LINUX PROGRAMMING

Course code:ACS010

III. B. Tech II semester

Regulation: IARE R-16

BY

Mrs. K Radhika

Assistant Professors

Mrs. G Sulakshana, Mrs. N.M Deepika, Mr. P Anjaiah

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

INSTITUTE OF AERONAUTICAL ENGINEERING

(Autonomous)

DUNDIGAL, HYDERABAD - 500 043

CO's	Course outcomes
CO 1	Understand the basic commands of Linux operating system and Demonstrate Sed and awk scripting
CO 2	Demonstrate shell scripts and understand creation of file systems and directories and operate them
CO 3	Synthesis creation of background and fore ground processes management through system calls and Generalize signal functions to handle interrupts by using system calls.
CO 4	Demonstrate Inter process communication using shared memory segments, pipes ,message queues
CO 5	Demonstrate various client server applications using TCP or UDP protocols.

LINUX UTILITIES

CLOs	Course Learning Outcome
CLO 1	Learn the importance of Linux architecture along with features.
CLO 2	Identify and use Linux utilities to create and manage simple file processing operations
CLO 3	Apply the security features on file access permissions by restricting the ownership using advance Linux commands.
CLO 4	Implement the SED Scripts, operation, addresses, and commands.
CLO 5	Implement the GREP and AWK commands for pattern matching and mathematical functions.

LINUX Features

- A computer operating system. It is designed to be used by many people at the same time (multi-user). Runs on a variety of processors.
- It provides a number of facilities:
- management of hardware resources
- directory and file system
- loading / execution / suspension of programs

LINUX File System

- **File:** is a container for storing information. A file is of 3 types.
- **Ordinary file**: It contains data as a stream of characters. It is of 2 types.
- Text file: contains printable characters.
- Binary file: contains both printable & non printable characters.
- **Directory file:** contains no data but it maintains some details of the files & subdirectories that it contains.
- Every directory entry contains 2 components:

  1.file name.

  2.Aunique identification number for the file or directory.

LINUX File System

- /Bin: contains executable files for most of the unix commands.
- /Dev: contain files that control various input &output devices.
- /Lib: contains all the library functions in binary form.
- /Usr: contains several directories each associated with a particular user.
- /Tmp: contain the temporary files created by unix or by any user.
- /Etc: contains configuration files of the system.

Modes of operation

- Vi has 3 mode of operation.
- **Command mode:** In this mode all the keys pressed by the user areinterpreted as commands. It may perform some actions like move cursor,save, delete text, quit vi, etc.
- **Input/Insert mode:** used for inserting text.

  – start by typing i; finish with ESC
- **Ex mode or last line mode**:
- Used for giving commands at command line.
- The bottom line of vi is called the command line.

Basic Cursor Movements

- H:move cursor one place to left
- down one
- up one
- right one
- W:move forward one
- b:word back one word

Basic Cursor Movements

- **Inserting Text**
- Move to insertion point Switch to input mode:i
- Start typing; BACKSPACE or DELETE
- for deletion
- ESCfinish; back in command mode

Basic Cursor Movements

Deletion

Must be in command mode.

- X Delete character that cursor is on.
- Dd Delete current line.
- D Delete from cursor position to end of line
- U Undo last command

Layered Architecture:

Linux System Architecture is consists of following layers

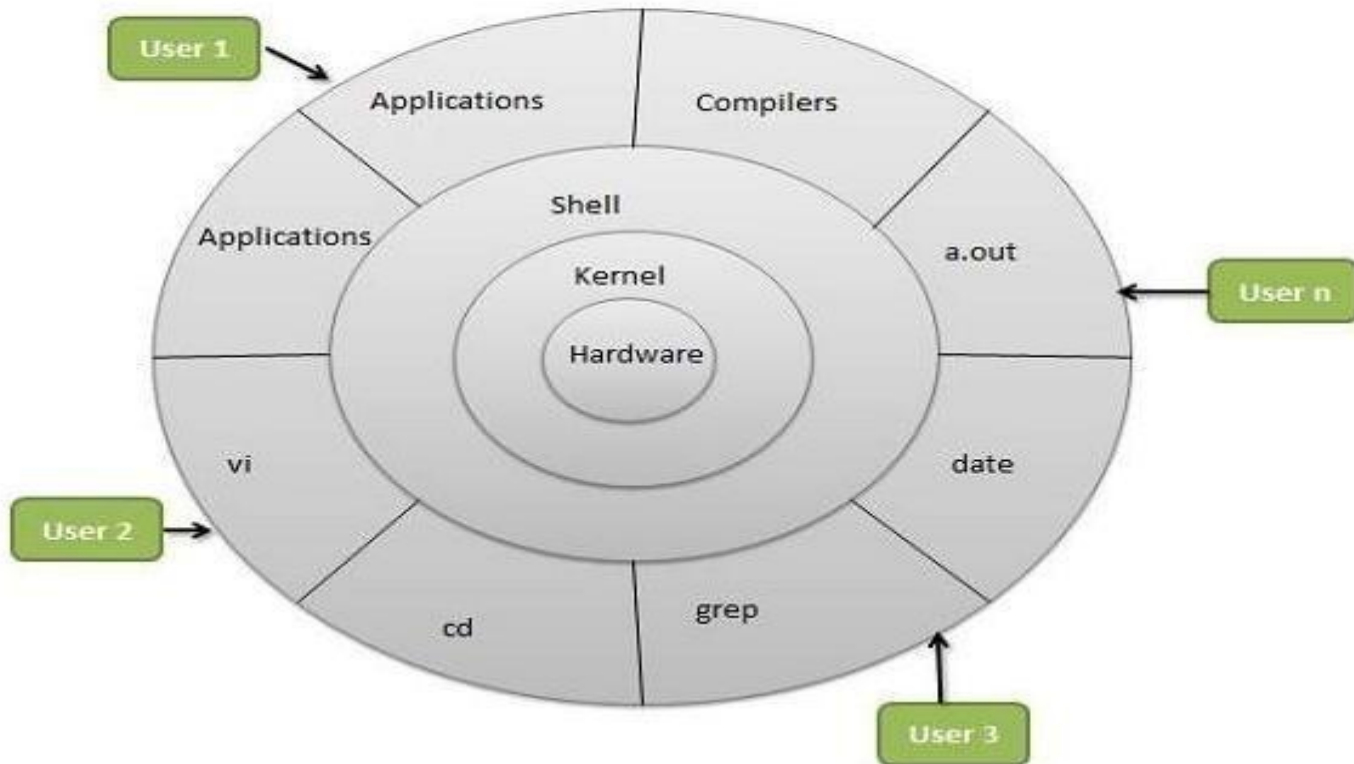
1.Hardware layer - Hardware consists of all peripheral devices (RAM/ HDD/ CPU etc).

2.Kernel - Core component of Operating System, interacts directly with hardware, provides low level services to upper layer components.

3.Shell - An interface to kernel, hiding complexity of kernel's functions from users. Takes commands from user and executes kernel's functions.

4.Utilities - Utility programs giving user most of the functionalities of an operating systems.

Layered Architecture



File handling utilities

- Cp (Copying Files)
- – To create an exact copy of a file you can use the —cp command. The format of this command is:
- cp [-option] source destination

Eg:

- Cp file1 file2
- Here file1 is copied to file2.

Eg:

- Cp file1 file2 dir
- File1 file2 are copied to dir.

Security by file permissions

- Unix follows a 3-tiered file protection system.
- Each group represents a category. There are 3 categories- owner ,group ,others
- Each category contains read ,write,execute permissions .
- rwx->presence of all permissions. r-x->absence of write permission
- r-- -> absence of write ,execute permission
- Chmod: changing file permission
- chmod sets a file's permissions (read, write and execute) for all three categories
- of users (owner, group and others).

Process utilities

Ps (process status):
Display some process attributes.
$ps

PID TTY TIME	CMD
1078 pts/2 0:00	bash

- Ps presents a snapshot of the process table.
- Ps with –f option displays a fuller listing that includes the PPID.
- Ps with –u option followed by user-id displays the processes owned by the user-id.

Ps with –e option displays the system processes.
Who: know the users
Displays the users currently logged in the system.

Disk utilities

Du: disk usage Du estimate the file space usage on the disk.
It produces a list containing the usage of each subdirectory of its argument and finally produces a summary.
$du /home/usr1

Df: displays the amount of free space available on the disk. The output displays for each file system separately.

$df
Mount:
Used to mount the file systems.
Takes 2 arguments-device name,mount point.

Networking commands

ftp: file transfer protocol

ftp is used to transfer files. It can be used with host name.

$ftp Saturn Connected to Saturn 220 Saturn ftp server

Name (Saturn: summit): Henry

Password: ******

To quit ftp use close and then bye or quit. ftp>close

221 good bye ftp>bye

Transferring files: Files can be of 2 types.

Uploading(put & mput):

To upload ur web pages & graphic files to website.

The put command sends a single file to the remote machine.

ftp>binary

200type set to I ftp>put penguin. Gif

Text processing utilities

cat: cat is used to create the files.
$cat> filename Type some text here Press ctrl+d
$Cat can also be used to display the contents Of a file.
$cat filename
Cat can also concatenate the contents of 2 files and store them in third file.
Cat>file1 file2>new file

To append the contents of two files into another file use
Cat>file1 file2>>new file

tail: tail command displays the end of the file.
It displays the last ten lines by default.
$tail file
To display last 3 lines use
$tail –n 3 file

Text processing utilities

head:

head command as the name implies, displays the top of the file.

When used without an option, it displays the first 10 lines of the file.

$head file

We can use –n option to specify a line count and display, say first 3 lines of the file.

$head –n 3 file   or $head -3 file

Sort: Sort can be used for sorting the contents of a file.

$sort shortlist

Sorting starts with the first character of each line and proceeds to the next character only when the characters in two lines are identical.

Sort options:

With –t option sorts a file based on the fields.

Text processing utilities

$sort –t —| +2 shortlist
The sort order can be reversed with –r option.
Sorting on secondary key:

U can sort on more than one field i.e.u can provide a secondary key to sort.
If the primary key is the third field and the secondary key the second field, we can use

$sort –t \| +2 -3 +1 shortlist
Numeric sort (-n):

To sort on number field use sort with –n option.
$sort –t: +2 -3 –n
group1 Removing duplicate lines (-u):The –u option u purge duplicate lines from a file.

Text processing utilities

nl:nl is used for numbering lines of a file.
Nl numbers only logical lines –those containing something other apart from
the new line character.
$nl file
nl uses a tab as a default delimiter, but we can change it with –s option.
$nl –s: file
nl won't number a line if it contains nothing.

Grep: globally search for a regular expression and print.
Grep scans a file for the occurrence of a pattern and depending on the options used, displays Lines containing the selected pattern.
Lines not containing the selected pattern (- v).
Line numbers where pattern occurs (-n)
No. of lines containing the pattern (-c)

Text processing utilities

File names where pattern occurs (-l) Syntax:

grep option pattern filename(s)

Egrep: extended grep

Egrepextendedsetincludes2 special characters + and ?.

--matches one or more occurrences of the pervious character.

?-- matches zero or more occurrences of the pervious character.

fgrep: fast grep

If search criteria requires only sequence expressions, fgrep is the best utility.

Fgrep supports only string patterns, no regular expressions.

To extract all the lines that contain an apostrophe use fgrep as follows:

$fgrep —‘ file

Text processing utilities

Cut: slitting the file vertically U can slice a file vertically with cut command.
Cutting columns(-c):
Cut with –c option cuts the columns.
To extract first 4 columns of the group file :
$cut –c 1-4 group1
The specification –c 1-4 cuts columns 1 to 4. Cutting fields:
To cut 1st and 3rd fields
use $cut –d: -f1,3 group1

Paste: pasting files
What u cut with the cut can be pasted back with paste command- but vertically rather than horizontally. u can view two files side by side by pasting them.
To join two files calc.lst and result.lst use
$paste –d= calc.lst result.lst

Text processing utilities

Join:

is a command in Unix-like operating systems that merges the lines of two sorted text files based on the presence of a common field.
The join command takes as input two text files and a number of options. If no command-line argument is given, this command looks for a pair of lines from the two files having the same first field (a sequence of characters that are different from space), and outputs a line composed of the first field followed by the rest of the two lines.
$join file1 file2

tee:

Unix tee command breaks up its input into two components; one component is saved in a file, and other is connected to the standard output.Tee doesn't perform any filtering action on its input; it gives exactly what it takes.tee can be placed any where in a pipeline.

Text processing utilities

Flags

-a Adds the output to the end of File instead of writing over it. **-i** Ignores interrupts.

Comm:

Suppose if u have 2 list of people, u are asked to find out the names available in one and not the other or even those common to both. Comm is the command that u need to for this work.

It requires two sorted file and lists the differing entries in different columns.

$comm file1 file2

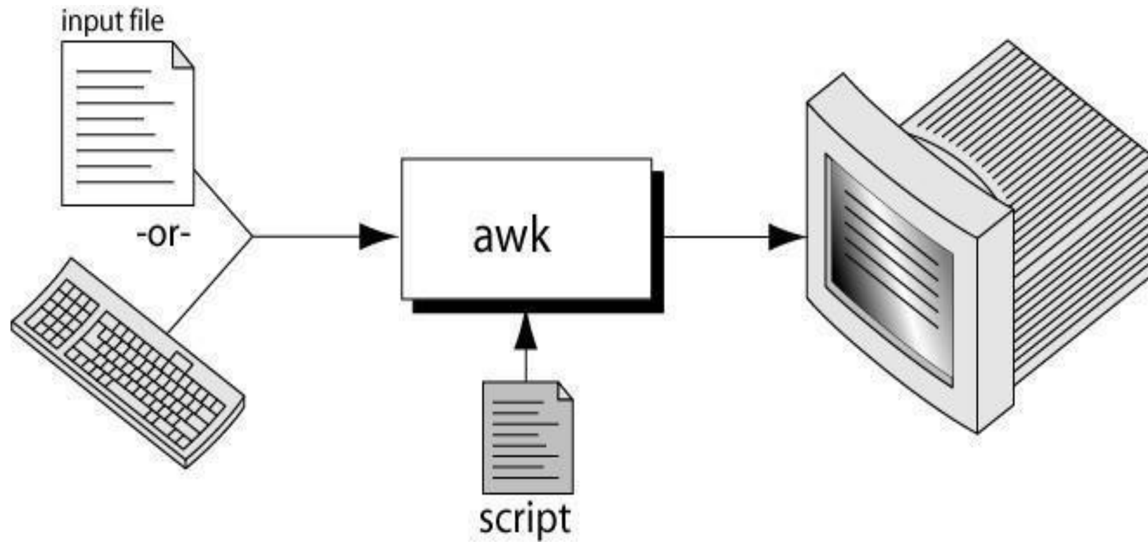
Comm display a three-column output.

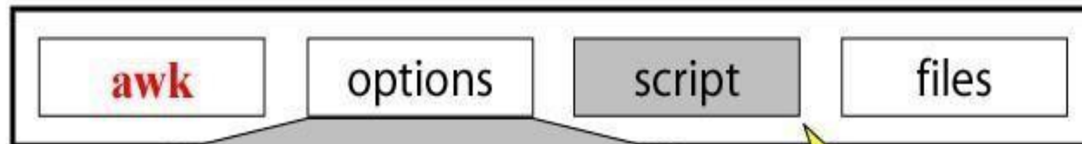
diff: converting one file to another

Diff takes different approach to displaying the differences.

When used with the same files it produces a detailed output.

The Command: awk



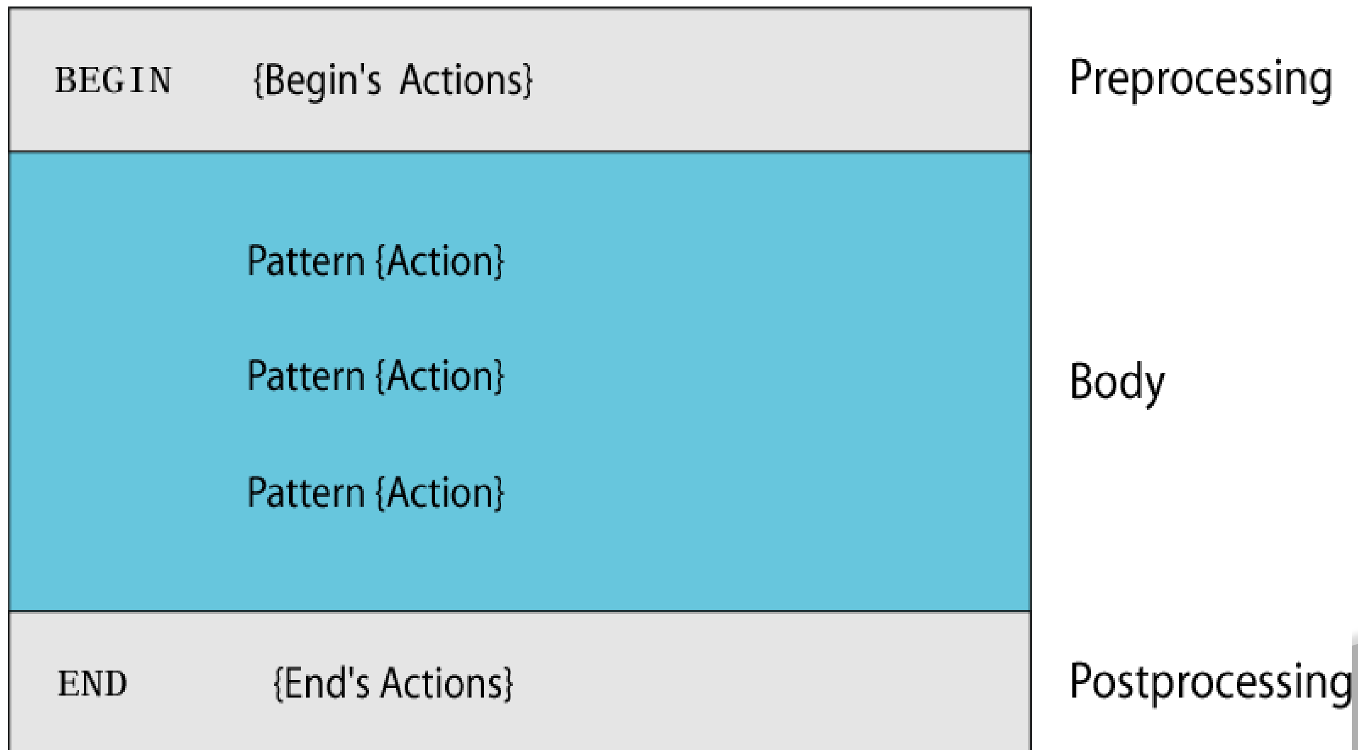


-F: input field separator
-f: script file

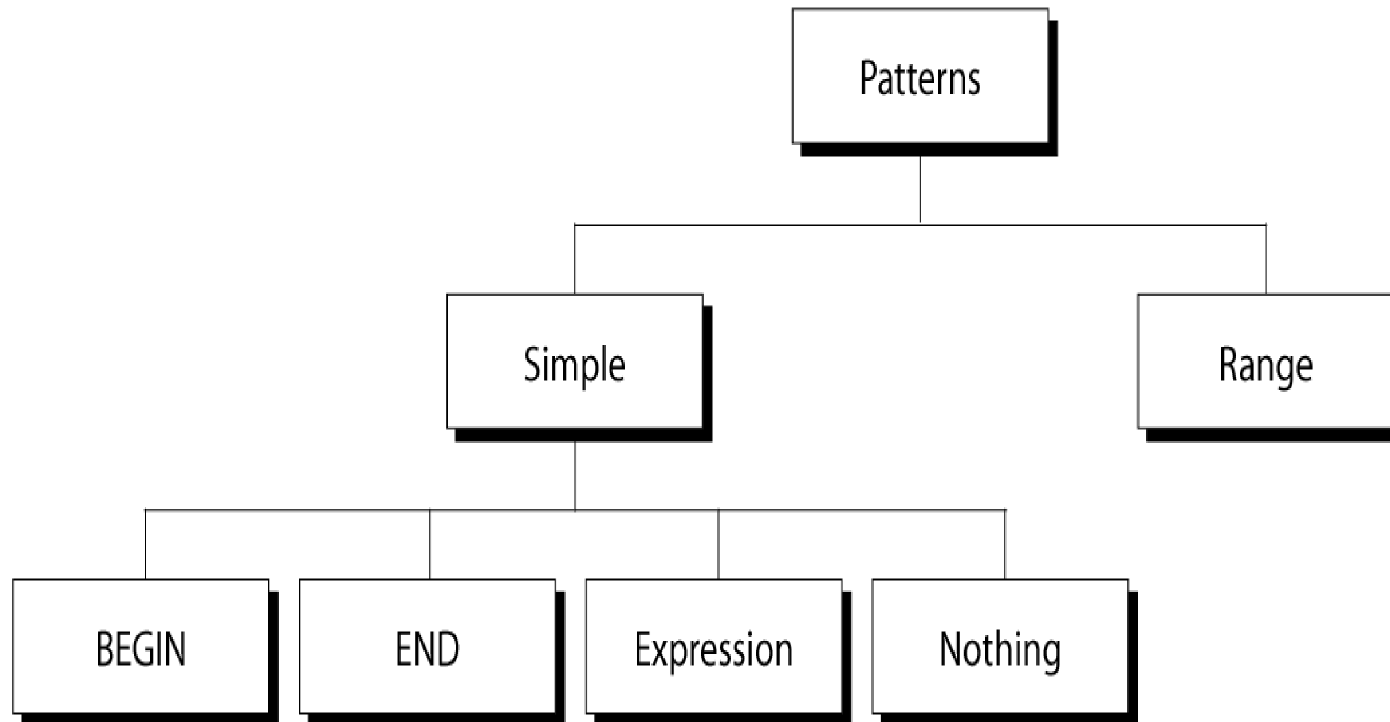
usually in a separate file

awk Scripts

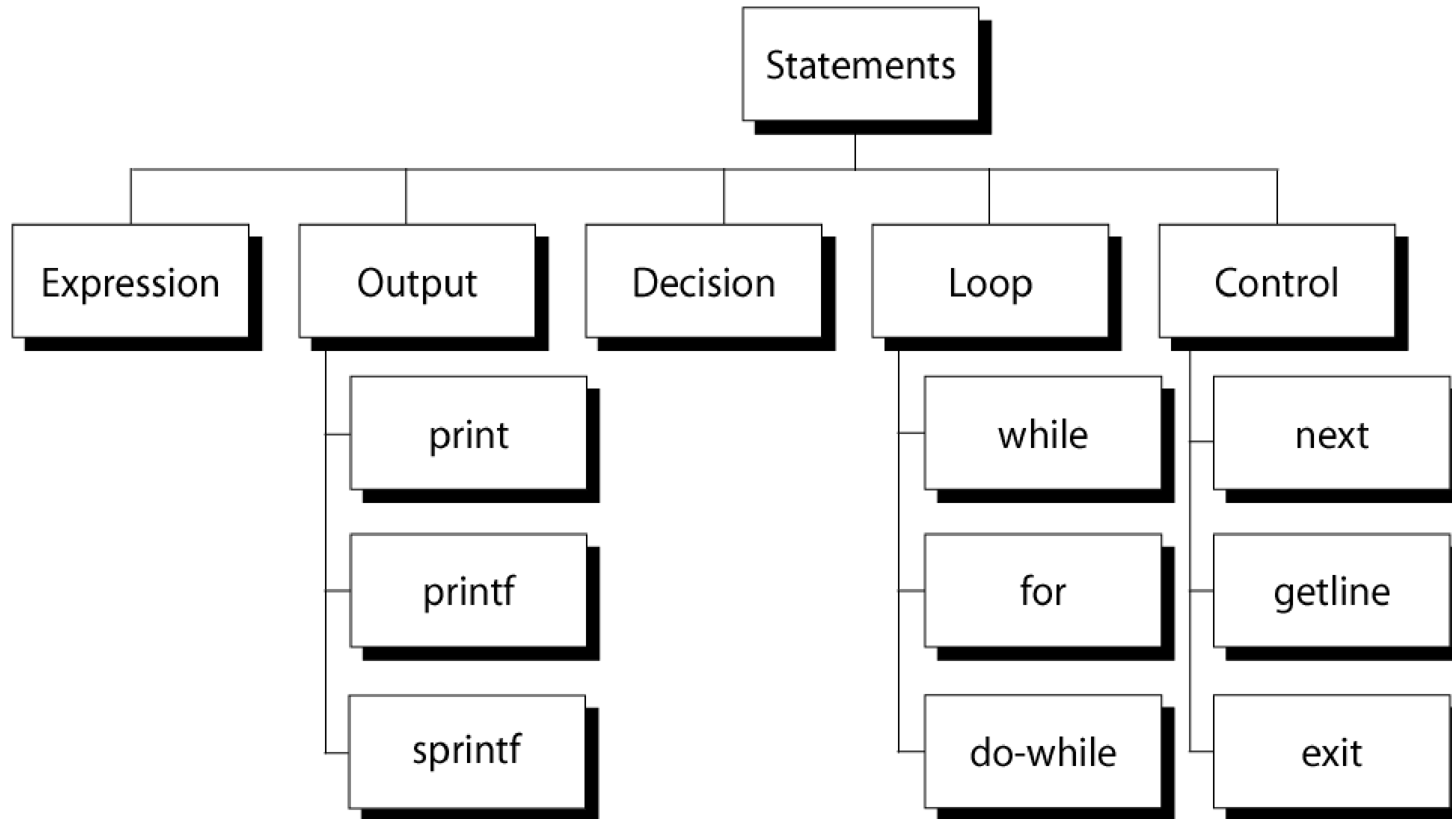
1.awk scripts are divided into three major parts:



Categories of Patterns



awk Actions



awk Actions

Awk: Aho, Weinberger and Kernighan

Awk is not just a command, but a programming language too.

Syntax:

awk options ‗selection criteria {action}‘ file(s)

Simple filtering

awk ‗/Simpsons/ { print }‘ homer|Simpsons

Splitting a line into fieldsawk -F ¦ ‗/Simpsons/ {print $1}‘ homer

tr: translating characters

tr command manipulates individual characters in a character stream.

tr options expr1 expr2< standard input

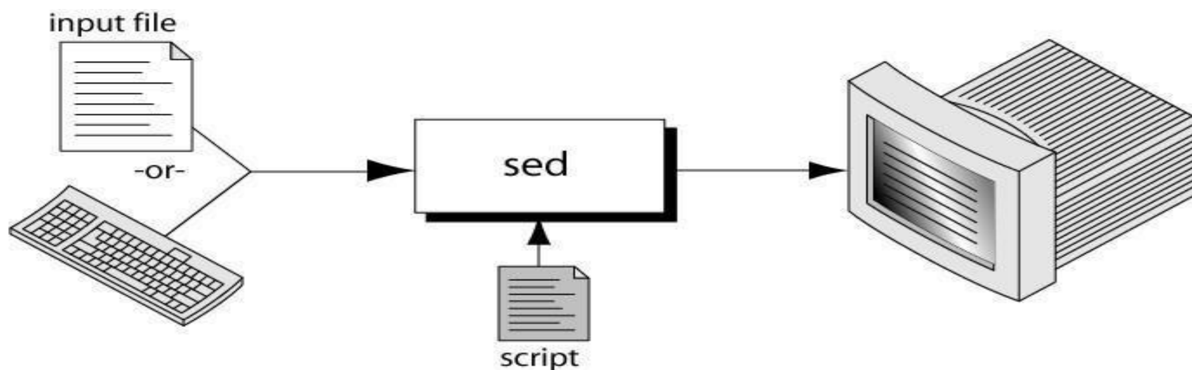
It takes input only from the standard input, it does not take input a file name as its argument.

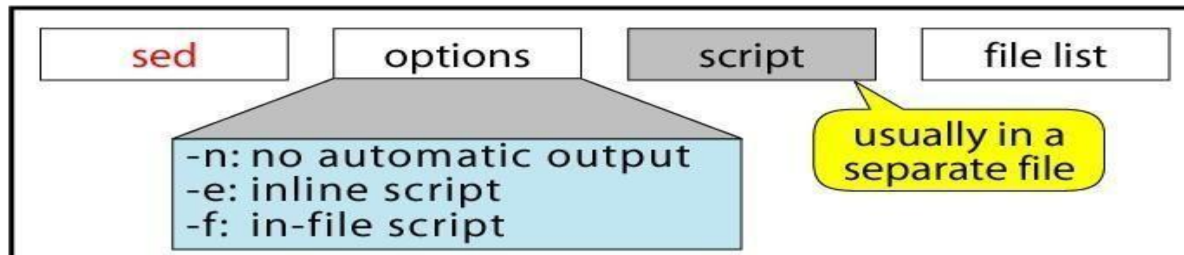
SED

What is sed?

•A non-interactive stream editor

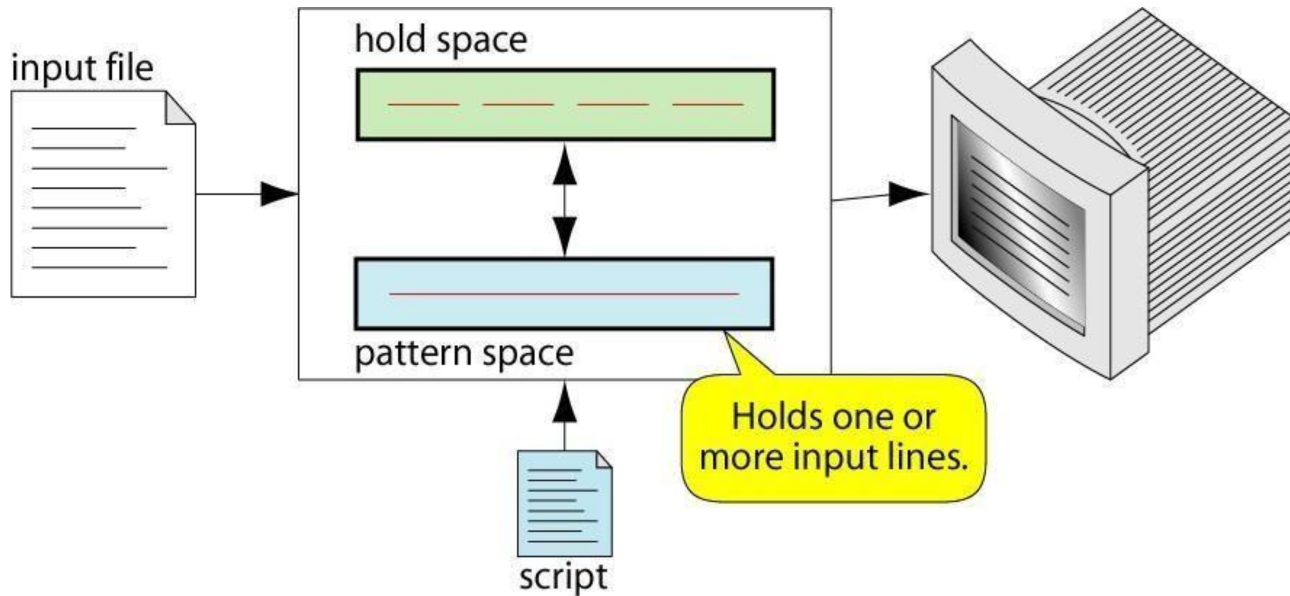
•Interprets sed instructions and performs actions





Sed Operation

How Does sed Work?



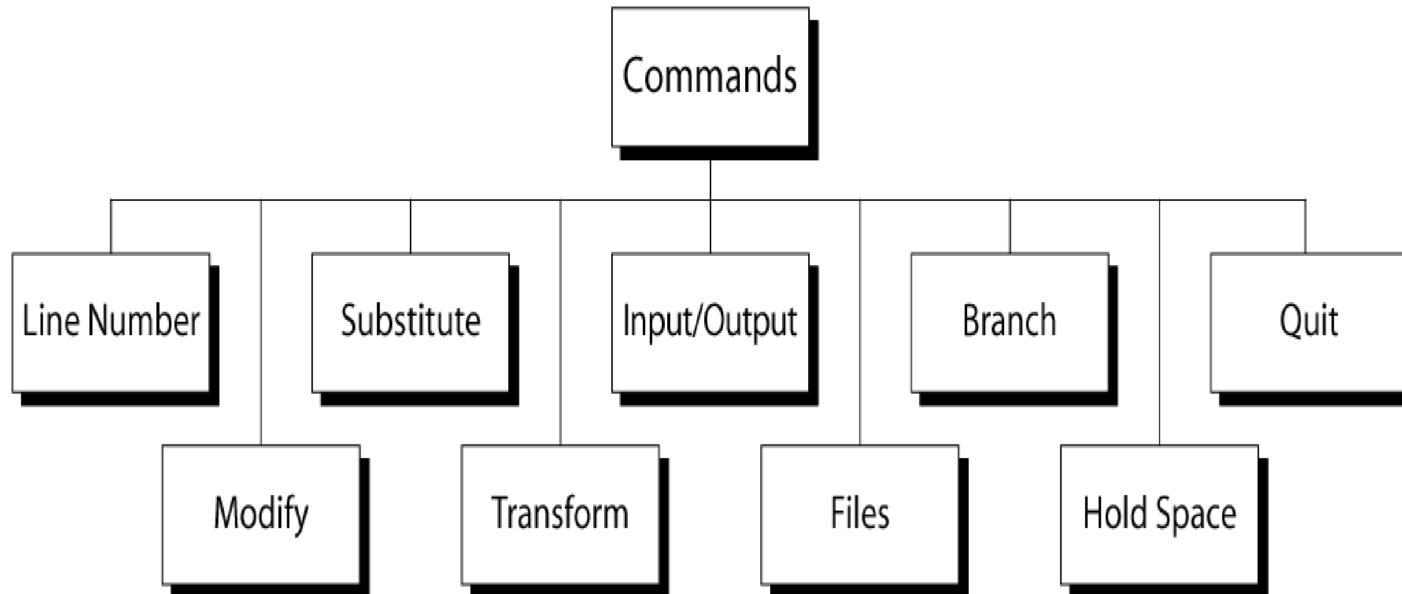
SED

sed reads line of input line of input is copied into a temporary buffer called pattern space editing commands are applied subsequent commands are applied to line in the pattern space, not the original input line once finished, line is sent to output (unless –n option was used) line is removed from pattern space sed reads next line of input, until end of file

sed instruction format(Sed Addresses):

- Address determines which lines in the input file are to be processed by the command(s)
- If no address is specified, then the command is applied to each input line
- address types:
- Single-Line address
- Set-of-Lines address
- Range address
- Nested address Single-Line Address
- Specifies only one line in the input file
- special: dollar sign ($) denotes last line of input file

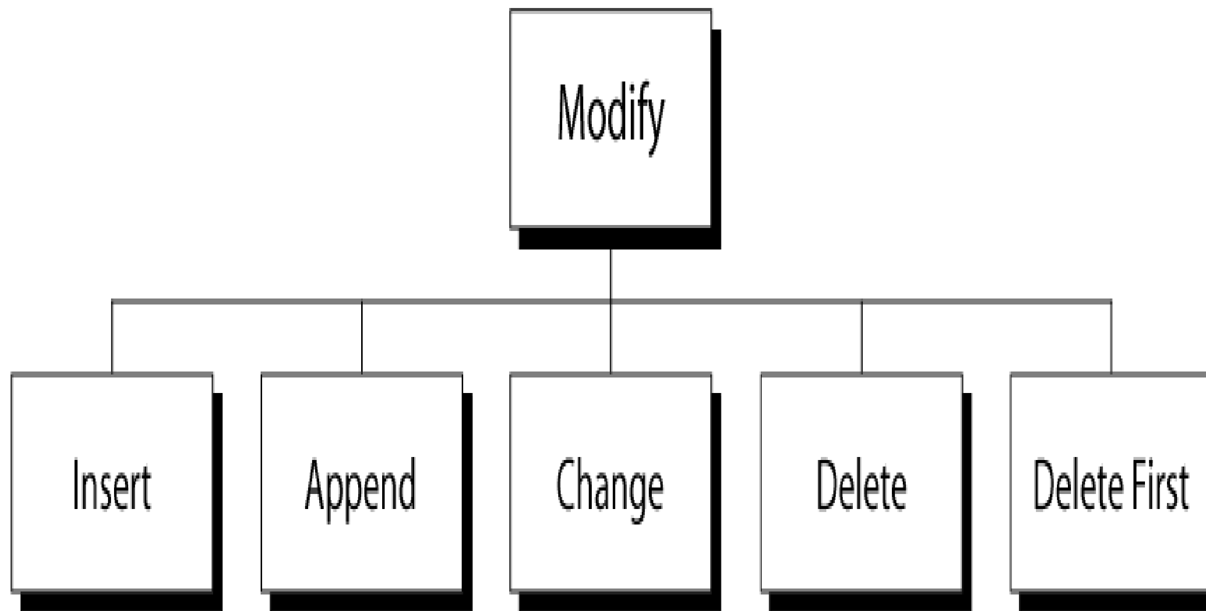
sed commands



sed commands

Line Number

line number command (=) writes the current line number before each matched/output line_modify commands:

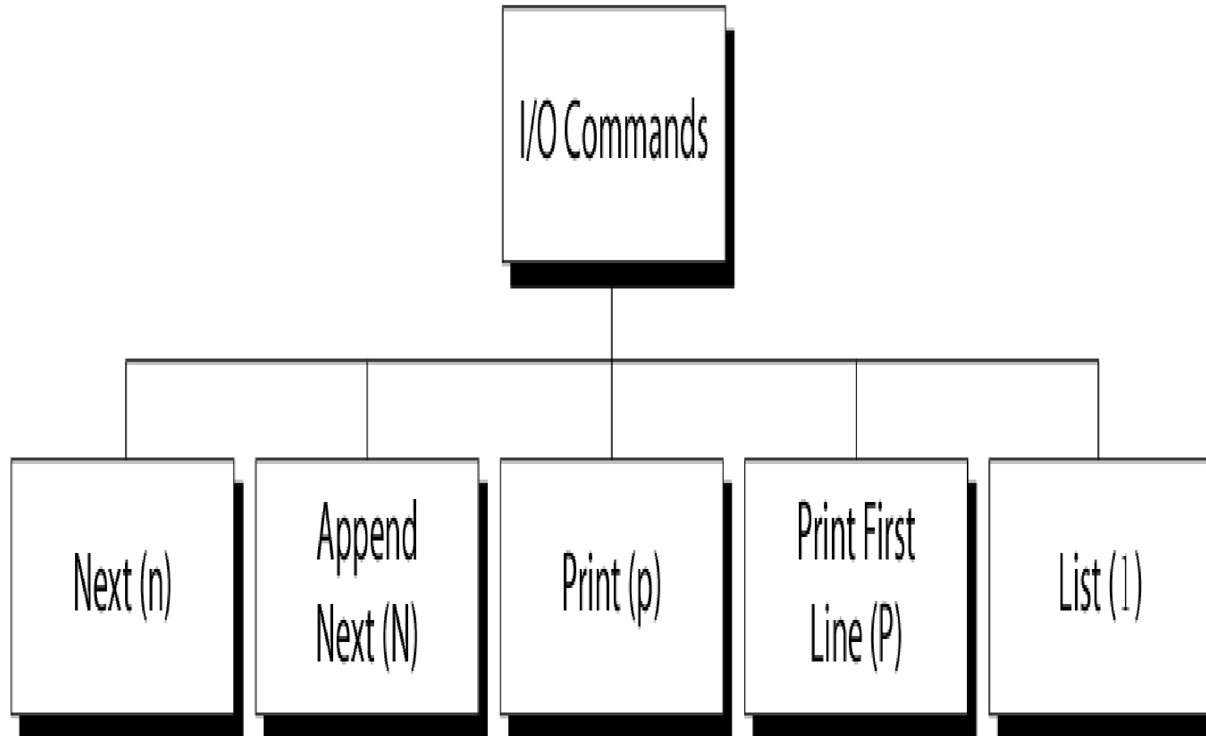


Regular Expressions: use with sed

2000 IARE EDUCATION FOR LIBERATION

Metacharacter	Description/Matches…
.	Any one character, except new line
*	Zero or more of preceding character
^	A character at beginning of line
$	A character at end of line
\char	Escape the meaning of *char* following it
[ ]	Any one of the enclosed characters
\(\)	Tags matched characters to be used later
x\{m\}	Repetition of character x, m times
\<	Beginning of word
\>	End of word

sed i/o commands



UNIT-II

WORKING WITH BOURNE SHELL

CLOs	Course Learning Outcome
CLO 6	Understand the shell responsibilities of different types of shells
CLO 7	Develop shell scripts to perform more complex tasks in shell programming environment.
CLO 8	Illustrate file processing operations such as standard I/O and formatted I/O.
CLO 9	Illustrate directory operations such as standard I/O and formatted I/O.

Shell

- The **Bourne shell**, or **sh**, was the default Unix shell of Unix Version 7. It was developed by Stephen Bourne, of AT&T Bell Laboratories.
- A **Unix shell**, also called "the command line", provides the traditional user interface for the Unix operating system and for Unix-like systems. Users direct the operation of the computer by entering command input as text for a shell to execute.

There are many different shells in use. They are

Bourne shell (**sh**)

C shell (**csh**)

Korn shell (**ksh)**

Shell responsibilities

- Program Execution
- Variable and Filename Substitution
- I/O Redirection
- Pipeline Hookup
- Environment Control
- Interpreted Programming Language

Shell responsibilities

Program Execution:

- The shell is responsible for the execution of all programs that your request from your terminal.
- Each time you type in a line to the shell, the shell analyzes the line and then determines what to do.
- The line that is typed to the shell is known more formally as the command line. The shell scans this command line and determines the name of the program to be executed and what arguments to pass to the program.

Shell responsibilities

Variable and Filename Substitution:

Like any other programming language, the shell lets you assign values to variables. Whenever you specify one of these variables on the command line, preceded by a dollar sign, the shell substitutes the value assigned to the variable at that point.

I/O Redirection:

It is the shell's responsibility to take care of input and output redirection on the command line. It scans the command line for the occurrence of the special redirection characters <, >, or >>.

Pipes

- Standard I/p & standard o/p constitute two separate streams that can be individually manipulated by the shell. The shell connects these streams so that one command takes I /p from other using **pipes**.
- Who produces the list of users , to save this o/p in a file use
- $who > user.lst
- To count the no. of lines in this user.lst use $wc –l <user.lst

Redirection

- Many of the commands that we used sent their output to the terminal and also taking the input from the keyboard. These commands are designed that way to accept not only fixed sources and destinations. They are actually designed to use a character stream without knowing its source and destination.
- A stream is a sequence of bytes that many commands see as input and output. Unix treats these streams as files and a group of unix commands reads from and writes to these files.

standard files

- There are 3 streams or standard files. The shell sets up these 3 standard files and attaches them to user terminal at the time of logging in.
- Standard i/p ----default source is the keyboard.
- Standard o/p ----default source is the terminal.
- Standard error ----default source is the terminal.

standard files

IARE 2000 EDUCATION FOR LIBERATION

- Instead of input coming from the keyboard and output and error going to the terminal, they can be **redirected** to come from or go to any file or some other device.
- **Standard o/p:** It has 3 sources. The terminal, default source
- A file using redirection with >, >>
- Another program using a pipeline.
- Using the symbols >,>> u can redirect the o/p of a command to a file.
- $who> newfile

standard files

- If the output file does not exist the shell creates it before executing the command. If it exists the shell overwrites it.
- $who>> newfile
- **Standard i/p:**
- The keyboard,default source A file using redirection with <
- Another program using a pipeline.
- $wc < calc.lst or
- $wc calc.lst or $wc

standard files

- **Standard Error:**
- When u enter an incorrect command or trying to open a non existing file, certain diagnostic messages show up on the screen. This is the standard error stream.
- Trying to cat nonexistent file produces the error stream.
- $cat bar
- Cat: cannot open bar :no such file or directory

Standard Files

- The standard error stream can also be redirected to a file.
- $cat bar 2> errorfile
- Here 2 is the filedescriptor for standard error file.
- Each of the standard files has a number called a file descriptor, which is used for identification.
- 0—standard i/p 1---standard o/p
- 2---standard error

Here Documents

- There are occasions when the data of ur program reads is fixed & fairly limited.
- The shell uses << symbols to read data from the same file containing the script. This referred to as a here document, signifying that the data is here rather than in a separate file.
- Any command using standard i/p can also take i/p from a here document.
- This feature is useful when used with commands that don’t accept a file name as argument.

Shell Meta characters

The shell consists of large no. of meta characters. These characters plays vital role in Unix programming.

Types of metacharacters:

1.File substitution

2.I/O redirection

3.Process execution

4.Quoting metacharacters

5.Positional parameters

6.Special characters

Filename Substitution:

- These metacharacters are used to match the filenames in a directory.
- Metacharacter significance
- * matches any no. of characters
- ? matches a single character
- [ijk] matches a single character either i,j,k
- [!ijk] matches a single character that is not
- an I,j,k

I/O redirection

- These special characters specify from where to take i/p & where to send o/p.
- >- to send the o/p to a specific file to take i/p from specific location but not from keyboard.
- >>- to save the o/p in a particular file at the end of that file without overwriting it.
- <<- to take i/p from standard i/p file.

Process execution:

IARE

- It is used when u want to execute more then one command at $ prompt.
- **Eg:**$date; cat f1>f2
- () –used to group the commands.
- **Eg:**(date; cat f1) >f2
- -used to execute the commands in background mode.
- **Eg:**$ls &

  this is used when u want to execute the second command only if the first command executed successfully.
- **Eg:**$grep Unix f1 && echo Unix found $cc f1 && a.out
- - used to execute the second command if first command fails.
- **Eg:**$grep unix f1 || echo no unix

Quoting

- \ (backslash)- negates the special property of the single character following it.

Eg:

- $echo \? * \?
- ?*?
- ‗‗(pair of single quotes)-negates the special properties of all enclosed characters.

Eg:$echo ‗send $100 to whom?‘

Quoting

- ——(pairof double quotes)-negates the special properties of all enclosed characters except $,`,\ .
- Eg:
- $echo —todaydate is $date or
- $echo —todaydate is `date` —

Positional parameters:

- $0- gives the name of the command which is being executed.
- $*-gives the list of arguments.
- $#-gives no. of arguments.

Special parameters

- **Special parameters:**
- `$$`- gives PID of the current shell.
- `$?`-gives the exit status of the last executed command.
- `$!`-gives the PID of last background process.
- `$-` -gives the current setting of shell.

Shell variables

- U can define & use variables both in the command line and shell scripts. These variables are called shell variables.
- No type declaration is necessary before u can
- use a shell variable.
- Variables provide the ability to store and manipulate the information with in the shell program. The variables are completely under the control of user.
- Variables in Unix are of two types.
  - 1.Unix defined or system variables
  - 2.User defined variables

User-defined variables:

Generalized form: variable=value.

Eg: $x=10

$echo $x 10

To remove a variable use unset.

$unset x

All shell variables are initialized to null strings by default. To explicitly set null values use

x= or x=‗‘ or x=—

To assign multiword strings to a variable use $msg=‗u have a mail‘

Environment Variables

- **$HOME -**Home directory
- **$PATH -**List of directories to search for commands
- **$PS1 -** Command prompt
- **$PS2 -**Secondary prompt
- **$SHELL -** Current login shell
- **$0 -** Name of the shell script
- **$# -** No . of parameters passed
- **$$ -** Process ID of the shell script

Shell commands

read:

- The read statement is a tool for taking input from the user i.e. making scripts interactive. It is used with one or more variables. Input supplied through the standard input is read into these variables.
- $read name
- What ever u entered is stored in the variable name.
- Printf:Printf is used to print formatted o/p. printf "format" arg1 arg2 ...

Eg:$ printf "This is a number: %d\n" 10 This is a number: 10

- $
- Printf supports conversion specification characters like %d, %s ,%x ,%o....

control structures

- **If conditional:**
- The if statement takes two-way decisions depending on the fulfillment of a certain condition. In shell the statement uses following form.
- If command is successful then execute commands else execute command if.
- **Eval**:eval   scans the command line twice before executing it. General form for eval is
-             eval command-line

- **Eg:**$ cat last
- eval echo \$$#
- $las tone two three four four

File Types:

The types of files are:

1. Regular file
2. Directory file
3. Block special file
4. Character special file
5. FIFO
6. Socket
7. Symbolic link

File Types

Macro	Type of file
S_ISREG()	regular file
S_ISDIR()	directory file
S_ISCHR()	character special file
S_ISBLK()	block special file
S_ISFIFO()	pipe or FIFO
S_ISLNK()	symbolic link
S_ISSOCK()	socket

System calls for file I/O operations

- To use the services in the OS Unix offers some special functions known as system calls. The system call is a task which performs very basic functions that requires communication with CPU, memory and other devices.
- UNIX system calls are used to manage the file system, control processes, and to provide inter process communication.
- Types of system calls in UNIX:

1.Open 2.create 3. read

4.write 5.Close 6.lseek

7.stat 8.fstat 9.ioctl

10.umask 11.dup 12.dup2

Directories

chmod and fchmod Functions:

These two functions allow us to change the file access permissions for an existing file.

```
#include <sys / stat.h>
int chmod(const char*pathname, mode t mode);
int fchmod(int filedes, mode tmode);
```

Both return: 0 if OK, -1 on error.

Scanning Directories

- The directory functions are declared in a header file, **dirent.h**. They use a structure, **DIR**, as a basis for directory manipulation.
- Here are these functions:
- Opendir
- closedir
- Readdir
- telldir
- seekdir

Scanning Directories

- The **opendir** function opens a directory and establishes a directory stream.
- The **readdir** function returns a pointer to a structure detailing the next directory entry in the directory stream
- The **telldir** function returns a value that records the current position in a directory stream.
- The **seekdir** function sets the directory entry pointer in the directory stream given by **dirp**.
- The **closedir** function closes a directory stream and frees up the resources associated with it.

IARE
2000
EDUCATION FOR LIBERATION

PROCESS AND SIGNALS

CLOs	Course Learning Outcome
CLO 10	Understand process structure, scheduling and management through system calls.
CLO 11	Generalize signal functions to handle interrupts by using system calls.

Process Identifier

- Every process has a unique process ID, a non-negative integer.
- There are two special processes.
- Process ID is usually the schedule process and is often known as the ‘swapper‘.
- The program files for this process loss /etc/init in older version of UNIX and is /sbin/init is newer version.
- init‘usually reads the system dependent initialization files and brings the system to a certain state.

Process,Process structure

- Every process has a unique process ID, a non-negative integer. Unique, process IDs are reused.
- As processes terminate, their IDs become candidates for reuse.
- Process ID 0 is usually the scheduler process and is often known as the swapper.
- Process ID 1 is usually the init process and is invoked by the kernel at the end of the bootstrap procedure. The init process never dies.
- process ID 2 is the page daemon.

Process,Process structure

- In addition to the process ID, there are other identifiers for every process. The following functions return these identifiers.
- #include <unistd.h> pid_t getpid (void);
- Returns: process ID of calling process. pid_t getppid (void);
- Returns: parent process ID of calling process. uid_t getuid (void);
- Returns: real user ID of calling process.

Process Table

Pointer	Process state
Process number	
Program counter	
Registers	
Memory limits	
List of open files	
⋮	

2000 IARE EDUCATION FOR LIBERATION

viewing processes

- It is responsible for scheduling running of user and other processes.
- It is responsible for allocating memory.
- It is responsible for managing the swapping between memory and disk.
- It is responsible for moving data to and from the peripherals.
- it receives service requests from the processes and honors them.

System processes

- finding host/domain name and IP address - **hostname**
- test network connection – **ping**
- getting network configuration – **ifconfig**
- Network connections, routing tables, interface statistics – **netstat**
- query DNS lookup name – **nslookup**
- communicate with other hostname – **telnet**
- outing steps that packets take to get to network host – **traceroute**
- checking status of destination host - **telnet**

Starting new processes

- Treat current process
- Select process
- Switch process

Waiting for a process

2000 IARE EDUCATION FOR LIBERATION

- All of these system calls are used to wait for state changes in a child of the calling process, and obtain information about the child whose state has changed.

- A state change is considered to be: the child terminated; the child was stopped by a signal; or the child was resumed by a signal.

wait3 and wait4 functions

- #include<sys/types.h>
- #include<sys/wait.h>
- pid_t wait3(int *statloc, int options, struct rusage *rusage);
- pid_t wait4(pid_t pid, int *statloc, int options, struct rusage *rusage) Both return: process ID if OK, 0, or -1 on error
- The resource information includes information such as the amount of user CPU time , the amount of system CPU time, number of page faults, number of signals received.

Process termination

- Now that we know what goes on when a process gets created, what is happening when we don't need it anymore?
- A process can exit using the _exit system call, this will free up the resources that process was using for reallocation
- There is another way to terminate a process and that involves using signals, which we will discuss soon.

Orphan Processes

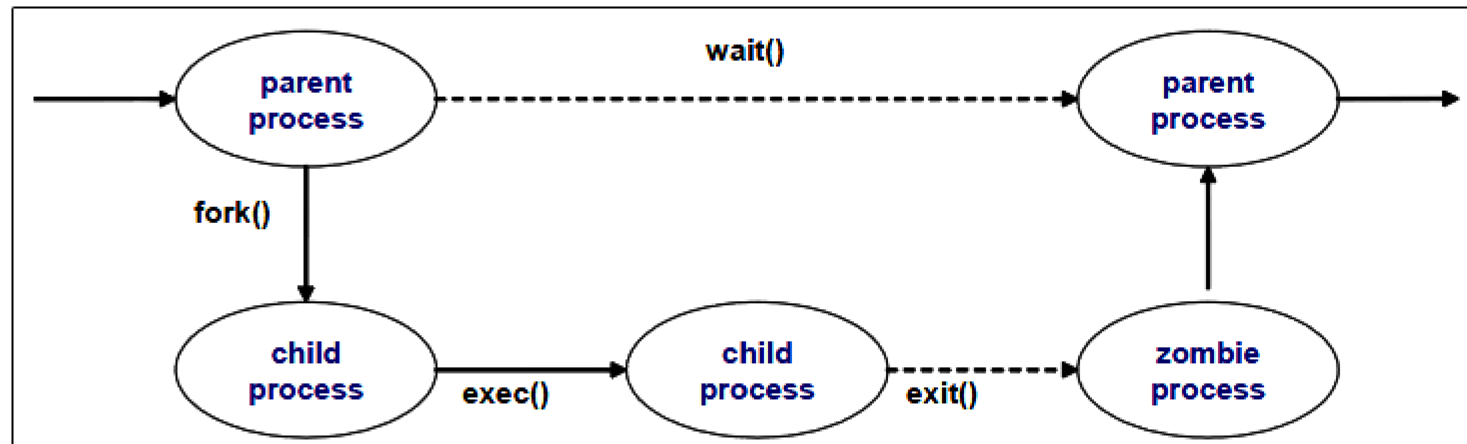
- **When a parent process dies before a child process, the kernel knows that it's not going to get a wait call, so instead it makes these processes "orphans" and puts them under the care of init (remember mother of all processes).**

- Init will eventually perform the wait system call for these orphans so they can die.

Zombie Processes

- When a child process terminates, an association with its parent survives until the parent in turn either terminates normally or calls **wait.**

- This terminated child process is known as a **zombie process**.



System call interface for process management

Services Provided by System Calls

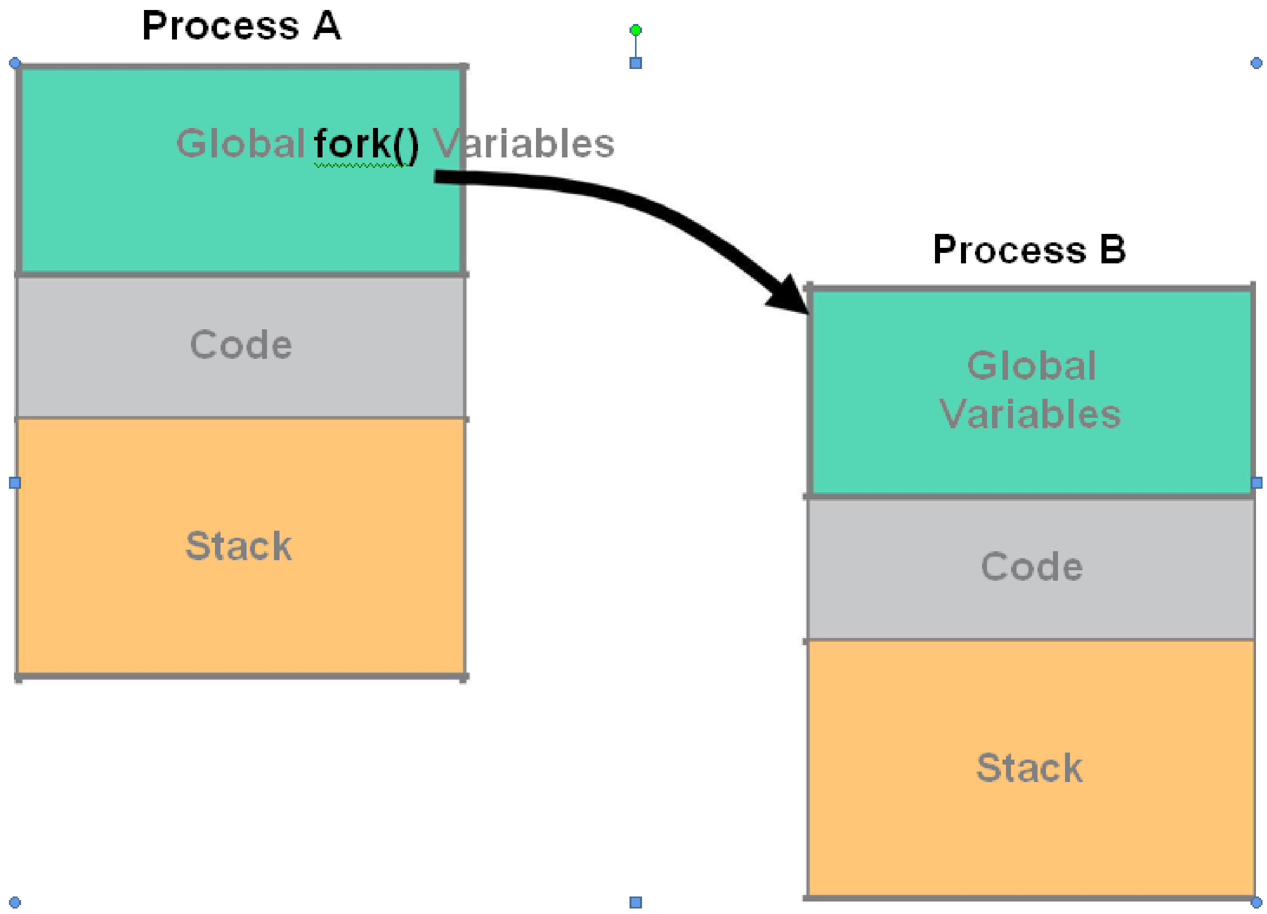
- Process creation and management
- Main memory management
- File Access, Directory and File system management
- Device handling(I/O)
- Protection
- Networking, etc.

System call interface for process management

There are 5 different categories of system calls

- **Process control**
– end, abort, create, terminate, allocate and free memory.
  File management
– create, open, close, delete, read file etc.
- **Device management**
- **Information maintenance**
- **Communication**

fork()



fork Function

- An existing process can create a new one by calling the fork function.
- #include <unistd.h> pid_t fork(void);
- Returns: 0 in child, process ID of child in parent, -1 on error.
- The new process created by fork is called the child process. This function is called once but returns twice. The only difference in the returns is that the return value in the child is 0, whereas the return value in the parent is the process ID of the new child.

Signals

- A signal is an electrical or electromagnetic current that is used for carrying data from one device or network to another.
- It is the key component behind virtually all:

  Communication

  Computing

  Networking

  Electronic devices

  A signal can be either analog or digital.

Signal functions

Signal	Value	Description
1	SIGHUP	Hang up the process.
2	SIGINT	Interrupt the process.
3	SIGQUIT	Stop the process.
9	SIGKILL	Unconditionally terminate the process.
15	SIGTERM	Terminate the process if possible.
17	SIGSTOP	Unconditionally stop, but don't terminate the process.
18	SIGTSTP	Stop or pause the process, but don't terminate.
19	SIGCONT	Continue a stopped process.

The signal() Function

The signal() Function

- The simplest way to register signal handler function with the kernel is by using the **signal()** function.

- Here is the syntax of **signal()** function :

```
#include <signal's>
typedef void (*sighandler_t)(int);
```

Signal dispositions

- Each signal has a current *disposition*, which determines how the process behaves when it is delivered the signal.
- The entries in the "Action" column of the tables below specify the default disposition for each signal.

kill and raise Functions

kill and raise Functions

- The kill function sends a signal to a process or a group of processes. The raise function allows a process to send a signal to it.

```
#include<sys/types.h>
#include<signal.h>
int kill(pid_t pid, int signo);
int raise(int signo);
```

Both return: 0 if OK, -1 on error

2000 IARE EDUCATION FOR LIBERATION

UNIT-IV

DATA MANAGEMENT

CLOs	Course Learning Outcome
CLO 12	Illustrate memory management of file handling through file/region lock
CLO 13	Design and implement inter process communication (IPC) in client server environment by using pipe.
CLO 14	Design and implement inter process communication (IPC) in client server environment by using named Pipes
CLO 15	Illustrate client server authenticated communication in IPC through messages queues, semaphores
CLO 16	Illustrate client server authenticated communication in IPC through shared memory.

Managing memory

- **Malloc:** The **malloc**() function allocates size bytes and returns a pointer to the allocated memory. The memory is not initialized. If size is 0, then **malloc**() returns either NULL, or a unique pointer value.

- **free:** The **free**() function frees the memory space pointed to by ptr, which must have been returned by a previous call to **malloc**(), **calloc**() or **realloc**(). Otherwise, or if free(ptr) has already been called before, undefined behavior occurs. If ptr is NULL, no operation is performed.

Managing memory

- **realloc:**The **realloc**() function changes the size of the memory block pointed to by ptr to size bytes. The contents will be unchanged in the range from the start of the region up to the minimum of the old and new sizes. If the new size is larger than the old size, the added memory will not be initialized.

- **calloc:** The calloc() function allocates memory for an array of nmemb elements of size bytes each and returns a pointer to the allocated memory. The memory is set to zero. If nmemb or size is 0, then calloc() returns either NULL, or a unique pointer value that can later be successfully passed to free().

File locking

- File locking is a mechanism which allows only one process to access a file at any specific time. By using file locking mechanism, many processes can read/write a single file in a safer way.

Creating lock files

- When a file can be accessed by more than one process, a synchronization problem occurs.

- What happens if two processes try to write in the same file location? Or again, what happens if a process reads from a file location while another process is writing into it?

File locking

- In traditional Unix systems, concurrent accesses to the same file location produce unpredictable results.
- However, Unix systems provide a mechanism that allows the processes to *lock* a file region so that concurrent accesses may be easily avoided.
- Traditional BSD variants implement advisory locking through the flock( )system call. This call does not allow a process to lock a file region, only the whole file.
- Traditional System V variants provide the lockf( ) library function, which is simply an interface to fcntl( ).

use of read and write with locking

- File Locking is a simple mechanism for coordinating file accesses. There are two types of locking mechanisms, Mandatory and Advisory.

- Advisory locks are just conventions

- If one process P1 locks a file, kernel doesn’t stop any other process(say P2) from modifying that file. But if the other process P2 obeys the same convention as the process P1, it can check before modifying that the file is locked by some other process and thus it wouldn’t be correct to modify it.

- advisory locks require proper coordination between the processes.

Deadlocks

- A deadlock is a situation in which two computer programs sharing the same resource are effectively preventing each other from accessing the resource, resulting in both programs ceasing to function.

- The earliest computer operating systems ran only one program at a time. All of the resources of the system were available to this one program.

- Later, operating systems ran multiple programs at once, interleaving them.

Deadlocks

- This led to the problem of the deadlock. Here is the simplest example.

- Program 1 requests resource A and receives it.
- Program 2 requests resource B and receives it.
- Program 1 requests resource B and is queued up, pending the release of B.
- Program 2 requests resource A and is queued up, pending the release of A.

Inter process communication

- Inter process communication (IPC) includes thread synchronization and data exchange between thread
- If threads belong to the same process, they execute in the same address space, i.e. they can access global (static) data or heap directly, without the help of the operating system's beyond the process boundaries.
- However, if threads belong to different processes, they cannot access each other’s address spaces without the help of the operating system.

Inter process communication

- There are two fundamentally different approaches in IPC:
- processes are residing on the same computer
- processes are residing on different computers
- The first case is easier to implement because processes can share memory either in the user space or in the system space. This is equally true for uniprocessors and multiprocessors.

Inter process communication

- The interpretation of the pid argument for waitpid depends on its value:
- pid == -1 Waits for any child process.
- pid > 0   Waits for the child whose process ID equals pid. pid
- == 0 Waits for any child whose process group ID equals that of the calling process.
- pid < 1 Waits for any child whose process group ID equals the absolute value of pid.

PIPES

- A pipe is a **serial** communication device (i.e., the data is read in the order in which it was written), which allows a **unidirectional communication**.
- The data written to end is read back from the other end.
- The pipe is mainly used to communicate between two threads in a single process or between parent and child process.
- Pipes can only connect the related process. In shell, the symbol can be used to create a pipe.

Creating pipes

- The pipe() function provides a means of passing data between two programs and also allows to read and write the data.

- #include<unistd.h>
- int pipe(int file_descriptor[2]);

- pipe()function is passed with an array of file descriptors. It will fill the array with new file descriptors and returns zero. On error, returns -1 and sets the err no to indicate the reason of failure.

- The file descriptors are connected in a way that is data written to file_ descriptor [1] can be read back from the file_descriptor [0].

Pipe processing

- The process of passing data between two programs can be done with the help of popen() and pclose() functions.

```
#include<stdio.h>
FILE *popen(const char *command , const char *open-mode);
int pclose(FILE *stream_to_close);
```

PIPE CALLS

popen():

- The popen function allows a program to invoke another program as a new process and either write the data to it or to read from it

pclose():

- By using pclose(), we can close the filestream associated with popen() after the process started by it has been finished.

Parent and child processes

- We can invoke the standard programs, ones that don‘t expect a file descriptor as a parameter.

```
#include<unistd.h>
int dup(int file_descriptor);
int dup2(int file_descriptor_1, int file_descriptor_2);
```

Named pipes (FIFOs)

- Similar to pipes, but allows for communication between unrelated processes. This is done by naming the communication channel and making it permanent.

- Like pipe, FIFO is the unidirectional data stream.

Creation of FIFO:

- We can create a FIFO from the command line and within a program.
- To create from **command line** we can use either mknod or mkfifo commands.
- $ mknod filename p
- $ mkfifo filename

Semaphore

- While we are using threads in our programs in multi-user systems, multiprocessing system, or a combination of two, we may often discover critical sections in the code.

- This is the section where we have to ensure that a single process has exclusive access to the resource.

- The common form of semaphore is the *binary semaphore*, which will control a single resource, and its value is initialized to 0.

Shared Memory

IARE
EDUCATION FOR LIBERATION
2000

- Shared memory is a highly efficient way of data sharing between the running programs.
- It allows two unrelated processes to access the same logical memory
- It is the fastest form of IPC because all processes share the same piece of memory. It also avoids copying data unnecessarily.

Message queue

- This is an easy way of passing message between two process. It provides a way of sending a block of data from one process to another. The main advantage of using this is, each block of data is considered to have a type, and a receiving process receives the blocks of data having different type values independently.

Creation and accessing of a message queue:

```
#include<sys/msg.h>
int msgget(key_t key,int msgflg);
```

- The first parameter is the key value, which specifies the particular message queue. The special constant IPC_PRIVATE will create a private queue. But on some Linux systems the message queue may not actually be private.
- The second parameter is the flag value, which takes nine permission flags.

Unix kernel support for shared memory

There is a shared memory table in the kernel address space that keeps track of all shared memory regions created in the system

Each entry of the tables store the following data:

- Name
- Creator user ID and group ID.
- Assigned owner user ID and group ID.
- Read-write access permission of the region.
- The time when the last process attached to the region.
- The time when the last process detached from the region.
- The time when the last process changed control data of the region.
- The size, in no. of bytes of the region.

UNIX APIs for shared memory

shmget:

- Open and create a shared memory.
- Function prototype:

```
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/shm.h>
int shmget ( key_t key, int size, int flag );
```

- Function returns a positive descriptor if it succeeds or -1 if it fails.

Shmat

Shmat:

- Attach a shared memory to a process virtual address space.
- Function prototype:

```
#include<sys/types.h>
#include<sys/ipc.h>
 #include<sys/shm.h>
 void * shmat ( int shmid, void *addr, int flag );
```

- Function returns the mapped virtual address of he shared memory if it succeeds or -1 ifit fails.

Sockets

CLOs	Course Learning Outcome
CLO 17	Demonstrate socket connections, socket attributes, socket addresses
CLO 18	Demonstrate various client server applications on network using TCP.
CLO 19	Demonstrate various client server applications on network using UDP protocols.
CLO 20	Design custom based network applications using the sockets interface in heterogeneous platforms

What is a socket?

An interface between application and network. It is a communication mechanism that allows client / server to be developed either locally, on a single machine or across networks.

Two Types :

- TCP
- UDP

Identified by Two values

- An IP Address
- A Port Number

Sockets

- How to use sockets
  - Setup socket
    - Where is the remote machine (IP address, hostname)
    - What service gets the data (port)
  - Send and Receive
    - Designed just like any other I/O in unix
    - send -- write
    - recv -- read
  - Close the socket

The Socket Interface

- **Berkeley Sockets API**
  - Originally developed as part of BSD Unix (under gov't grant)
    - BSD = Berkeley Software Distribution
    - API=Application Program Interface
  - Now the most popular API for C/C++ programmers writing applications over TCP/IP
    - Also emulated in other languages: Perl, Tcl/Tk, etc.
    - Also emulated on other operating systems: Windows, etc.

Socket Creation in C: socket

int socket(int family, int type, int protocol);

- s: socket descriptor, an integer (like a file-handle)
- family: integer, communication domain, it specifies the network medium that the socket communication will use.

  e.g., AF_INET (IPv4 protocol) – typically used

- type: communication type

  SOCK_STREAM: reliable, 2-way, connection-based

  SOCK_DGRAM: unreliable, connectionless,

Step 1 – Setup Socket

- **Both client and server need to setup the socket**
  - int socket(int domain, int type, int protocol);
- domain
  - AF_INET -- IPv4 (AF_INET6 for IPv6)
- type
  - SOCK_STREAM -- TCP
  - SOCK_DGRAM -- UDP
- protocol
  - 0
- For example,
  - int sockfd = socket(AF_INET, SOCK_STREAM, 0);

Step 2 (Server) - Binding

- **Only server need to bind**
  - int bind(int sockfd, const struct sockaddr *my_addr, socklen_t addrlen);
- sockfd
  - file descriptor socket() returned
- my_addr
  - struct sockaddr_in for IPv4
  - cast (struct sockaddr_in*) to (struct sockaddr*)

What is that Cast?

- bind() takes in protocol-independent (struct sockaddr*)

```
struct sockaddr
 {
    unsigned short sa_family;   // address family
    char sa_data[14]; // protocol address
};
```

  - C's polymorphism
  - There are structs for IPv6, etc.

Step 2 (Server) - Binding contd.

- addrlen
  - size of the sockaddr_in

```
struct sockaddr_in saddr;
int sockfd;
unsigned short port = 80;

if((sockfd=socket(AF_INET, SOCK_STREAM, 0) < 0) {  // from back a couple slides
printf("Error creating socket\n");
...
}

memset(&saddr, '\0', sizeof(saddr));          // zero structure out
saddr.sin_family = AF_INET;                   // match the socket() call
saddr.sin_addr.s_addr = htonl(INADDR_ANY);    // bind to any local address
saddr.sin_port = htons(port);                 // specify port to listen on

if((bind(sockfd, (struct sockaddr *) &saddr, sizeof(saddr)) < 0) { // bind!
printf("Error binding\n");
```

What is htonl(), htons()?

- Byte ordering
  - Network order is big-endian
  - Host order can be big- or little-endian
    - x86 is little-endian
    - SPARC is big-endian
- Conversion
  - htons(), htonl(): host to network short/long
  - ntohs(), ntohl(): network order to host short/long

Step 3 (Server) - Listen

- **Now we can listen**
  - int listen(int sockfd, int backlog);
- sockfd
  - again, file descriptor socket() returned
- backlog
  - number of pending connections to queue
- For example,
  - listen(sockfd, 5);

Step 4 (Server) - Accept

- **Server must explicitly accept incoming connections**
  - int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen)
- sockfd
  - again... file descriptor socket() returned
- addr
  - pointer to store client address, (struct sockaddr_in *) cast to (struct sockaddr *)
- addrlen
  - pointer to store the returned size of addr, should be sizeof(*addr)
- For example
  - int isock=accept(sockfd, (struct sockaddr_in *) &caddr, &clen);

Close the Socket

- Don’t forget to close the socket descriptor, like a file
  - int close(int sockfd);
- Now server can loop around and accept a new connection when the old one finishes

Socket attributes

- The socket() function takes the following arguments:
- Domain
- Type
- Protocol

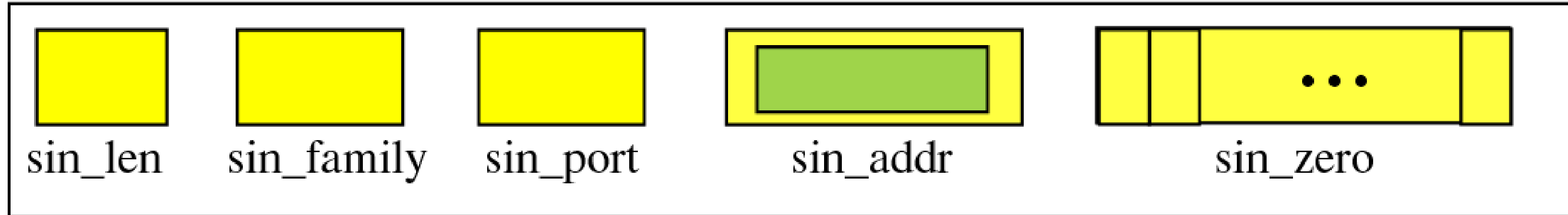
Protocol family constants

Family	Description
AF_INET	IPv4 protocol
AF_INET6	IPv6 protocol
AF_LOCAL	UNIX DOMAIN PROTOCOL
AF_ROUTE	Routing socket
AF_KEY	Key socket

Protocol of socket

protocol	description
IPPROTO_TCP	TCP transport protocol
IPPROTO_UDP	UDP transport protocol
IPPROTO_SCTP	SCTP transport protocol

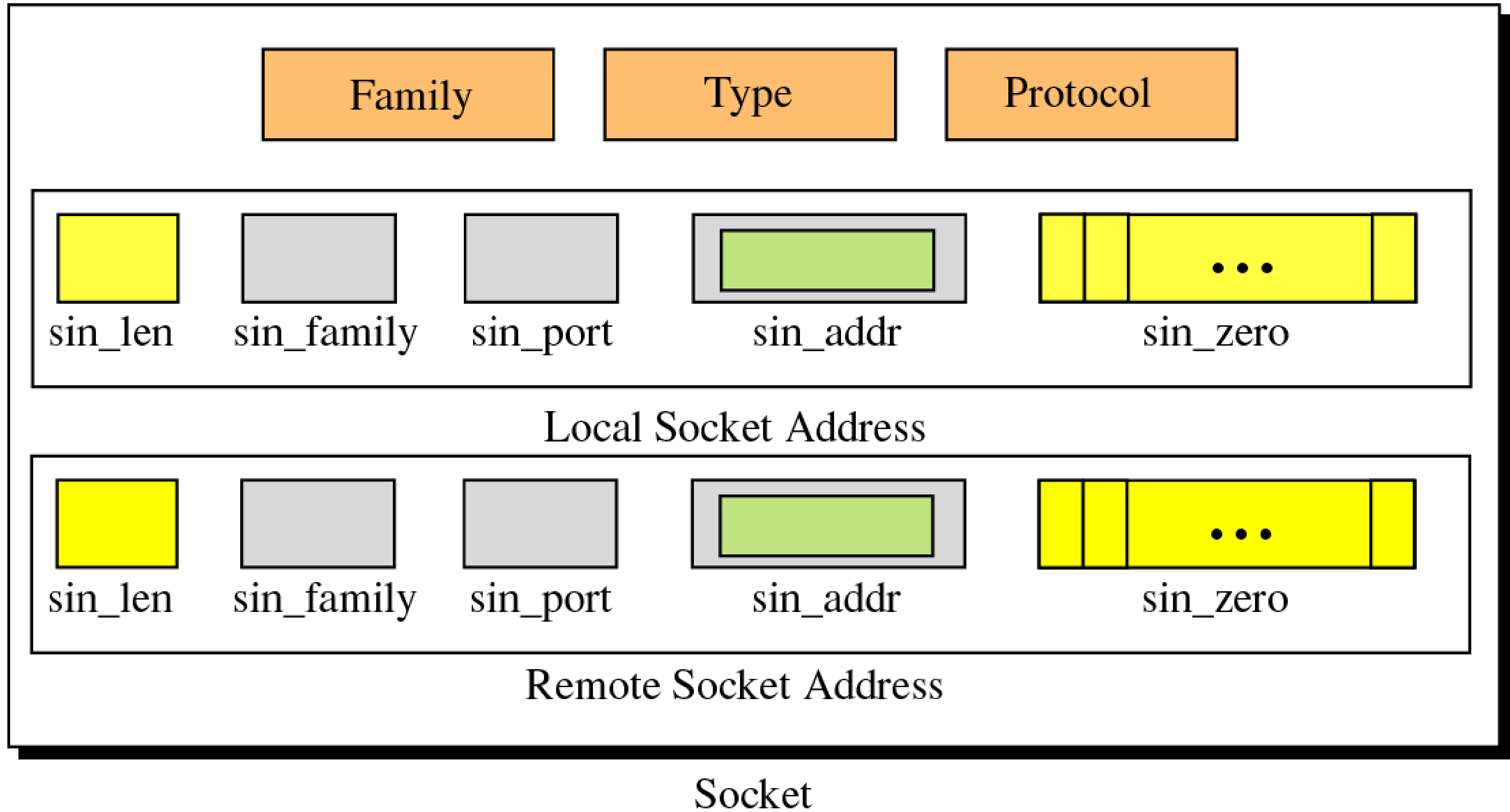
Socket address structure



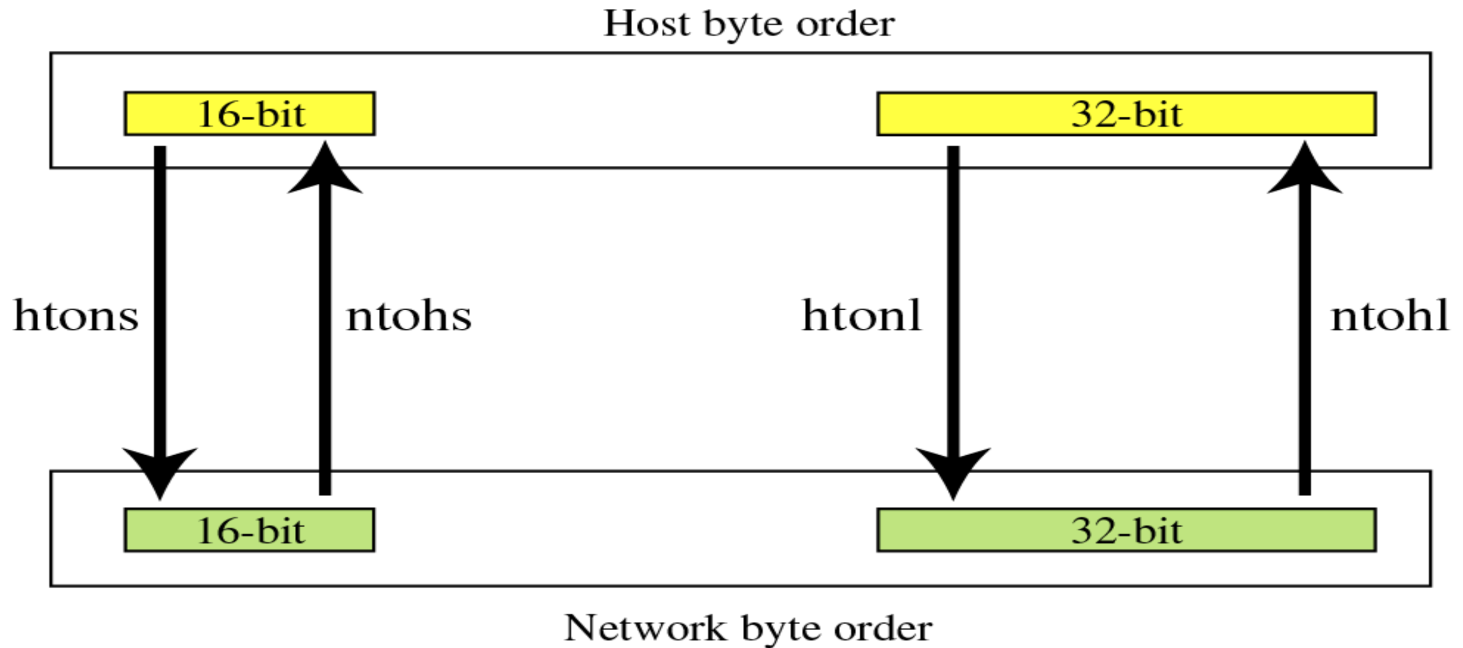
sockaddr_in

```
struct    sockaddr_in
{
          u_char              sin_len ;
          u_short             sin_family ;
          u_short             sin_port ;
          struct  in_addr     sin_addr ;
          char                sin_zero [8] ;
} ;
```

Socket Structure



Byte-Order Transformation

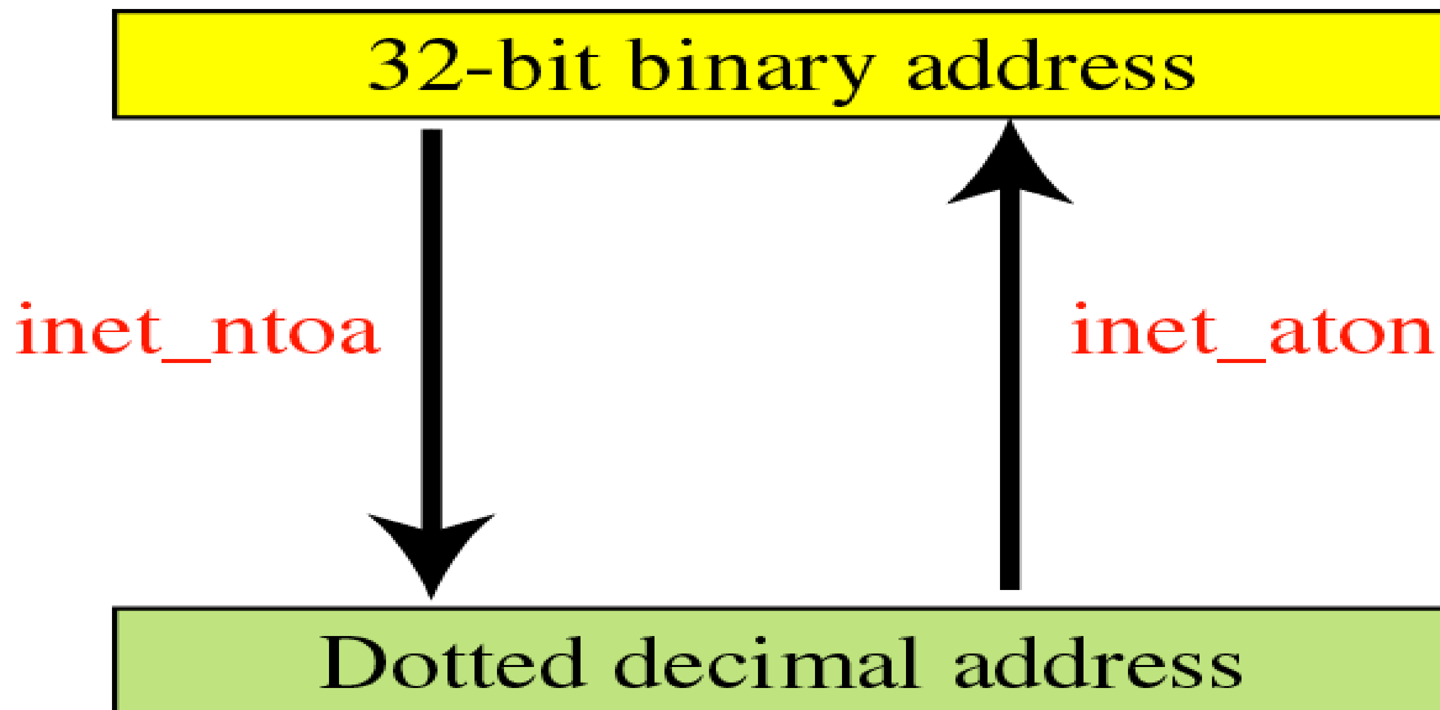



```
u_short   htons  ( u_short   host_short ) ;

u_short   ntohs  ( u_short   network_short ) ;

u_long    htonl  ( u_long    host_long ) ;

u_long    ntohl  ( u_long    network_long ) ;
```

Address Transformation

int inet_aton (const char *strptr , struct in_addr *addrptr) ;

char *inet_ntoa (struct in_addr inaddr) ;



Generic Socket Address structure

- A Socket address structure must be passed by reference
- socket function that takes one of these pointers as an argument must deal with socket address structures from any of the supported protocol families.
- How to declare the type of pointer
- Soln : void *
- Define Generic socket address structure

<sys/socket.h>

Generic Socket Address structure

```
Struct sockaddr
 {
        uint8_tsa_len;  sa_family_t
                                        sa_family;

        char sa_data[14];/* protocol specific  address*/
   };
```

From an application programmer's point

IPv6 Socket Structure

```
Struct in6_addr{
      uint8_t     s6_addr[16];                 /*128bit IPv6 address*/
   };                                          /*network byte ordered*/
#define SIN6_LEN                  /* required for compile-time tests */
struct sockaddr_in6 {
   uint8_t            sin6_len;                /* length of structure(24) */
   sa_family_t        sin6_family;             /* AF_INET6*/
   in_port_t          sin6_port;               /* Transport layer port# */
                                               /*network byte ordered*/
   uint32_t           sin6_flowinfo;           /* priority & flow label */
                                               /*network byte ordered*/
   struct
           in6_addr   sin6_addr;               /* IPv6 address */
                                               /*network byte ordered*/
}; /* included in <netinet/in.h> */
```

.

connect()

int connect(int sockfd, const struct sockaddr *servaddr, socklen_t addrlen);

- sockfd is socket descriptor from socket()
- servaddr is a pointer to a structure with:
  - port number and IP address
  - must be specified (unlike bind())
- addrlen is length of structure
- client doesn’t need bind()
  - OS will pick ephemeral port
- returns socket descriptor if ok, -1 on error

Sending and Receiving

int recv(int sockfd, void *buff, size_t mbytes, int flags);

int send(int sockfd, void *buff, size_t mbytes, int flags);

- Same as read() and write() but for flags
  - MSG_DONTWAIT (this send non-blocking)
  - MSG_OOB (out of band data, 1 byte sent ahead)
  - MSG_PEEK (look, but don't remove)
  - MSG_WAITALL (don't give me less than max)
  - MSG_DONTROUTE (bypass routing table)

Procedures That Implement The Socket API

Creating and Deleting Sockets

- fd=**socket**(protofamily, type, protocol)
  Creates a new socket. Returns a file descriptor (fd). Must specify:
  - the protocol family (e.g. TCP/IP)
  - the type of service (e.g. STREAM or DGRAM)
  - the protocol (e.g. TCP or UDP)
- **close**(fd)
  Deletes socket.
  For connected STREAM sockets, sends EOF to close connection.

Procedures That Implement The Socket API

Putting Servers "on the Air"

- **bind**(fd)
  Used by server to establish port to listen on.
  When server has >1 IP addrs, can specify "ANY", or a specific one
- **listen** (fd, queuesize)
  Used by connection-oriented servers only, to put server "on the air"
  Queuesize parameter: how many pending connections can be waiting
- afd = **accept** (lfd, caddress, caddresslen)
  Used by connection-oriented servers to accept one new connection

How Clients Communicate with Servers

- **connect** (fd, saddress, saddreslen)
  Used by connection-oriented clients to connect to server
  - There must already be a socket bound to a connection-oriented service on the fd
  - There must already be a listening socket on the server
  - You pass in the address (IP address, and port number) of the server.

  Used by connectionless clients to specify a “default send to address”
  - Subsequent “writes” or “sends” don’t have to specify a destination address

How Clients Communicate with Servers

- **send** (fd, data, length, flags)
  sendto (fd, data, length, flags, destaddress, addresslen)
  sendmsg (fd, msgstruct, flags)
  write (fd, data, length)
  Used to send data.
    - **send** requires a connection (or for UDP, default send address) be already established
    - **sendto** used when we need to specify the dest address (for UDP only)
    - sendmsg is an alternative version of sendto that uses a struct to pass parameters
    - write is the "normal" write function; can be used with both files and sockets
- **recv** (...) **recvfrom** (...) **recvmsg** (...) **read** (...)

System Calls

int sendto (int socket, char *message, int nbytes, int flags, struct sockaddr *dest, int dest_len);

Description

The sendto() function sends a message through connectionless-mode socket.The message will be sent to the address specified by dest.

The system call takes the following arguments:

Socket Specifies the socket descriptor.

Message Points to a buffer containing the message to be sent.

nbytes Specifies the size of the message in bytes.

System Calls

int recvfrom (int socket, char *message, int nbytes int flags, struct sockaddr *from, int *from_len);

Description

The recvfrom() function call receives a message from a connectionless-mode socket.

The recvfrom system call fills in the protocol specific address of who sent the data into *from* . The length of this address is also returned to the caller in *from_len.*

Connection Oriented Protocol

Server

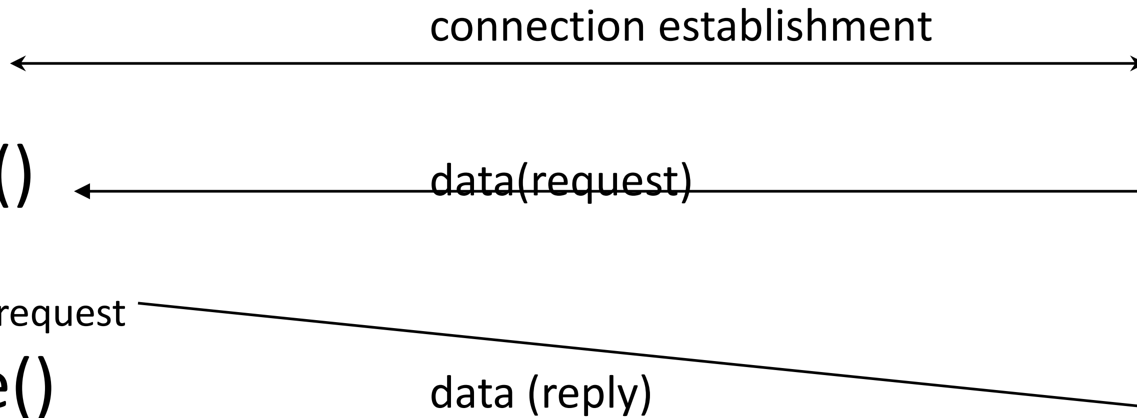
socket()

bind()

listen()

accept()

blocks until connection from client



read()

process request

write()

Client

socket()

connect()

write()

read()

Connectionless Protocol

Server

socket()

bind()

recvfrom()

Client

blocks until connection from client

socket()

bind()

data (request)

sendto()

process request

write()

data (reply)

recvfrom()

Comparison of ipc mechanisms

1. Pipes: It is related data only send from one pipe output is giving to another pipe input to share resources pipe are used

2. Message Queues: message queues are unrelated process are also communicate with message queues.

3. Sockets: it is used to communicate clients and server 193 with socket system calls connection oriented and connection less also

4. Message Queues: Any number of processes can read/write from/to the queue.

5. Shared Memory: Accessing Shared memory is faster than any other IPC mechanism.
6. Semaphore: Semaphores are used for process synchronization.