsdirs works quite fine with different directory names (in absolute or relative form).**

Problems with this design :

1. It does not handle properly the case of more than 1 arguments; in fact when more than 1 are supplied, it displays the directories of the first one and stops (is silent about the other arguments).

Must indicate an error message

Shell program

2. **If the argument is not a directory, a error message, issued by “ls” is displayed; user of lsdirs may be confused since he/she never invoked ls directly.**

   lsdirs should check for such conditions and issue an appropriate error message;

Design of lsdirs with error messages :

- **To overcome problem 1, lsdirs must find out whether it was invoked with 2 or more arguments. $# can be used to determine the number of arguments.**
- **If $# is equal to 0 (no arguments) or**

  $# is equal to 1 (one argument), the commands should be executed
- **If $# is greater than or equal to 2, issue error message and stop.**

Shell Program

- **The shell provides two constructs for handling such conditional situations.**

  - if-then-else or if-then construct :

  This construct runs commands based on the exit status of a command; its syntax is:

```
if  command
then
# if command has exit status true
    commands
else
# if command has exit status false
    commands
 fi
```

then, else and fi are recognised only after a newline or semicolon.

Else part is optional (if-then construct)

Shell Program

- Command test and its use : this program is often used in the condition part of an if-construct.
- This program can test many things

  - status of a command:

    if test command

    returns true if command returns true else it returns false

  - files : tests charcteristics of files such as file type, size and protections : a few commonly used options are

```
test -r file   # file exists & readable
test -w file  # file exists & writable
test -f file   # file exists and not a diry
test -d file  # file exists & is a diry
test  -s file  # file exists and size > 0
```

Shell Program

- numeric values : two numbers can be compared using the following operators

  equal -eq   not equal -ne

  greater than -gt less than -lt

  greater than or equal to -ge

  less than or equal to   -le

example :

```
if  test  $# -eq 0
then
    commands  # executed if $# = 0
```

- string values

  strings may be compared using the following operators :

  equality =   inequality !=

  zero length -z *string*

  non-zero length -n *string*

Shell Program

Example :

```
if test -z “$1”
then
  commands # executed if  $1 is null
  …..
fi
```

The quotes surrounding the variable are needed since otherwise the shell will substitute null for $1 before executing test.

lsdirs with error messages :

The following are used :

a) S# for the number of arguments

b) if-construct and test command for checking the value of $#

c) exit command : this command causes the termination of a shell program; 1 or 0 may be returned if desired.

Shell Program

```
# lsdirs program: version 2 ( error
#  messages included )
    if  test $# -eq 0
    then # no arguments
       dir=$PWD
    else
      if  test $# -eq 1
      then # exactly 1 argument
        dir=$1
      else  # case when arguments are >= 2
          echo “ 2 or more arguments given”
          echo “ usage : lsdirs diry-name”
          exit 1
       fi  # end of inner if : all part of outer else
    fi     # end of outer if
# code for version1; $dir replacing $1
        ls -l  $dir | grep ‘^d’
        echo -n “ No. of directories in $dir =“
        ls -l  $dir | grep ‘^d’ | wc -l
        exit 0
```

Shell Program

- **Check that version 2 of lsdirs works correctly with any number of arguments.**
- **Now we turn our attention to problem 2, namely handling arguments that may not be directories.**

- Solution : The solution is not to run ls and other commands of lsdirs if the input argument is not a directory

we use test -d directory-name

and check the return value.

Return value = 0 indicates directory does exist and not otherwise.

An easy way to do above is to protect the commands of version 2 towards the end of this program by enclosing these in an if-construct as shown in the following.

Shell Program

```
# lsdirs program: version 3
    if  test $# -eq 0
    then # no arguments
       dir=$PWD
    else
      if  test $# -eq 1
      then # exactly 1 argument
        dir=$1
      else  # case when arguments are >= 2
          echo " 2 or more arguments given"
          echo " usage : lsdirs diry-name" ; exit 1
       fi  # end of inner if : all part of outer else
    fi     # end of outer if
   if test -d $dir  #  version2  enclosed in if-then-else
   then  # directory exists - use earlier code
          ls -l  $dir | grep '^d'
          echo -n " No. of directories in $dir ="
          ls -l  $dir | grep '^d' | wc -l ; exit 0
        else
          echo " wrong use : $dir not a diry" ; exit 2
     fi
```

Shell Program

REMARKS :

1. **The heart of the lsdirs program is only 3 lines of command**
2. **Improving on error messages and handling of improper usage increases the code size.**

CASE CONSTRUCT :

The shell another construct for situations where a selection among certain alternatives have to be made.

Example : In lsdirs, we had three different situations to handle, a) when $# is 0, b) $# is 1 and c) $# >= 2

We coded this in versions 2 and 3 using 2 if-then-else constructs, one nested inside the other. The same may also be rewritten using the case construct.

Shell Program

- **Case construct : Its syntax is**

```
case  word  in
 pattern) commands ; ;
pattern)  commands ; ;
  ….
esac
```

This construct compares *word* with the *patterns*, from top to bottom, and performs the commands with the first and only the first pattern that matches. After execution of the commands, the shell goes on to execute the command next to esac.

- **The last pattern may be written in such a way that it becomes the catch-all or default. If no patterns match, none of the associated commands are executed.**

Shell Program

- **This construct permits one to select one (or none) from several alternatives.**

Rewriting lsdirs using a case construct :

```
# lsdirs program: version 4 ( using case )
    case $#  in # only one of three cases are executed
    0) dir=$PWD ; ; # no arguments
   1)  dir=$1 ; ; # exactly 1 argument
   *)  echo " 2 or more arguments given"
        echo " usage : lsdirs diry-name"
        exit 1 ; ; # case when arguments are >= 2
    esac
   if test -d $dir  # same  if-then-else  as in version 3
   then  # directory exists - use earlier code
          ls -l  $dir | grep '^d'
          echo -n " No. of directories in $dir ="
          ls -l  $dir | grep '^d' | wc -l ; exit 0
        else
          echo " wrong use : $dir not a diry" ; exit 2
     fi
```

Shell Program

Review of lsdirs :

1. **It works fine with any number of arguments supplied to it.**
2. **It checks for existence of input directory and also for number of arguments and issues appropriate error message.**

Can we extend our program so that it works like “ls” for 2 or more arguments : it should list the directories of each argument supplied to it in the order given?

Solution : All we need to do is to repeat the entire code for $2, $3 and so on.

This could be achieved as follows :

a) For $# = 0 ; we do the listing and quit.

b) For $# > 0; we repeat the entire code for $1, $2, etc.

Shell Program

```
# lsdirs program: draft version 5 ( for $# >= 2 )
    case $#  in # only one of three cases are executed
    0) dir=$PWD ; ls -l  $dir | grep ‘^d’
          echo -n “ No. of directories in $dir =“
          ls -l  $dir | grep ‘^d’ | wc -l ; exit 0
          ; ; # no arguments, display and quit
   *)     ; ; # for > = 2  arguments, do nothing
   esac
   dir=$1
   if test -d $dir  # same  if-then-else  as in version 3
   then  # directory exists - use earlier code
          ls -l  $dir | grep ‘^d’
          echo -n “ No. of directories in $dir =“
          ls -l  $dir | grep ‘^d’ | wc -l ; exit 0
        else
          echo “ wrong use : $dir not a diry” ; exit 2
     fi
    dir=$2
           Same commands as in box above
```

Shell Program

```
dir=$3
```

Same commands as in box in earlier slide

```
dir=$4
```

Same commands as in box in earlier slide

```
dir=$5
```

Same commands as in box in earlier slide

```
dir=$6
```

Same commands as in box in earlier slide

```
dir=$7
```

Same commands as in box in earlier slide

```
dir=$8
```

Same commands as in box in earlier slide

```
dir=$9
```

Same commands as in box in earlier slide

Shell Program

- Comments on draft version 5 :

  - it is not very readable

  - it is fairly lengthy

  - it will not work for >= 10 arguments

  - there is obvious repetition of same code, can’t this be avoided

- Iterative Construct of Shell

  - The shell has a few iterative constructs that permit us to write iterative sequences elegantly.

  - for construct : syntax

```
for var in list of words
do
   commands        # loop body
 # body is executed once each for
 # successive elements of the list; may
 # refer to $var for value of variable var
done
```

Shell Program

- **Example :**

```
for i in  *  # * is expanded by shell
do
  echo $i    # echoes each file/dir name
done
for i in $*  # expands to all arguments
do
  echo $i     # all arguments are listed
done
```

- **In the first loop, variable i is set to each file/dir name in the current directory in turn and the name can be accessed by referring to it as $i.**
- **In the second loop, variable i is set to each positional parameter in turn and this parameter is accessed by referring to it as $i.**

Shell Program

```
# lsdirs program: final  version 5 ( for $# >= 2 )
    case $#  in # only one of three cases are executed
    0) dir=$PWD ; ls -l  $dir | grep ‘^d’
          echo -n “ No. of directories in $dir =“
          ls -l  $dir | grep ‘^d’ | wc -l ; exit 0
          ; ; # no arguments, display and quit
   *)    ; ; # for > = 2  arguments, do nothing
   esac
   for  i  in  $*  # i  is set to each parameter
   do
     dir=$i
    if test -d $dir  # same  if-then-else  as in version 3
    then  # directory exists - use earlier code
          ls -l  $dir | grep ‘^d’
          echo -n “ No. of directories in $dir =“
          ls -l  $dir | grep ‘^d’ | wc -l
        else
          echo “ wrong use : $dir not a diry” ; exit 2
     fi
    done
```

Shell Program

- **Remarks on version 5 of lsdirs :**

1. It is fairly general now; can handle varying number of arguments

2. It stops as soon as it detects an argument that is not a directory.

It is fairly easy to extend lsdirs so that it can continue even after it detects an invalid argument

- displays a message “ continuing processing” and checks the remaining arguments

- displays a message :

do you wish to continue (y/n) :

If the user enters “y”, then it continues else it quits. This is achieved by

```
echo -n “ wish to continue ? ( enter y/n) :”
read response  #  $response holds user i/p
```

Shell Program

Problem : How to undelete a file ?

- a file deleted using “rm” is lost for ever

- to support undelete we have to design our own program

Solution :

- write a program delete for deleting files

- write a program undelete for undoing the effect of delete

- since both these programs would be written by us, delete must do some actions that helps undelete in recovering

- no file can be recovered if it is genuinely returned to OS

Syntax and Semantics of delete :

- **syntax is delete <filename>**
- **semantics : first perform routine error check on the filename (single argument)**

Shell Program

File to be deleted is saved in a special directory, say, back_up

Then file is actually deleted from its current location.

While storing a file on the back_up directory, checks for the existence of this directory, existence of the file etc., need to be performed.

Syntax and Semantics of undelete :

- syntax is undelete <filename>
- semantics : first perform routine error check on the filename (single argument) for existence

  Check whether this file exits in the back_up directory (exist only if erased using our delete program)

Shell Program

- # delete program : this program first copies the # file to be deleted in a fixed place in back-up

```
 # directory before deleting it.
case $# in
  0)  echo “ error : no file ; quitting” ; exit 1 ;;
  1)  file=$1
   if test ! -f $file
   then
     echo “ error : file $file does not exist/not a file”
     exit 2    # error indication
    fi  ;;
  *)  echo “ error : give only 1 arg” ; exit 2 ;;
esac
  if  test ! -d "$HOME/back_up”
 then
     mkdir "$HOME/back_up”
 fi
#  if  file exists in back_up : decide your action ?
     cp $1 $HOME/back_up ;  rm $1 ; exit 0
```

Shell Program

```
#  undelete program : to be used to
# undelete a file (not dir) provided the
# file was  deleted using  delete program
# perform error checks on arguments as
 # done in case of delete : not shown here
dir=$PWD
if test -f  $HOME/back-up/$1
then
  mv $HOME/back-up/$1  $dir/$1
  rm -f  $HOME/back-up/$1
else
echo "Sorry, this file can't be undeleted”
exit 1
fi ; exit 0
```

# delete and undelete versions assume $1 to be a

# filename; how to make it accept a pathname ?

More Features of Shell

Shell supports many more features which are easily learned by practice. A few more features are introduced in the following :

- **More Looping Constructs**
  - **while-do loop : its syntax is**

```
while command
do
 commands     # body of the loop
done
```

The body of the loop is executed so long as the command returns status true. Every iteration of the loop comprises i) command execution followed by ii) if status true then body else quit from the loop.

This iterative construct is useful when the number of times to iterate is not known in advance.

More Features of Shell

Example 1: A file, named marks_data, has the following data : rollno; marks (out of 10) in the format as shown below:

```
98001001        9
…………        .
```

One line for every student. Problem is to list the students who secured full marks.

Solution : Each line from the file is picked up and examined one by one.

```
cat  marks_data |
while  read rollno  marks
do
 if test $marks -eq 10
 then
    echo “ $rollno   $marks “
 fi
 done
```

More Features of Shell

Explanation : 1. Cat supplies the file contents to the while construct through |

2. While uses read command and its status to iterate through its body.

3. Read command reads one line at a time, separates strings using blanks and assigns a string to a variable, in the order listed. The first read assigns the string 98001001 to variable rollno and string 9 to marks and returns true so that the body is executed.

4. The subsequent reads get another line from the file and assign appropriate strings to these variables. When eof (end of file) is reached, read returns false which causes the termination of the loop.

More Features of Shell

Example 2 : create a file that contains the numbers 1 to 50, a number per line

```
count=1    # set a variable to 1
while test  $count -le 50
do
   echo “ $count” >> numbers
   count=`expr $count + 1`
done
```

- **The command test compares the value of count with 50 and returns true so long as 1 <= count <= 50.**
- **In the body, a value is first written.**
- **The expr command performs the operation (+ here) on the operands, provided these can be interpreted as numbers. The purpose is to increment the existing value of count by 1.**

More Features of Shell

Example 3 : Program to read the passwd file, extract different pieces of data and output them in a pretty form (1 per line).

Solution Strategy : The file /etc/passwd has one line per registered user:

loginid:encrypted_password:uid:gid:misc: home_directory:shell

- uid and gid are numbers assigned by the system and are known as user-identifier and group-identifier respectively.

- Misc holds general information such as full name, etc.

- **issue is to read a line from this file, store in seven different variables the parts of this line that are separated by colons.**

More Features of Shell

```
cat /etc/passwd |
while read loginid crypt uid  gid  misc
                   home shell
do
   ……
 done
```

The code segment given above in general will read one line at a time from the file /etc/passwd. But the intention of assigning the seven variables to the respective parts does not work. Why ?

- **Consider the following two lines that are for the users sb and sbb on parvati's /etc/passwd file (real data)**

```
sb:3j07KngVNPw2o:763:100:S

sbb:0NU80swQj1HmY:1728:100:Sandeep
34,
```

H3:/home/btech/sbb:/bin/sh
B

More Features of Shell

sb:3j07KngVNPw2o:763:100:S

Consider the line for user sb; the

above indicates a blank character.

The read command on reading this line

results in assigning the variables to :

loginid=sb:3j07KngVNPw2o:763:100:S

crypt=Biswas:/home/users/sb:/bin/sh

remaining 5 variables are assigned nothing

(null string).

- **The processing of line for user sbb :**

sbb:0NU80swQj1HmY:1728:100:Sandeep

34,

results in the following asignments :

loginid=sbb:0NU80swQj1HmY:1728:100:Sandeep

crypt=B uid=Bhatkar, gid=94005034,

misc=H-3:/home/btech/sbb:/bin/sh

home and shell are assigned null string

Summary : read uses blanks for separation

H3:/home/btech/sbb:/bin/sh

- **B**

More Features of Shell

Solution to pretty printing of password-file

- **the strategy of reading using a while-loop if fine**
- **if the separator should be changed from**

blank to colon; code will work fine

Shell Features : Shell has a predefined variable IFS (internal field separator), which is used by read command. We can set this variable to another character of our choice. Read then would automatically use user-defined string as the separator.

```
IFS=“:”
cat /etc/passwd |
while read loginid crypt uid  gid  misc
                home shell
do
  ……   #  same body including done
```

More Features of Shell

Usually the solution above would work fine except for the following situation :

- **When a field is absent, two consecutive colons may be seen; read command treats consecutive occurrences of the separator (like two or more blanks) as a single occurrence**
- **A missing field therefore may cause wrong values to be assigned to the variables . To overcome this, we separate consecutive occurrences of ":" by a blank and then pass a line to read**
- **stream editor, sed, is another linux filter program that is handy for this purpose.**
- **sed 'list of commands' filenames**
  is the general syntax; sed reads lines one at a time from input files, applies the commands from the list and writes its edited form on standard output.

More Features of Shell

Read the manual and find out the various features supported by sed.
We are going to use the following feature of sed :   sed “s/old/new/g” filename
old and new are patterns that are specified the s stands for substitute and g for global.
sed finds all occurrences (because of g) of old in filename and replaces these by new

```
        IFS=“:”
        cat /etc/passwd |
        sed “s/$IFS$IFS/$IFS $IFS/g” |
 while read loginid crypt uid  gid  misc home shell
     do
        echo “ loginname is $loginid”
        echo “ encrypted password is $crypt”
        ………….# add other echo commands
       echo “ shell is $shell” ; echo “---------------------”
      done
```

More Features of Shell

- **Another looping construct : until-do**

  Its syntax is :

```
until command
do
  commands
done
```

- **The command is executed first and if the status is false the body is executed. After every iteration the command is executed and the body is executed so long as the command returns false. The loop is exited when the command executes to true.**
- **Do-while and until-do are similar. Both constructs are used when the number of iterations are not known in advance. The condition decides when to stop iteration.**

More Features of Shell

Example : create a file that contains the numbers 50 to 1, a number per line

```
count=50    # set a variable to 50
until test  $count -lt 1
do
   echo " $count" >> numbers
   count=`expr $count - 1`
done
```

We want to generate the numbers in the sequence from 50 to 1; the test ensures that the until condition is false for all values of count, 50 >= count >= 1, inclusive of 1. The test is true only when count is 0 and then the loop exits, as is our requirement.

Expr also supports multiplication (*), division (/) and remainder (%); the only constraint is that the operands be integers only (not decimal arithmetic).

Shell Features : A summary

Shell as a programming language : review

- **supports variables - predefined by shell**
  - also user defined
- **structured language constructs**

  - sequencing -selection -iteration

  case if for while until
- **predefined programs (or commands) ;**

  user may add to the collection
- **Other features :**

  - parametrization : user can write shell programs using shell variables, enabling passing of arguments to the program

  - execute programs either sequentially or in parallel

  - constructing list of commands using the operators supported by shell; redirection of i/o; creation of files, etc.