LINUX INTERNALS

Course code:AIT005

B.Tech VI Semester

Regulation: IARE R-16

BY
Mr. A Krishna Chaitanya

Assistant Professors
Mr. D Rahul

DEPARTMENT OF INFORMATION TECHNOLOGY
INSTITUTE OF AERONAUTICAL ENGINEERING
(Autonomous)
DUNDIGAL, HYDERABAD - 500 043

CO’s	Course outcomes
CO1	Understand the basic commands of linux operating system and can write shell scripts.
CO2	Create file systems and directories and operate those using programs.
CO 3	Understand the processes background and fore ground by process and signals system calls.
CO 4	Create shared memory segments, pipes, message queues and can exercise inter process communication.
CO 5	Create sockets and semaphores to interact between process of different system.

UNIT-I

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 and text processing operations
CLO 3	Develop shell scripts to perform more complex tasks in shell programming environment.

Operating System :

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

Why use UNIX:

- Multi-tasking / Multi-user Networking Capability Graphical (With Command Line) Easy To Program

- Portable (Pc's, Mainframes, Super-computers)

File: s 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:

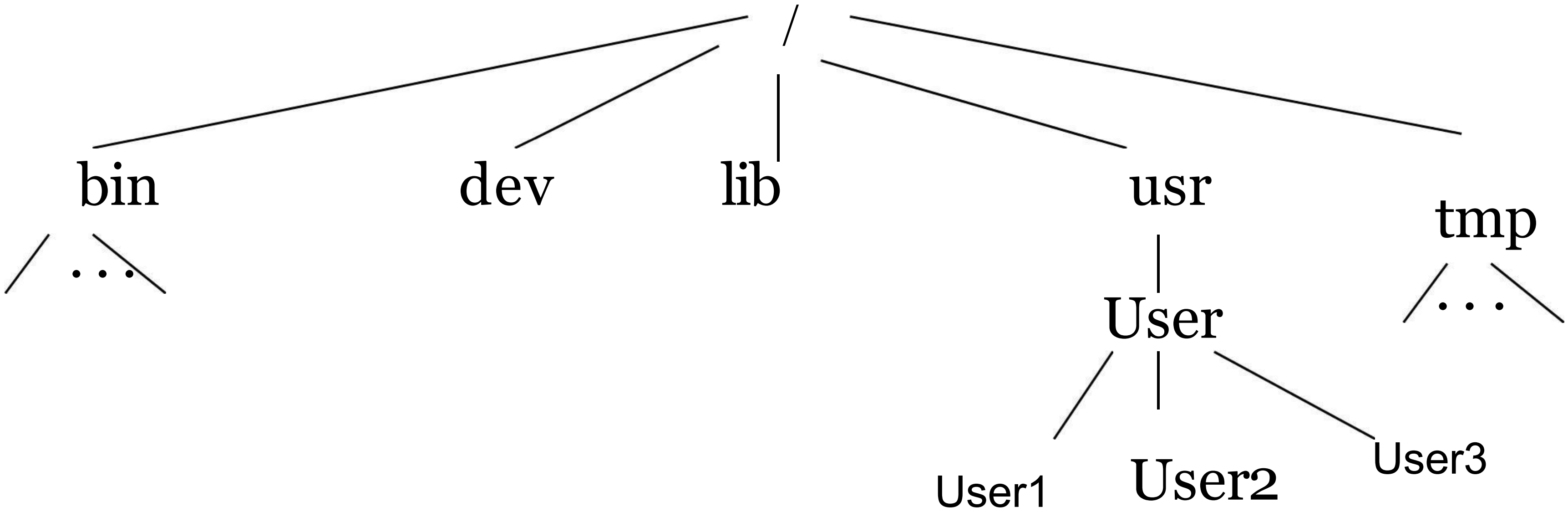
1.file name.

2.a unique identification number for the file or directory.

Device file: It represents the device or peripheral.

The Architecture of UNIX

A simplified UNIX directory/file system:



/

bin

. . .

dev

lib

usr

User

User1

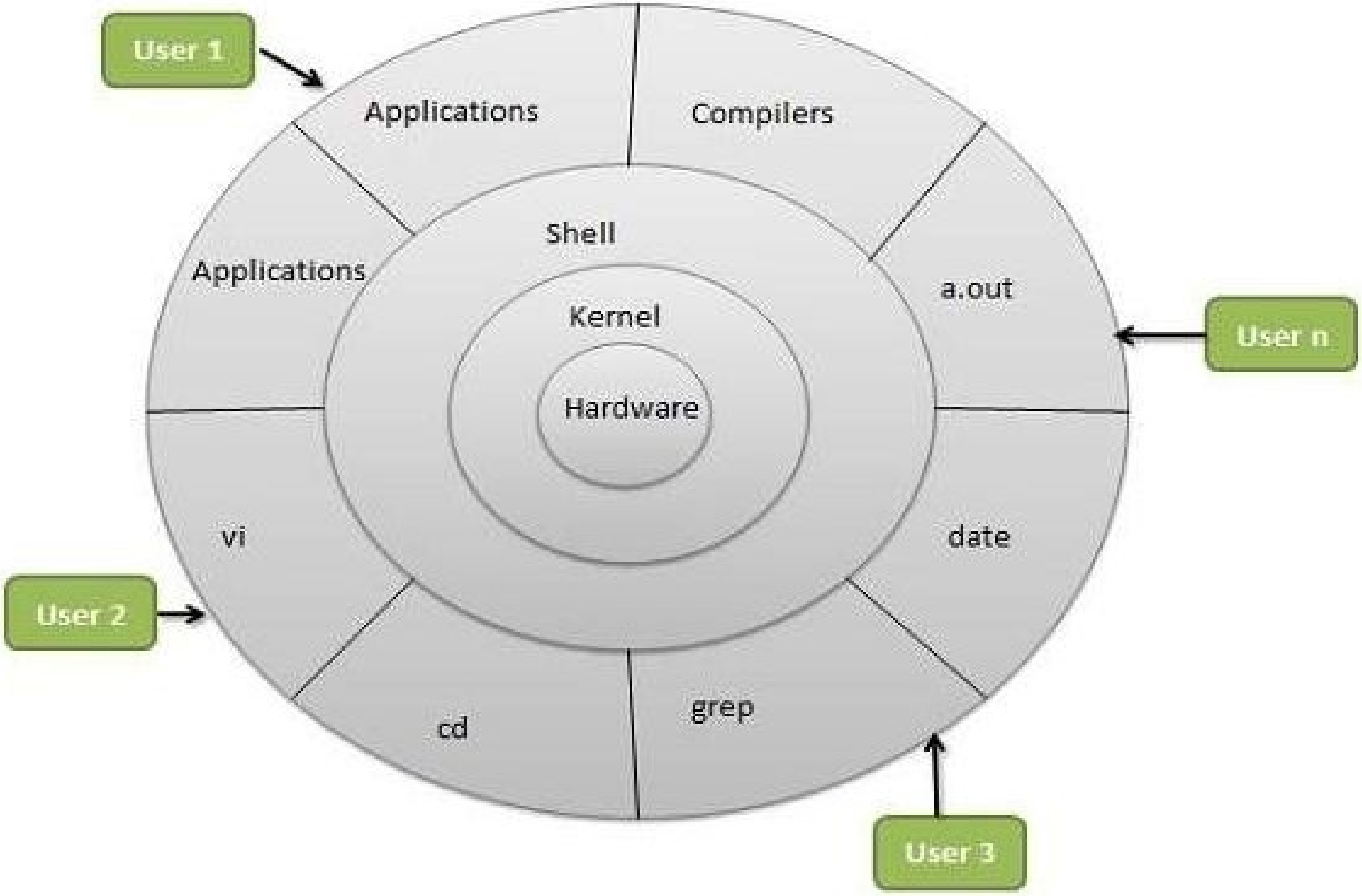
User2

User3

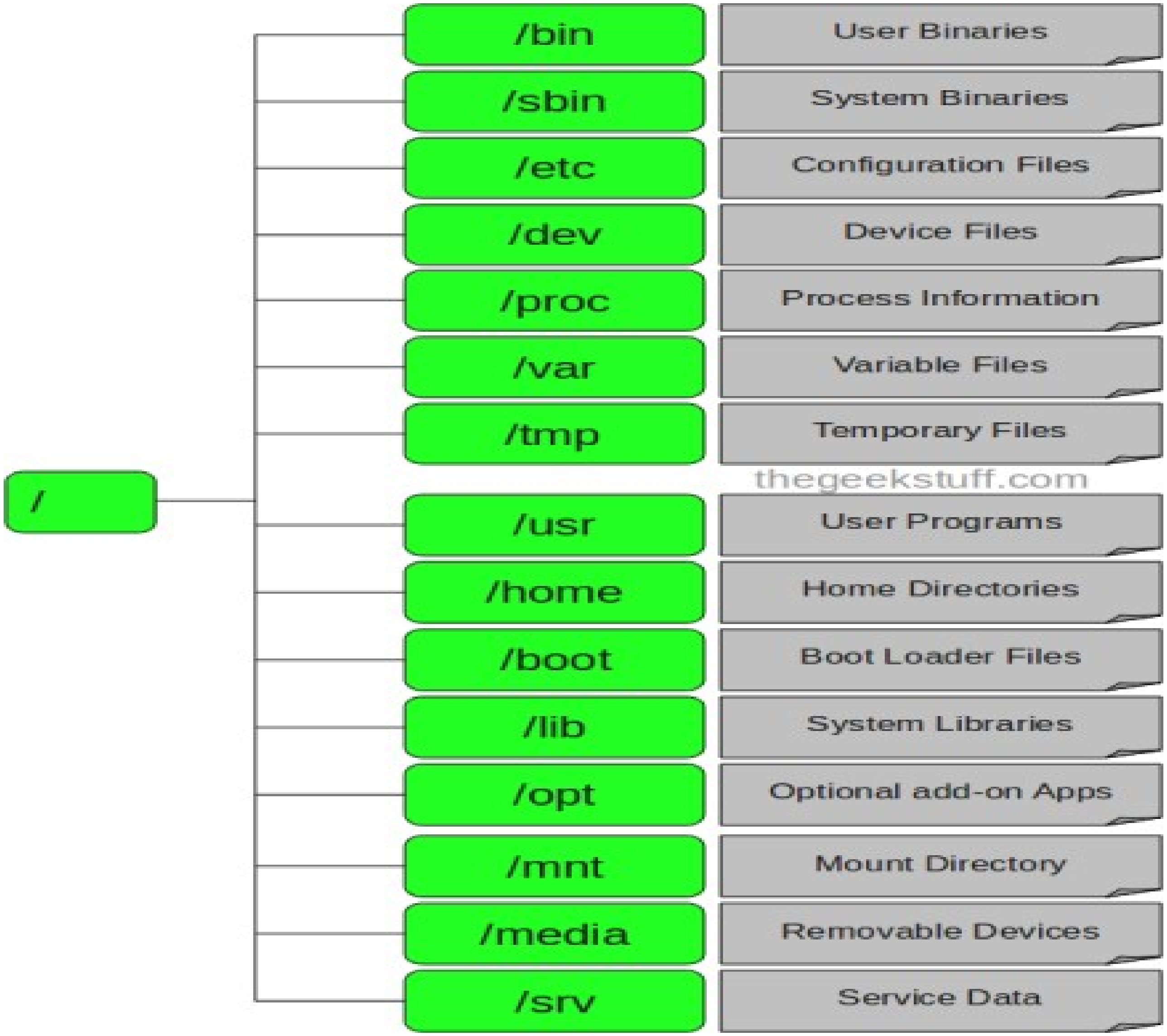
tmp

. . .

Architecture



- Linux System Architecture is consists of following layers
  - **Hardware layer** - Hardware consists of all peripheral devices (RAM/ HDD/ CPU etc).
  - **Kernel** - Core component of Operating System, interacts directly with hardware, provides low level services to upper layer components.
  - **Shell** - An interface to kernel, hiding complexity of kernel's functions from users. Takes commands from user and executes kernel's functions.
  - **Utilities** - Utility programs giving user most of the functionalities of an operating systems.



/

/bin	User Binaries
/sbin	System Binaries
/etc	Configuration Files
/dev	Device Files
/proc	Process Information
/var	Variable Files
/tmp	Temporary Files
/usr	User Programs
/home	Home Directories
/boot	Boot Loader Files
/lib	System Libraries
/opt	Optional add-on Apps
/mnt	Mount Directory
/media	Removable Devices
/srv	Service Data

thegeekstuff.com

/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.

Editor:

vi is a full screen text editor. It was created by Bill Joy.

Bram Moolenaor improved it and called it vim (vi improved).

Invoking vi:

$Vi file name

Modes of Operation

Vi has 3 mode of operation.

1.Command mode: In this mode all the keys pressed by the user are interpreted as commands. It may perform some actions like move cursor, save, delete text, quit vi, etc.

2.Input/Insert mode: used for inserting text.

– start by typing i; finish with ESC

MODES OF INSERTING TEXT

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
`j`	down one
`k`	up one
`l`	right one
`w`	move forward one
`b`	word back one word

Finishing a vi Session

Get to command mode (press ESCs)

`ZZ`	save changes to the file and quit (no RETURN)
`:q!`	quit without saving (press RETURN)

:wq! Saves the file & quit.

Inserting Text

Move to insertion point

Switch to input mode: i

Start typing; BACKSPACE or DELETE for deletion

ESCfinish; back in command mode

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

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.

Copying Files

Cp turns to interactive when –i option is used & destination file also exists.

```
$cp -i file1 file2  overwrite file2
(yes/no)?
```

Y at this prompt overwrites the file.

mv (Moving and Renaming Files)

Used to rename the files/directories.

```
$mv test sample
```

Here test is renamed as sample.

ln (link):

Used to create links (both soft & hard links).

It creates the alias & increase the link count by one.

$ln file1 file2

ln won‘t work if the destination file also exists.

rm (Deleting Files and Directories)

– To delete or remove a file, you use the ―rm‖ command. For example,

```
$rm my. listing
```

will delete ―my.listing‖.

With –i option removes the files interactively.

```
$rm –i file1
```

With –r option recursively removes directories.

```
$rm –r dir1
```

mkdir: used to create one or more directories.

$mkdir book

Creates the directory named book.

$mkdir dbs doc dmc Creates

three directories.

rmdir (remove directories): Removes empty directories.

$rmdir book

removes directory named book if it is empty.

$rmdir dbs doc dmc

Removes 3 directories.

find: It recursively examines a directory tree to look for matching some criteria and then takes some action on the selected files.

Syntax:

find path_list selection_criteria action

To locate all files named a. out use

```
$find / -name a. out –print
```

‗/‘ indicates search should start from root directory.

To locate all c files in current directory

```
$find . -name —*c‖–print
```

To find all files begin with an uppercase letter use

```
$find . –name ‗[A-Z]*‘ –print
```

Find operators:

Find uses 3 operators

!,-a ,-o

Security by file permissions:

chmod Command:

chmod command allows you to alter / Change access rights to files and directories.

SYNTAX:

The Syntax is

chmod [options] [MODE] FileName

File Permission

#	File Permission
0	none
1	execute only
2	write only
3	write and execute
4	read only
5	read and execute
6	read and write
7	set all permissions

- **OPTIONS:**

- -c Displays names of only those files whose permissions
- are being changed
- -f Suppress most error messages
- -R Change files and directories recursively
- -v Output version information and exit.

Abbreviations used by chmod:

Category	operation
u-user	+-assign permission
g-group	--remove permission
o-others	=-assigns absolute permission
a-all	

permissions

r-read permission

w-write permission

x-execute permission

Absolute assignment:

Absolute assignment by chmod is done with the = operator. Unlike the + or – operator s, it assigns only those permissions that are specified along with it and removes other permissions.

If u want to assign only read permission to all three categories and remove all other permissions from the file small use

```
chmod g-wx,o-x small
```

Or simply use = operator in any of the following ways.

```
chmod   ugo=r   small
chmod     a=r     small
Chmod =r small
```

The octal notation:

Chmod also takes a numeric argument that describes both the category and the permission. The notation uses octal numbers. Each permission is assigned a number like

read permission-4, write permission- 2, execute permission-1

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.

PROCESS UTILITIES

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.

$who

who am i: Show you the owner of this account

$whoami

w: Tell you who is logging in and doing what!

$w

Finger: Displays the information about the users.

$finger ***user***

Find out the personal information of a user

$finger ***name***

Try to find the person‘s info. by his/her name

finger ***email-address***

Try to find the person‘s info across the network

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.

Mount uses an option to specify the type of file system.

To mount a file system on the /oracle directory on Linux system use

```
$mount –t ext2 /dev/hda3 /oracle
$mount –t iso9660 /dev/cdrom /mnt
/cdrom $mount –t vfat /dev/hda1 /msdos
$mount –t msdos /dev/fd0 /floppy
```

Umount: unmounting file systems

Unmounting is achieved with the umount command. which requires either file system name or the mount point as argument.

```
$umount /oracle
```

```
$umount /dev/hda3
```

Unmounting a file system is not possible if the file is opened.

ulimit: user limit

It contains a value which signifies the largest file that can be created by the user in the file system.

When used by itself it displays the current setting.

```
$ulimit
unlimited
```

User can also set the ulimit value by using $ulimit 10

unmask:

When u create files and directories, the default permissions that are assigned to them depend on the system‘s default setting. Actually this default is transformed

By subtracting the user mask from it to remove one or more permissions.
This value is evaluated by umask
without arguments.

```
$umask
022
```

Network Utilities

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
200   type   set   to   I
ftp>put penguin. gif
```

To copy multiple files use mput.

```
ftp>mput t*.sql
```

Downloading files: get & mget To download the files from remote machine use get & mget.

```
ftp>get   ls-lR.gz
ftp>_
```

NETWORK COMMANDS

telnet: Remote login

If u have an account on the host in a local network (or on internet),u can use this with the host name or the ip address as argument.

```
$telnet Saturn
Trying to 192.168.0.1…
Connected to Saturn
```

Login:----

Password:-----

U can quit telnet by using exit command.

telnet prompt:

When telnet used without Ip address the system displays a telnet> prompt . U can invoke a login session from here with open.

```
telnet>    open    192.168.0.8
Trying     to     192.168.0.8…
Connected to 192.168.0.8
```

rlogin: remote login without password

rlogin is the Berkley's implementation of the remote login facility.

U can log on to ur own identical remote account without using either the user name or password.

$rlogin   Jupiter

Last login :….

rlogin is terminated with ctrl+d or exit or logout.

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         or
$tail -3 file
```

We can also address the lines from the beginning of the file instead of the end.

The + count allows to do that.

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.

```
$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.

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)

File names where pattern occurs (-l)

Syntax:

grep option pattern filename(s)

egrep: extended grep

egrep extended set includes 2 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
```

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

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.

User can use tee to save the output of the who command in a file and display it as well:

```
$who |tee user.lst
```

The tee command reads standard input, writes its content to standard output and simultaneously copies it into the specified file or files.

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.

cmp: comparing two files

The two files are compared byte by byte and the location of the first mismatch is echoed to the screen using cmp.

cmp when invoked without options it does not bother about possible subsequent mismatches.

```
$cmp group1 group2
```

If two files are identical cmp display s no message, but simply returns the $ prompt.

diff: converting one file to another

diff takes different approach to displaying the differences.

It tells u which lines in one file have to be changed to make two files identical.

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

BACKUP UTILITIES

tar: the tape archive program

Tar doesn‘t normally write to the standard output but creates an archive in the media. Tar accepts file and directory names as arguments.

It operates recursively.

It can create several versions of same file in a single archive.

It can append to an archive without overwriting the entire archive.

-c option is used to copy files to backup device.

```
$tar –cvf /dev/rdsk/foq18dt
 /home/sales/sql/*.sql
```

The verbose option (-v) shows the no. of blocks used by each file.

Files are restored with the –x (extract) key option. when no file or directory name is specified it restores all files from the backup device.

cpio: copy input-output

Cpio command copies files to and from a backup device. It uses standard input to take the list of filenames.

It then copies them with their contents and headers into stream which can be redirected to a file or a device.

Cpio can be used with redirection and piping.

Cpio uses two options-o (output) and –i (input) either of which must be there in the command line.

STREAM EDITOR

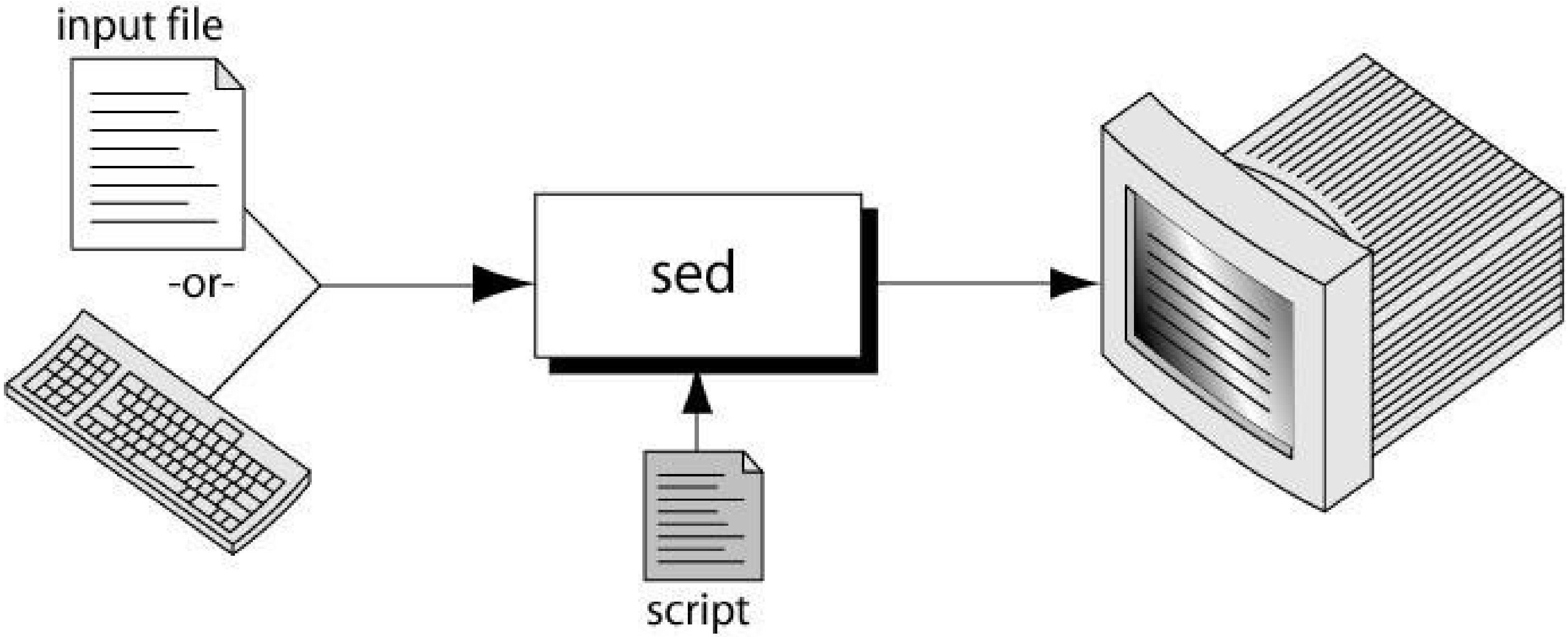
SED:

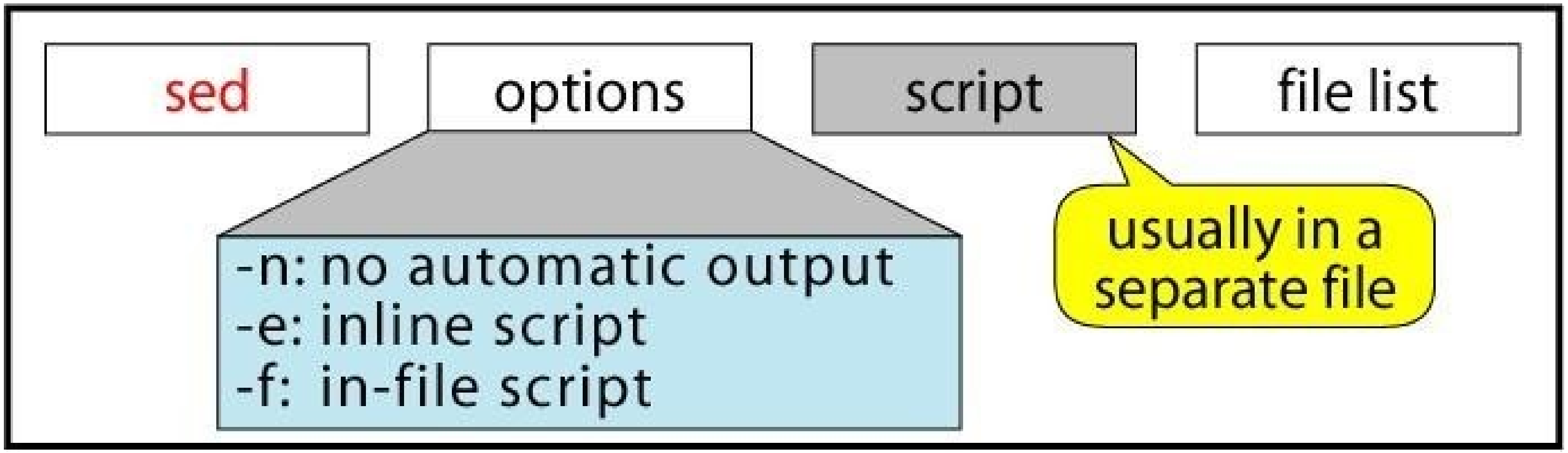
What is sed?

- A non-interactive stream editor
- Interprets sed instructions and performs actions

Use sed to:

- Automatically perform edits on file(s)
- Simplify doing the same edits on multiple files
- Write conversion programs





sed

options

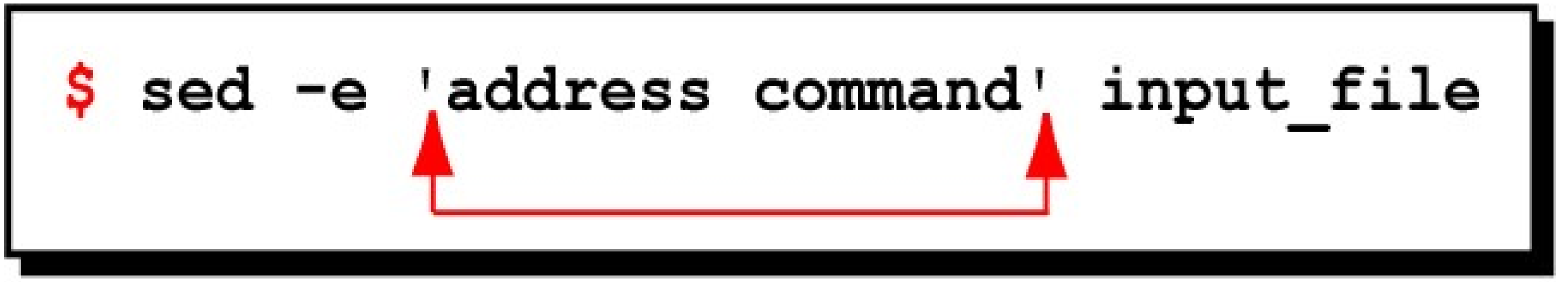
script

file list

-n: no automatic output
-e: inline script
-f: in-file script

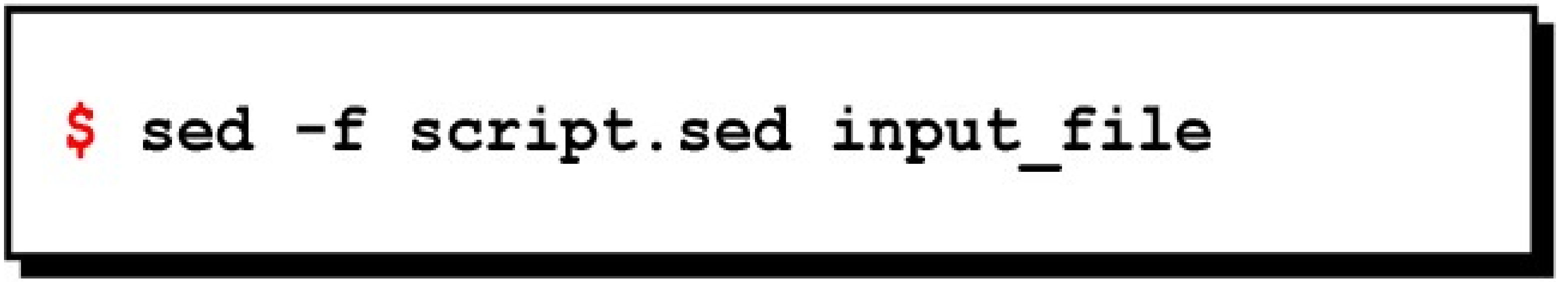
usually in a separate file

sed Command Syntax(Sed Scripts):



```
$ sed -e 'address command' input_file
```

(a) Inline Script

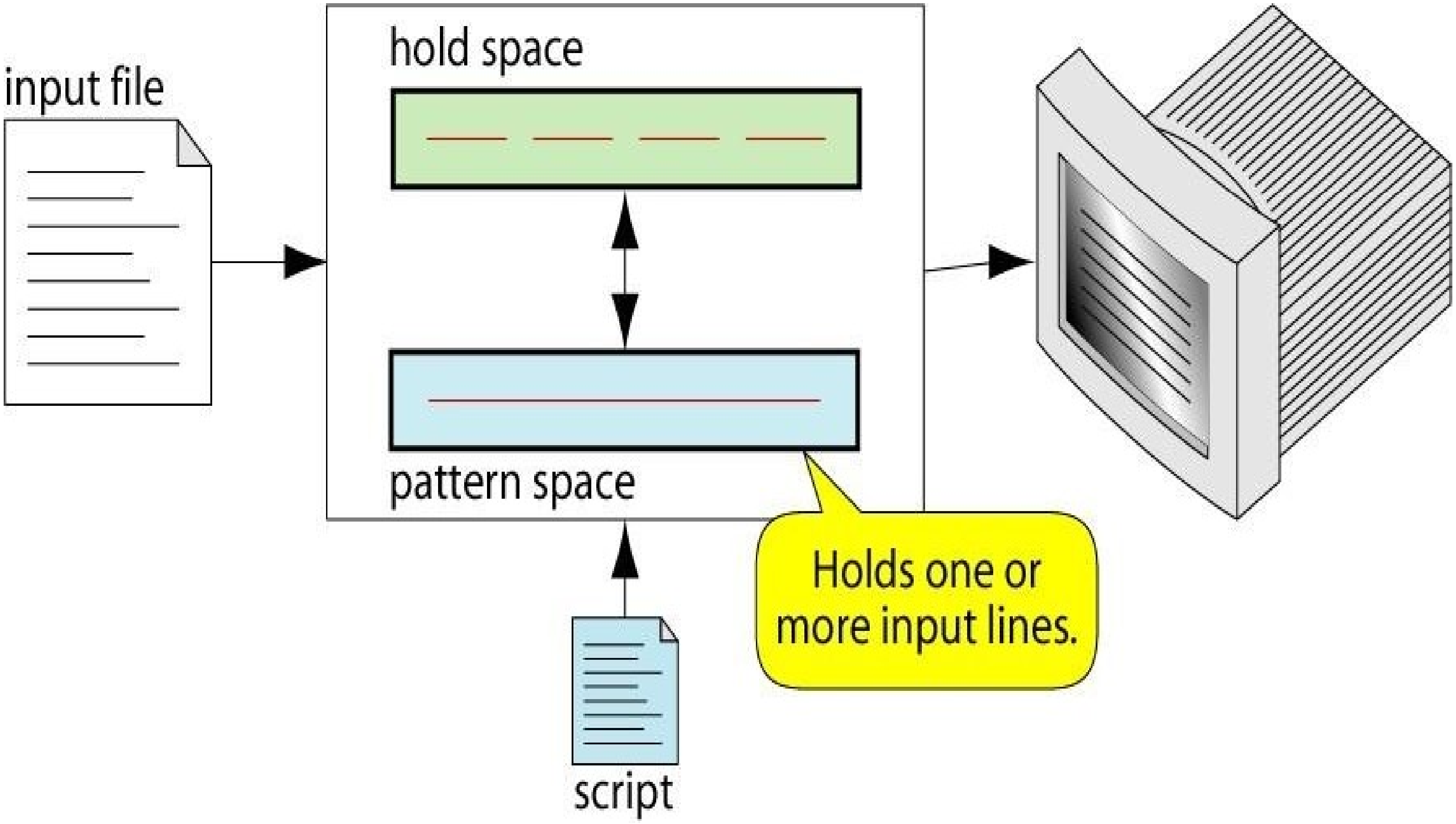


```
$ sed -f script.sed input_file
```

(b) Script File

Sed Operation

How Does sed Work?



- 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

Examples:

- show only line 3

sed -n -e '3 p' input-file

- show only last line

sed -n -e '$ p' input-file

- substitute “endif” with “fi” on line 10

sed -e '10 s/endif/fi/' input-file

- special: dollar sign ($) denotes last line of input file

Set-of-Lines Address

- use regular expression to match lines
  - written between two slashes
  - process only lines that match
  - may match several lines
  - lines may or may not be consecutives

Examples:

```
sed -e ‘/key/ s/more/other/’ input-file
sed -n -e ‘/r..t/ p’ input-file
```

Range Address

- Defines a set of consecutive lines
- Format:

  start-addr,end-addr (inclusive)

  Examples:

  10,50 line-number,line-number 10,/R.E/
  line-number,/RegExp//R.E./,10
  /RegExp/,line-number/R.E./,/R.E/
  /RegExp/,/RegExp/

  Example:

  Range Address

```
% sed -n -e ‘/^BEGIN$/,/^END$/p’ input-file
```

- Print lines between BEGIN and END, inclusive

```
BEGIN
Line 1 of input Line 2 of input Line3 of input
END
Line 4 of input Line 5 of input
```

Nested Address

- Nested address contained within another address
- Example:

print blank lines between line 20 and 30

```
20,30{
/^$/ p
}
```

Address with !

- address with an exclamation point (!):

instruction will be applied to all lines that do not match the address

Example:

print lines that do not contain “obsolete”

sed -e ‘/obsolete/!p’ input-file

awk command:

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 fields

```
awk –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.

When executed, the program reads from the standard input and writes to the standard output. It takes as parameters two sets of characters, and replaces occurrences of the characters in the first set with the corresponding elements from the other set.

Examples:

```
$ tr "[a-z]" "z[a-y]" < computer.txt
$tr –d ‘|/’ <shortlist | head -3
$tr ‘|’ ‘\012’ <shortlist | head -6
$tr ‘|/’ ‘~-’ < shortlist |head -3
```

pg:

pg is used to display the output of a file in page by page.

```
$pg file1
```

AWK ARRAY:

1.awk allows one-dimensional arrays to store strings or numbers
2.index can be number or string
3.array need not be declared

- its size
- its elements

4.array elements are created when first used
5.initialized to 0 or “”
Arrays in awk
Syntax:
arrayName[index] = value
Examples:
list[1] = "one" list[2] = "three"
list["other"] = "oh my !"

Awk builtin split functions

split(string, array, fieldsep)

- divides string into pieces separated by fieldsep, and stores the pieces in array
- if the fieldsep is omitted, the value of FS is used.

Example:

split("auto-da-fe", a, "-")

sets the contents of the array a as follows:

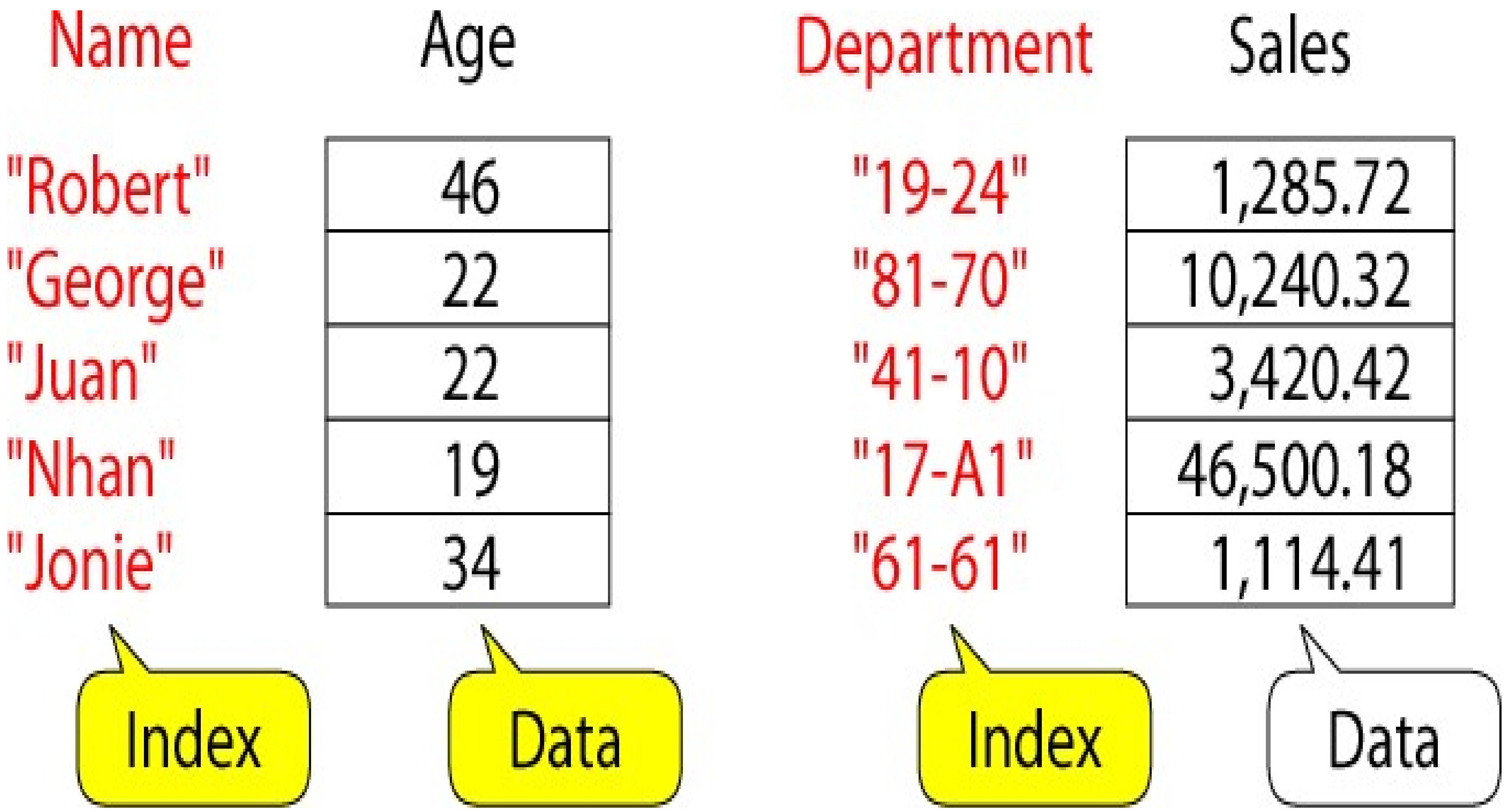
a[1] = "auto"

a[2] = "da"

a[3] = "fe"

Associative Arrays

1.awk arrays can use string as index

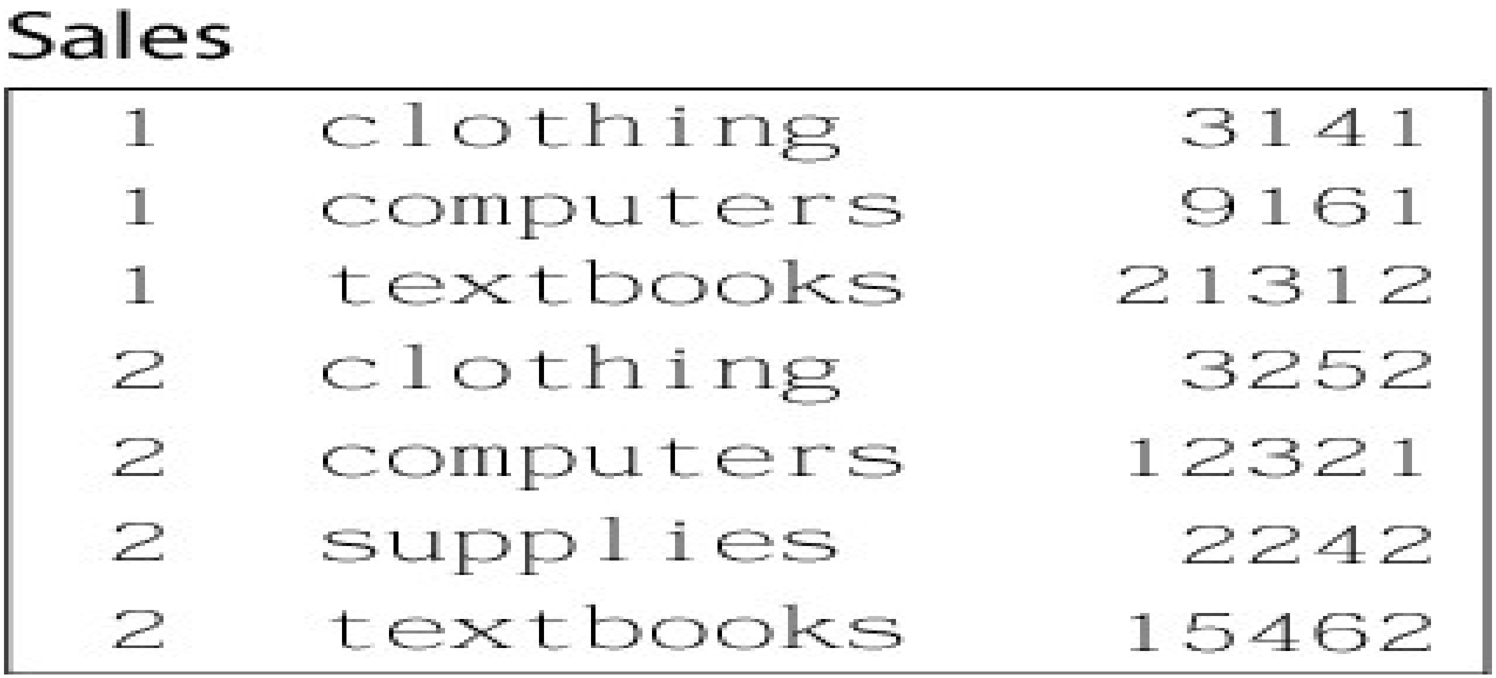


Name	Age
"Robert"	46
"George"	22
"Juan"	22
"Nhan"	19
"Jonie"	34

Department	Sales
"19-24"	1,285.72
"81-70"	10,240.32
"41-10"	3,420.42
"17-A1"	46,500.18
"61-61"	1,114.41

Example: process sales data

1.input file:

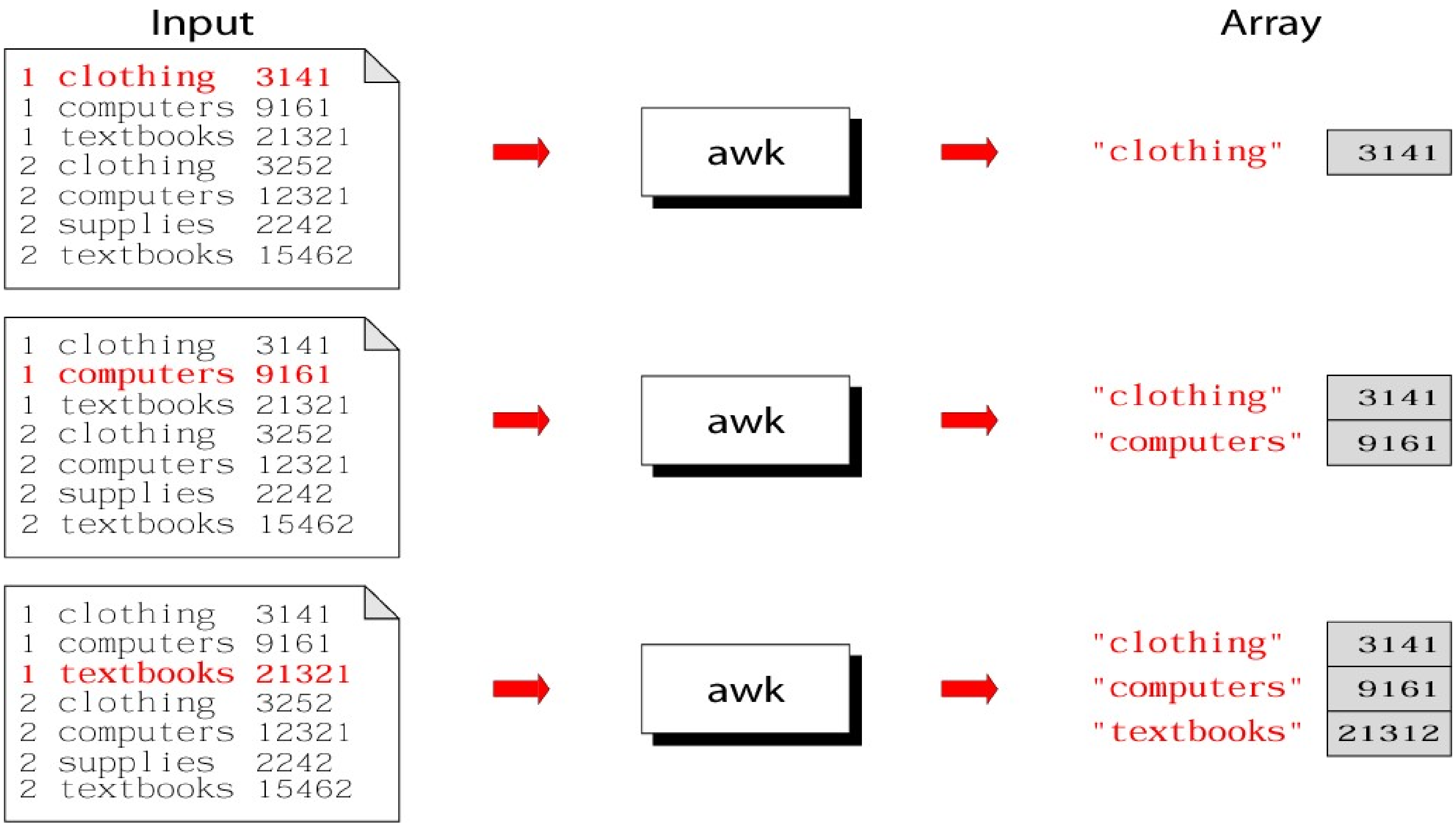


Sales

1	clothing	3141
1	computers	9161
1	textbooks	21312
2	clothing	3252
2	computers	12321
2	supplies	2242
2	textbooks	15462

- output:

summary of category sales Illustration: process each input line



Input

Array

```
1 clothing   3141
1 computers  9161
1 textbooks  21321
2 clothing   3252
2 computers  12321
2 supplies   2242
2 textbooks  15462
```

awk

"clothing" 3141

```
1 clothing   3141
1 computers  9161
1 textbooks  21321
2 clothing   3252
2 computers  12321
2 supplies   2242
2 textbooks  15462
```

awk

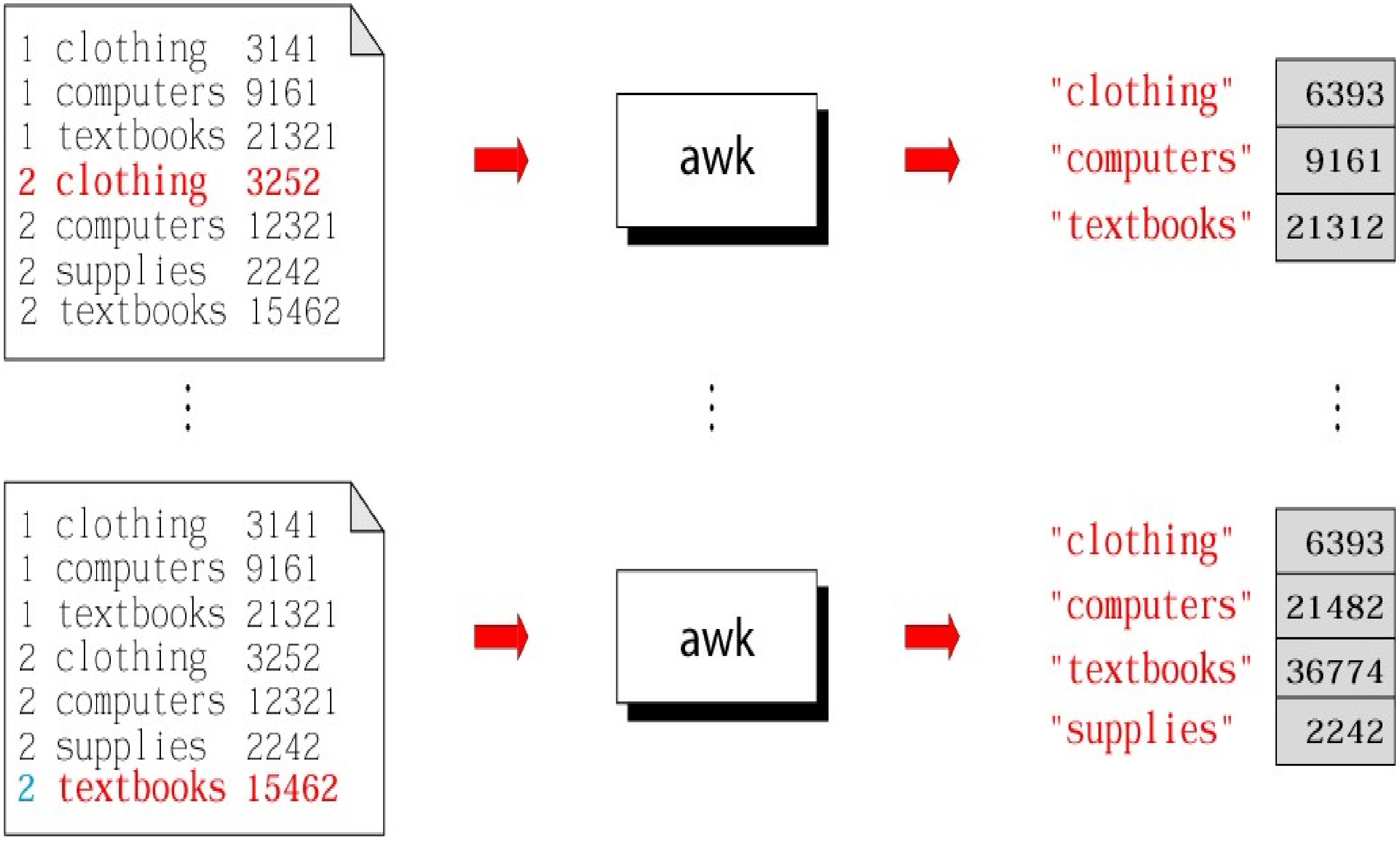
"clothing" 3141
"computers" 9161

```
1 clothing   3141
1 computers  9161
1 textbooks  21321
2 clothing   3252
2 computers  12321
2 supplies   2242
2 textbooks  15462
```

awk

"clothing" 3141
"computers" 9161
"textbooks" 21312

Illustration: process each input line



```
1 clothing   3141
1 computers  9161
1 textbooks  21321
2 clothing   3252
2 computers  12321
2 supplies   2242
2 textbooks  15462
```

awk

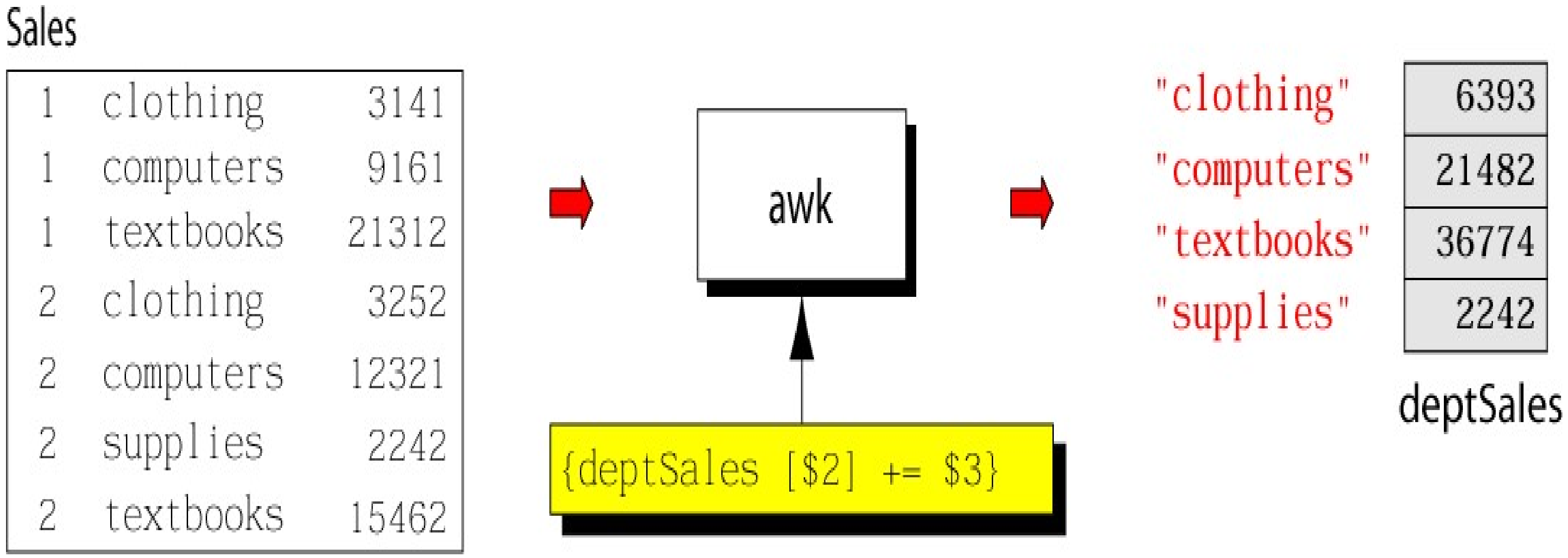
"clothing"	6393
"computers"	9161
"textbooks"	21312

```
1 clothing   3141
1 computers  9161
1 textbooks  21321
2 clothing   3252
2 computers  12321
2 supplies   2242
2 textbooks  15462
```

awk

"clothing"	6393
"computers"	21482
"textbooks"	36774
"supplies"	2242

Summary: awk program



Sales

1	clothing	3141
1	computers	9161
1	textbooks	21312
2	clothing	3252
2	computers	12321
2	supplies	2242
2	textbooks	15462

awk

```
{deptSales [$2] += $3}
```

"clothing"	6393
"computers"	21482
"textbooks"	36774
"supplies"	2242

deptSales

Example: complete program

```
% cat sales.awk
{
deptSales[$2] += $3
}
END {
for (x in deptSales)
print x, deptSales[x]
}
% awk –f sales.awk sales
```

awk built in functions

tolower(string)

- returns a copy of string, with each upper-case character converted to lower-case.
- Nonalphabetic characters are left unchanged.

Example: tolower("MiXeD cAsE 123")
returns "mixed case 123“

toupper(string)

returns a copy of string, with each lower-case character converted to upper-case.

Working with Bourne 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)**

When we issue a command the shell is the first agency to acquire the information. It accepts and interprets user requests. The shell examines &rebuilds the commands &leaves the execution work to kernel. The kernel handles the h/w on behalf of these commands &all processes in the system.

The shell is generally sleeping. It wakes up when an input is keyed in at the prompt. This input is actually input to the program that represents the shell.

Shell Responsibilities

Program Execution

Variable and Filename Substitution

I/O Redirection

Pipeline Hookup

Environment Control

Interpreted Programming Language

Program Execution:

The shell is responsible for the execution of all programs that you 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.

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 >>.

Pipeline Hookup:

Just as the shell scans the command line looking for redirection characters, it also looks for the pipe character |. For each such character that it finds, it connects the standard output from the command preceding the | to the standard input of the one following the |. It then initiates

Environment Control:

The shell provides certain commands that let you customize your environment. Your environment includes home directory, the characters that the shell displays to prompt you to type in a command, and a list of the directories to be searched whenever you request that a program be executed.

INTERPRETED PROGRAMMING LANGUAGE:

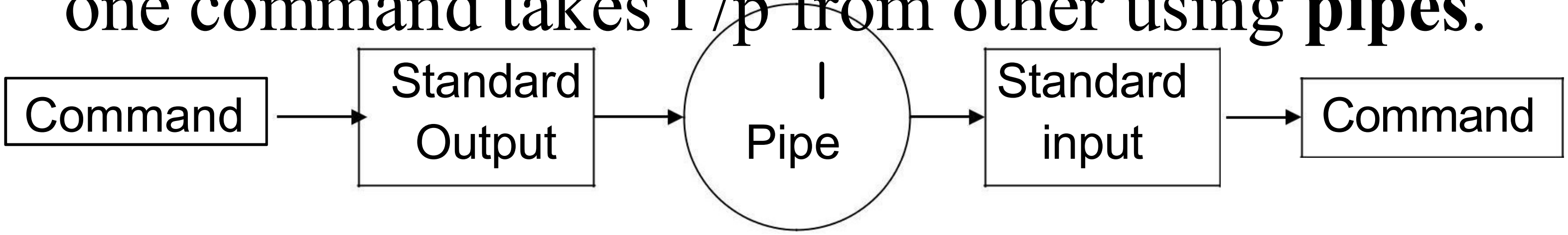
The shell has its own built-in programming language. This language is interpreted, meaning that the shell analyzes each statement in the language one line at a time and then executes it. This differs from programming languages such as C and FORTRAN, in which the programming statements are typically compiled into a machine- executable form before they are executed.

Programs developed in interpreted programming languages are typically easier to debug and modify than compiled ones. However, they usually take much longer to execute than their

compiled equivalents.

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

This method of using two commands in sequence has certain disadvantages.

1. The process is slow.
2. An intermediate file is required that has to be removed after the command has completed its run.
3. When handling large files, temporary files can build up easily &eat up disk space in no time.

Instead of using two separate commands, the shell can use a special operator as the connector of two commands-the pipe(|).

```
$who | wc –l
```

Here who is said to be piped to wc.

When a sequence of commands is combined together in this way a pipeline is said to be formed.

To count no. of files in current directory

```
$ls | wc –l
```

There‘s no restriction on the no. of commands u can use a pipeline.

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.

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.

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
```

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 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 a nonexistent file produces the error stream.

```
$cat bar
Cat: cannot open bar :no such file  or directory
```

The standard error stream can also be redirected to a file.

$cat bar 2> errorfile

Here 2 is the file descriptor 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.

Example:

```
mail Juliet << MARK
Ur pgm for printing the invoices has been executed on `date`. Check the print queue
The updated file is known as
$flname MARK
```

The shell treats every line followed by three lines of data and a delimited by MARK as input to the command. Juliet at other end will only see the three lines of message text, the word MARK itself doesn‘t show up.

SHELL METACHARACTERS:

The shell consists of large no. of metacharacters. 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:

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:

-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?‘
```

――(pair of double quotes)-negates the special properties of all enclosed characters except $,`,\ .

Eg:

$echo ―today date is $date‖ or

$echo ―today date 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:

$$- 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‘

ENVIRONMENTAL VARIABLES

They are initialized when the shell script starts and normally capitalized to distinguish them from user-defined variables in scripts

To display all variables in the local shell and their values, type the **set** command.

The **unset** command removes the variable from the current shell and sub shell

Environment Variables	Description
$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

PARAMETER VARIABLES

Parameter Variable	Description
`$1, $2, ….`	The parameters given to the script
`$*`	A list of all the parameters separated by the first character of IFS
`$@`	A list of all the parameters that doesn’t use the IFS environment variable

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….

Exit status of a command:

Every command returns a value after execution .This value is called the exit status or return value of a command.

This value is said to be true if the command executes successfully and false if it fails.

There is special parameter used by the shell it is the $?. It stores the exit status of a command.

exit:

The exit statement is used to prematurely terminate a program. When this statement is encountered in a script, execution is halted and control is returned to the calling program- in most cases the shell.

U don‘t need to place exit at the end of every shell script because the shell knows when script execution is complete.

set:

Set is used to produce the list of currently defined variables.

```
$set
```

Set is used to assign values to the positional parameters.

```
$set welcome to Unix
```

The do-nothing(:)Command

It is a null command.

In some older shell scripts, colon was used at the start of a line to introduce a comment, but

expr:

The **expr** command evaluates its arguments as an expression:

```
$ expr 8 + 6
14
$ x=`expr 12 / 4 `
$ echo $x
3
```

export:

There is a way to make the value of a variable known to a sub shell, and that's by exporting it with the export command.

The format of this command is

export variables

where variables is the list of variable names that you want exported. For any sub shells that get executed from that point on, the value of the exported variables will be passed down to the sub shell.

eval:

eval scans the command line twice before executing it. General form for eval is

```
eval    command-line
```

Eg:

```
$ cat last
eval echo \$$#
$  last  one  two  three
four four
```

${n}

If u supply more than nine arguments to a program, u cannot access the tenth and greater arguments with $10, $11, and so on.

${n} must be used. So to directly access argument 10, you must write

${10}

Shift command:

The shift command allows u to effectively left shift your positional parameters. If u execute the command

Shift

whatever was previously stored inside $2 will be assigned to $1, whatever was previously stored in $3 will be assigned to $2, and so on. The old value of $1 will be irretrievably lost.

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 commands fi

```
if command is successful
    then
    execute commands
fi
                      or
if command is successful
    then
    execute commands
elif command is successful
then..
else..
fi
```

FILE CONDITIONS

`-d file`	True if the file is a directory
`-e file`	True if the file exists
`-f file`	True if the file is a regular file
`-g file`	True if set-group-id is set on file
`-r file`	True if the file is readable
`-s file`	True if the file has non-zero size
`-u file`	True if set-user-id is set on file
`-w file`	True if the file is writeable
`-x file`	True if the file is executable

Example

```
$ mkdirtemp
$ if [ -f temp ];then
  echo "temp is a directory"  fi
```

ARITHMETIC COMPARISION

ep1 `-eq` *ep2*	True if *ep1* = *ep2*
ep1 `-ne` *ep2*	True if *ep1* ≠ *ep2*
ep1 `-gt` *ep2*	True if *ep1* > *ep2*
ep1 `-ge` *ep2*	True if *ep1* >= *ep2*
ep1 `-lt` *ep2*	True if *ep1* < *ep2*
ep1 `-le` *ep2*	True if *ep1* <= *ep2*
! *ep*	True if *ep* is false

Space is necessary

Example

```
$ x=5; y=7
$ if [ $x -lt $y ]; then
> echo "x is less than y"
> fi
```

Case conditional:

Case is a multi way branching. It matches strings with wild cards.

syntax: case

```
  expr in
pattern1) command1;;
Pattern2)    command1;;
pattern3)    command1;;
esac
```

While and until: looping

While statement repeatedly performs a set of instructions till the command returns a true exit status.

Syntax:

```
while condition is true
do
        commands
done
```

Until: while’s complement

The until statement complements the while construct in that the loop body here is executed repeatedly as long as the condition remains is false.

Syntax:

```
until condition is false
do
      commands
done
```

for loop:

Unlike while and until for doesn‘t test a condition but uses a list instead.

Syntax:

```
for variable in list
do
   commands
done
```

The list here comprises a series of character strings separated by whitespace.

Shell script examples

Example:

```
#!/bin/sh
echo "Is it morning? (Answer yes or  no)" read
timeofday
if [ $timeofday = "yes" ];  then echo
        "Good Morning"
else
        echo "Good afternoon"
fi  exit 0
```

elif - Doing further Checks

```
#!/bin/sh
echo "Is it morning? Please answer yes or  no" read
timeofday
if [ $timeofday = "yes" ];
        then echo "Good Morning"  elif [
$timeofday = "no" ];
        then echo "Good afternoon"
else
        echo "Wrong answer! Enter yes or no"  exit 1
fi  exit 0
```

Looping -- for

for *variable* in *values*
do
....
done

```
#!/bin/sh
for i in 1 2 3 4
5 do
    echo "Number: $i"
done
```

```
#!/bin/sh
for file in U N I  X do
      echo $i
done
```

case

```
case variable   in
        pattern [  | pattern]   . . . ) statements ; ;
        pattern [  | pattern]   . . . ) statements ; ;
        . . . .
 esac
```

```
#!/bin/sh
echo "Is it morning? Enter yes or no";read  timeofday case
"$timeofday" in
   yes | y | Yes | YES)  n* | N*        echo "Good Morning";;  echo
                     )                       "Good Afternoon";;
          ) echo "Sorry, answer not recognized"  echo "Please
               answer yes or no"  exit 1;;
esac
```

EXECUTING SHELL SCRIPT

U can execute a shell script by invoking its filename.

$filename

U can also use *sh* command which takes script name as argument.

$sh filename

PROBLEMS SOLVING APPROACHES IN UNIX:

Using single command:

If only one command is used for solving a problem then the command is known as single Unix command for solving a problem.

Eg:

```
$mv file1 file2
$cd /usr/bin
```

This is the simplest approach for solving a

Using compound commands:

When a single command is not sufficient to solve a problem, try to join the commands together.

Two approaches for this are:

Redirection

Piping

Redirection:

Unix commands are built-up in such a way that they can take the input from the keyboard, often called standard input and usually send their output to the screen, often called standard output. Commands also send error information to the screen.

We can also change these defaults by using a process called **redirection**.

There are three standard files. The shell sets up these three 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 i/p:

It has three sources

The keyboard, default source

A file using redirection with <

Another program using a pipeline.

Eg:

```
$wc    or $wc file1
$wc < file1 or
$cat file1 | wc
```

Standard o/p:

It has 3 destinations.

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.

Eg:

```
$cat file1
 $cat file1>file2
 $who | newfile
```

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 a nonexistent file produces the error stream.

```
$cat bab
Cat: cannot open bab :no such file or directory
```

The standard error stream can also be redirected to a file.

```
$cat bar 2> errorfile
```

Here 2 is the file descriptor for standard error file.

Each of the standard files has a number called a file descriptor, which is used for identification.

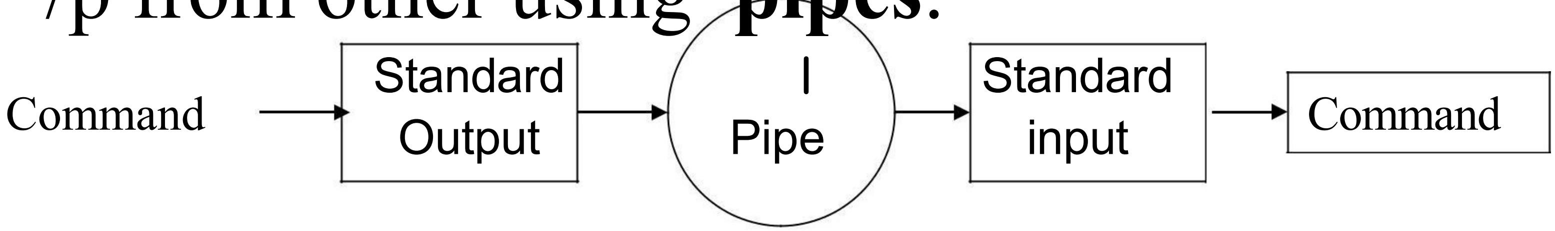
0—standard i/p

1---standard o/p

2---standard error

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**.



Command → Standard Output → | Pipe → Standard input → Command

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

Instead of using 2 separate commands we can combine them using pipes.

```
$who | wc –l
```

Here who is said to be piped to wc.

To count no. of files in current directory

```
$ls | wc –l
```

There‘s no restriction on the no. of commands u can use a pipeline.

SHELL SCRIPT

All the Unix commands can be entered in the command line itself. When a group of commands have to be executed regularly, they are better stored in a file.

All such files are called shell scripts or shell programs.

There‘s no restriction on the extension of the file name, though it is helpful to use .sh as the extension.

The following shell script is a sequence of commands:

```
$cat script.sh
echo the date today is `date`
echo ur shell is $SHELL
echo  ur  home  directory  is  $HOME
```

U can use vi to create this script.

To execute this script use

```
$sh script.sh            or
$script.sh
```

Positional parameters:

Shell script accept arguments in another manner-from the command line itself. This is the non interactive method of specifying the command line arguments.

When the arguments are specified with a shell script, they are assigned to certain special variables-rather positional parameters.

The first argument is read by the shell into the parameter $1, second argument into $2 and so on.

The various special parameters used in the script are

$1—the first argument

$2—the second argument

$0—the name of the script

$#--the no. of arguments

$*--the complete set of positional parameters as a single string.

PROGRAMS:

C is the robust language & most of the Unix is written in c.

C is a compiled language, i.e. u can use one of the system editor to write the program. Then submit this program to C compiler, then run the program.

U can build C programs in Unix also.

Steps for writing a C program in UNIX:

Use an editor such as vi, ex or ed to write the program. The file name of the program must end with .c in order to identify it as a C program by the compiler.

Submit the file to C compiler using cc.

Eg:

$cc filename.c

Run the program using $a.out

UNIT-II

FILES AND DIRECTORIES

CLOs	Course Learning Outcome
CLO 4	Illustrate file processing operations such as standard I/O and formatted I/O.
CLO 5	Illustrate memory management of file handling through file/region lock.
CLO 6	Design and Implement in C some standard linux utilities.

SYSTEM CALLS

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

File descriptor: To the kernel all open files are referred to by file descriptors. A fd is a non negative integer. When open an existing file or create a new file, the kernel returns a (position in the table) fd to the process.

open function:

A file is opened or created by calling the open function.

#include <fcntl.h>

int open(const char *pathname, int oflag, .../ mode_t mode */);

Returns: file descriptor if OK, -1 on error.

The pathname is the name of the file to open or create. This function has a multitude of options, which are specified by the oflag argument.

This argument is formed by ORing together one or more of the following constants from the <fcntl.h> header:

O_RDONLY	open for reading only.
O_WRONLY	open for writing only.
O_RDWR	open for reading and writing.

Most implementations define O_RDONLY as 0, O_WRONLY as 1, and O_RDWR as 2, for compatibility with older programs.

The following constants are optional:

O_APPEND Append to the end of file on each write.

O_CREAT Create the file if it doesn't exist. This option requires a third argument to the open function, the mode, which specifies the access permission bits of the new file.

O_EXCL Generate an error if O_CREAT is also specified and the file already exists

O_TRUNC If the file exists and if it is successfully opened for either write-only or readwrite, truncate its length to 0.

creat Function

A new file can also be created by calling the creat function.

Syntax:

```
#include <fcntl.h>
 int creat( const char *pathname,
   mode_t mode);
```

Returns: file descriptor opened for write-only if OK, -1 on error.

This function is equivalent to

open (pathname, O_WRONLY | O_CREAT | O_TRUNC, mode);

close Function:

An open file is closed by calling the close function.

```
#include <unistd.h>
int close(int filedes);
```

Returns: 0 if OK,-1 on error.

read :

Data is read from an open file with the read function.

```
#include <unistd.h>
ssize_t read( int filedes, void *buf, size_t nbytes);
```

Returns: number of bytes read, 0 if end of file, -1 on error

read attempts to read *nbyte* characters from the file descriptor *filedes* into the buffer *buf*. *buf* represents the pointer to the generic buffer. *nbyte* is the size of the buffer.

write function:

Data is written to an open file with the write function.

#include <unistd.h>

ssize_t write(int filedes, const void *buf, size_t nbytes);

Returns: number of bytes written if OK,-1 on error

write writes *nbyte* no. of bytes from the generic buffer *buf* to the file descriptor *filedes*.

Implementing cat:

```
#include< fcntl.h> #define
BUFSIZE 1 main(int argc,
        char *argv[])
{
int fd, n; char buf;
fd=open(argv[1],O_RDONLY);
printf(—contents of %s file  are‖,argv[1]);
while((n=read(fd,&buf,1))>0)
{
write(1,&buf,1);
}}
```

Implementing mv:

```
#include< fcntl.h>
#include< stdio.h>
#include<unistd.h>
#include<sys/ stat.h>
main(int argc, char *argv[])
{
int fd1,fd2;
fd1=open(argv[1],O_RDONLY);
fd2= creat(argv[2],S_IWUSR);
rename(fd1,fd2);  unlink(argv[1]);
}
```

lseek function:

An open file's offset can be set explicitly by calling lseek.

```
#include <unistd.h>
off_t lseek(int filedes, off_t offset,
    int whence);
```

Returns: new file offset if OK, -1 on error.

The interpretation of the offset depends on the value of the whence argument.

If whence is:

SEEK_SET--the file's offset is set to offset bytes from the beginning of the file.

SEEK_CUR--the file's offset is set to its current value plus the offset. The offset can be positive or negative.

SEEK_END--the file's offset is set to the size of the file plus the offset. The offset can be positive or negative.

Eg: lseek(fd, 10, SEEK_CUR)

It moves the pointer forwarded by 10 characters from its current position.

Eg:

lseek(fd, -10, SEEK_END)

It sets pointer 10 characters before EOF.

lseek returns position of pointer in bytes from the beginning of the file.The return value can be used to determine the file size.

Size= lseek(fd, 0,SEEK_END)

An existing file descriptor is duplicated by either of the following functions.

```
#include <unistd.h>
int dup(int filedes);
int dup2(int filedes, int filedes2);
```

Both return: new file descriptor if OK,-1 on error.

The new file descriptor returned by dup is guaranteed to be the lowest-numbered available file descriptor.

With dup2, we specify the value of the new descriptor with the filedes2 argument. If filedes2 is already open, it is first closed. If filedes equals filedes2, then dup2 returns filedes2 without closing it.

dup(filedes); is equivalent to fcntl(filedes,

F_DUPFD, 0);

dup2(filedes, filedes2); is equivalent to
close(filedes2);

fcntl(filedes, F_DUPFD, filedes2);

ioctl Function:

It performs a variety of control functions on devices. The ioctl function has always been the catchall for I/O operations.

```
#include <unistd.h>        /* System V */
#include <sys/ioctl.h>/* BSD and Linux*/
#include <stropts.h> /* XSI STREAMS */
int ioctl(int filedes, int request, ...);
```

Returns: -1 on error, something else if OK.

stat, fstat, and lstat Functions:

```
#include <sys/stat.h>
int stat(const char * pathname, struct stat *
buf); int fstat(int filedes, struct stat *buf);  int
lstat(const char * pathname, struct stat
   * buf);
```

All three return: 0 if OK, -1 on error.

Given a pathname, the stat function returns a structure of information about the named file. The fstat function obtains information about the file that is already open on the descriptor filedes.

The definition of the structure can be:

```
struct stat {
mode_t st_mode; /*file type & mode*/
ino_t st_ino;           /* i-node number */
dev_t st_dev;         /* device number */
nlink_t st_nlink; /* number of links */
uid_t    st_uid;      /* user ID of owner */
gid_t    st_gid;      /* group ID of owner */
off_t st_size;        /* size in bytes, for regular files */
time_t st_atime;      /* time of last access */
time_t st_mtime;       /* time of last modification */
time_t st_ctime; /* time of last file status change */

blksize_t st_blksize; blkcnt_t st_blocks; };
```

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

The type of a file is encoded in the st_mode member of the stat structure.

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

File type macros are defined in <sys/stat.h>.

File Access Permissions:

The st_mode value also encodes the access permission bits for the file.

S_IRUSR	user-read
S_IWUSR	user-write
S_IXUSR	user-execute
S_IRGRP	group-read
S_IWGRP	group-write
S_IXGRP	group-execute
S_IROTH	other-read
S_IWOTH	other-write
S_IXOTH	other-execute

Program to print type of file.

```
#include<sys/ types.h> #include<sys/stat.h>  int
                main(int argc, char *argv[])
{ int i;
Struct stat buf;
Char *ptr;
for(i=1;i< argc;i++)
{
printf(—%sǁ,argv[i]);
if( lstat(argv[i],&buf)<0)
{
printf(—lstat errorǁ);
Continue;}
If(S_ISREG( buf.st_mode))
ptr=—regular||;
elseif(S_ISDIR( buf.st_mode))  ptr=—directory||;
elseif(S_ISCHR( buf.st_mode))  ptr=—character special||;
elseif(S_ISBLK( buf.st_mode))
ptr=—block special||; elseif(S_ISFIFO(buf.st_mode))  ptr=—fifo||;
else    ptr=—unknown||; printf(—%s||,ptr); }
 exit(0); }
```

access function:

#include <unistd.h>

int access(const char *pathname, int mode); Returns: 0 if OK,-1 on error.

The mode is the bitwise OR of any of the constants shown below.

Mode	description
R_OK	test for read permission test
W_OK	for write permission
X_OK	test for execute permission
F_OK	test for existence of a file

umask Function:

The umask function sets the file mode creation mask for the process and returns the previous value.

```
#include <sys/stat.h>
mode_t umask(mode_t cmask);
```

Returns: previous file mode creation mask.

The cmask argument is formed as the bitwise OR of any of the nine constants like S_IRUSR, S_IWUSR, and so on.

File Sharing loop:

The UNIX System supports the sharing of open files among different processes.

The kernel uses three data structures to represent an open file:

1.**Process table**: Every process has an entry in the process table. Within each process table entry is a table of open file descriptors, with one entry per descriptor.

Associated with each file descriptor are

a)The file descriptor flags

b)A pointer to a file table entry

2.**File table**: The kernel maintains a file table for all open files. Each file table entry contains

a)The file status flags for the file, such as read, write, append, sync, and non blocking

b) The current file offset

A pointer to the v-node table entry for the file

3.**v-node table**: Each open file (or device) has v-node structure that contains information about the type of file and pointers to functions that operate on the file.

File and Directory Maintenance functions:

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_t  mode);
```

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

The chmod function operates on the specified file, whereas the fchmod function operates on a file that has already been opened.

The mode constants for chmod functions, from <sys/stat.h>

Mode

S_IRUSR

S_IWUSR

S_IXUSR

S_IRWXU

chown, fchown, and lchown Functions:

The chown functions allow us to change the user ID of a file and the group ID of a file.

```
#include <unistd.h>
int chown (const char *pathname, uid_t owner, gid_t group);
int fchown (int filedes, uid_t owner, gid_t group);
int lchown (const char *pathname, uid_t owner, gid_t group);
```

All three return: 0 if OK,-1 on error.

These three functions operate similarly unless the referenced file is a symbolic link.

unlink:

To remove an existing directory entry, we call the unlink function.

```
#include <unistd.h>
int unlink(const char *pathname);
```

Returns: 0 if OK, 1 on error.

This function removes the directory entry and decrements the link count of the file referenced by pathname. If there are other links to the file, the data in the file is still accessible through the other links. The file is not changed if an error occurs.

link:

we can create a link to an existing file is with the link function.

```
#include <unistd.h>
int link(const char *existingpath, const char *newpath);
```

Returns: 0 if OK, -1 on error.

This function creates a new directory entry, newpath, that references the existing file existingpath. If the newpath already exists, an error is returned. Only the last component of the newpath is created.

symlink :

A symbolic link is created with the symlink function.

```
#include <unistd.h>
int symlink(const char *actualpath, const char *sympath);
```

Returns: 0 if OK, -1 on error.

A new directory entry, sympath, is created that points to actualpath. It is not required that actualpath exist when the symbolic link is created.

mkdir and rmdir :

Directories are created with the mkdir function and deleted with the rmdir function.

```
#include <sys/stat.h>
int mkdir (const char *pathname, mode_t mode);
```

Returns: 0 if OK, -1 on error.

rmdir:

An empty directory is deleted with the rmdir function. An empty directory is one that contains entries only for dot and dot-dot.

```
#include <unistd.h>
int  rmdir(const  char  *pathname);
```

Returns: 0 if OK, -1 on error.

If the link count of the directory becomes 0 with this call, and if no other process has the directory open, then the space occupied by the directory is freed.

chdir, fchdir:

We can change the current working directory of the calling process by calling the chdir or fchdir functions.

```
#include <unistd.h>
int chdir (const char *pathname);
int fchdir(int filedes);
```

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

We can specify the new current working directory either as a pathname or through an open file descriptor.

Example of chdir function:

```
int main(void)
 {
 if (chdir("/tmp") < 0)
err_sys("chdir failed");
 printf("chdir to /tmp succeeded\n");
exit(0);
 }
```

getcwd :

```
 #include <unistd.h>
char *getcwd (char *buf, size_t size);
```

Returns: buf if OK, NULL on error

We must pass to this function the address of a buffer, buf, and its size (in bytes). The buffer must be large enough to accommodate the absolute pathname plus a terminating null byte, or an error is returned.

Example of getcwd function:

```
Int main(void)
 {
 char *ptr; int size;
if (chdir("/usr/spool/uucppublic") < 0)
   err_sys("chdir failed");
 ptr = path_alloc(&size); /* our own function */
 if (getcwd(ptr, size) == NULL)  err_sys("getcwd
 failed");
 printf("cwd = %s\n", ptr);
 exit(0); }
```

Directory handling system calls:

opendir:

#include <dirent.h>

DIR *opendir(const char *pathname);

Returns: pointer if OK, NULL on error.

readdir:

struct dirent *readdir(DIR *dp);

Returns: pointer if OK, NULL at end of directory or error.

The dirent structure defined in the file <dirent.h> is implementation dependent. Implementations define the structure to contain at least the following two members:

```
struct dirent
 {
ino_t d_ino;/* i-node number */
char d_name [NAME_MAX + 1]; /* null-terminated filename */
}
```

closedir, rewinddir, telldir, seekdir:

int closedir(DIR *dp);

Returns: 0 if OK, -1 on

error. void rewinddir(DIR

*dp); long telldir (DIR *dp); void

seekdir(DIR *dp, long loc);

Returns: current location in directory associated with dp.

STANDARD IO:

All the I/O routines centered around file descriptors. When a file is opened, a file descriptor is returned, and that descriptor is then used for all subsequent I/O operations. With the standard I/O library, the discussion centers around streams.

When we open a stream, the standard I/O function *fopen* returns a pointer to a FILE object. This object is normally a structure that contains all the information required by the standard I/O library to manage the stream.

Standard Input, Standard Output, and Standard Error:

Three streamsarepredefined and automatically available to a process: standard input, standard output, and standard error. These streams refer to the same files as the file descriptors STDIN_FILENO, STDOUT_FILENO, and STDERR_FILENO.

These three standard I/O streams are referenced through the predefined file pointers stdin, stdout, and stderr. The file pointers are defined in the <stdio.h> header.

Opening a Stream: fopen()

The following three functions open a standard I/O stream.

```
 #include < stdio.h>
FILE *fopen(const char *pathname,
const char *type);
FILE *freopen(const char *pathname,
const char *type,FILE *fp);
FILE   *fdopen(int   filedes,const   char   *type);
```

All three return: file pointer if OK, NULL on error.

The fopen function opens a specified file.

The freopen function opens a specified file on a specified stream, closing the stream first if it is already open. If the stream previously had an orientation, freopen clears it.

The fdopen function takes an existing file descriptor, which we could obtain from the open, dup, dup2, fcntl, pipe, socket, socketpair, or accept functions, and associates a standard I/O stream with the descriptor.

Type	**Description**
r / rb →	open for reading
W / wb →	truncate to 0 length or create for writing
a / ab →	append; open for writing at end of file, or create for writing
r+ / r+b / rb+ →	open for reading and writing
w+ / w+b / wb+ →	truncate to 0 length or create for reading and writing
a+/ a+b / ab+ →	open or create for reading and writing at end of file

fclose:

An open stream is closed by calling fclose.

#include <stdio.h>

int fclose(FILE *fp); Returns:

0 if OK, EOF on error.

Any buffered output data is flushed before the file is closed. Any input data that may be buffered is discarded. If the standard I/O library had automatically allocated a buffer for the stream, that buffer is released.

Reading and Writing a Stream:

Once we open a stream, we can choose from among three types of unformatted I/O:

1. Character-at-a-time I/O. We can read or write one character at a time.
2. Line-at-a-time I/O. If we want to read or write a line at a time, we use fgets and fputs.
3. Direct I/O. This type of I/O is supported by the fread and fwrite functions

Input Functions:

Three functions allow us to read one character at a time.

```
#include    <stdio.h>
int  getc  (FILE  *fp);
int  fgetc  (FILE  *fp);
int getchar (void);
```

All three return: next character if OK, EOF on end of file or error.

The function getchar is equivalent to getc(stdin). getc can be implemented as a macro, whereas fgetc cannot be implemented as a macro.

Output Functions:

an output function that corresponds to each of the input functions are:

```
#include <stdio.h>
int putc (int c, FILE *fp);
int fputc (int c, FILE
*fp); int putchar (int c);
```

All three return: c if OK, EOF on error.

Like the input functions, putchar(c) is equivalent to putc(c, stdout), and putc can be implemented as a macro, whereas fputc cannot be implemented as a macro.

Line-at-a-Time I/O:

#include <stdio.h>

char *fgets (char *buf, int n, FILE

*fp); char *gets(char *buf);

Both return: buf if OK, NULL on end of file or error.

Both specify the address of the buffer to read the line into. The gets function reads from standard input, whereas fgets reads from the specified stream.

With fgets, we have to specify the size of the buffer, n. This function reads up through and including the next newline, but no more than n1 characters, into the buffer. The buffer is terminated with a null byte.

The gets function should never be used. The problem is that it doesn't allow the caller to specify the buffer size. This allows the buffer to overflow, if the line is longer than the buffer, writing over whatever happens to follow the buffer in memory.

Line-at-a-time output is provided by fputs& puts.

```
#include <stdio.h>
int fputs(const char *str, FILE *fp);
int puts(const char *str);
```

Both return: non-negative value if OK, EOF on error.

The function fputs writes the null-terminated string to the specified stream. The null byte at the end is not written.

Copy standard input to standard output using getc and putc:

```
int main(void)
 {
 int c;
 while ((c = getc( stdin)) != EOF)  if
(putc(c, stdout) == EOF)
err_sys("output error"); if (  ferror(
stdin))
err_sys("input error");
 exit(0);}
```

Copy standard input to standard output using fgets and fputs:

```
int main(void)
 {
char buf[MAXLINE];
while (fgets( buf, MAXLINE, stdin) != NULL)  if
( fputs( buf, stdout) == EOF)  err_sys("output
error"); if (ferror( stdin))


err_sys("input error");
   exit(0); }
```

DIRECT IO FUNCTIONS:

```
#include<stdio.h>
size_t fread(void *ptr, size_t size, size_t nobj, FILE *fp);  size_t
fwrite(const void *ptr, size_t size, size_t nobj, FILE *fp);
```

Both return: number of objects read or written

Example:

```
struct {
Short count;
Long total;
Char name[NAMESIZE];
}item;
If (fwrite(&item, sizeof(item),1,fp)!=)
err_sys(—fwrite error‖);
```

Formatted Output:

Formatted output is handled by the four printf functions.

```
#include <stdio.h>
 int printf(const char *format, ...);
 int fprintf(FILE *fp, const char *format, ...);
```

Both return: number of characters output if OK, negative value if output error.

int sprintf (char *buf, const char *format, ...);

int snprintf (char *buf, size_t n, const char *format, ...);

Both return: number of characters stored in array if OK, negative value if encoding error.

The printf function writes to the standard o/p, fprintf writes to the specified stream& sprintf places the formatted characters in the array buf. The sprintf function automatically appends a null byte at the end of the array, but this null byte is not included in the return value.

Formatted Input:

Formatted input is handled by the three scanf functions.

```
#include <stdio.h>

 int scanf (const char *format, ...);
int fscanf (FILE *fp, const char *format, ...);
int sscanf (const char *buf, const char *format, ...);
```

All three return: number of input items assigned, EOF if input error or end of file before any conversion.

The scanf family is used to parse an input string and convert character sequences into variables of specified types.

Handling errors:

```
#include<stdio.h>
int ferror(FILE *fp);
int feof(FILE *fp);
```

Both return: nonzero (true) if condition is true, 0 (false) otherwise

These functions return the same value whether an error occurs or the end of file is reached.

UNIT-III

PROCESS AND SIGNALS

CLOs	Course Learning Outcome
CLO 7	Understand process structure, scheduling and management through system calls.
CLO 8	Implement C programs to control process using system calls and identify difference between process and threads.
CLO 9	Generalize signal functions to handle interrupts by using system calls.

PROCESS AND SIGNALS:

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.

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.

uid_t geteuid (void);

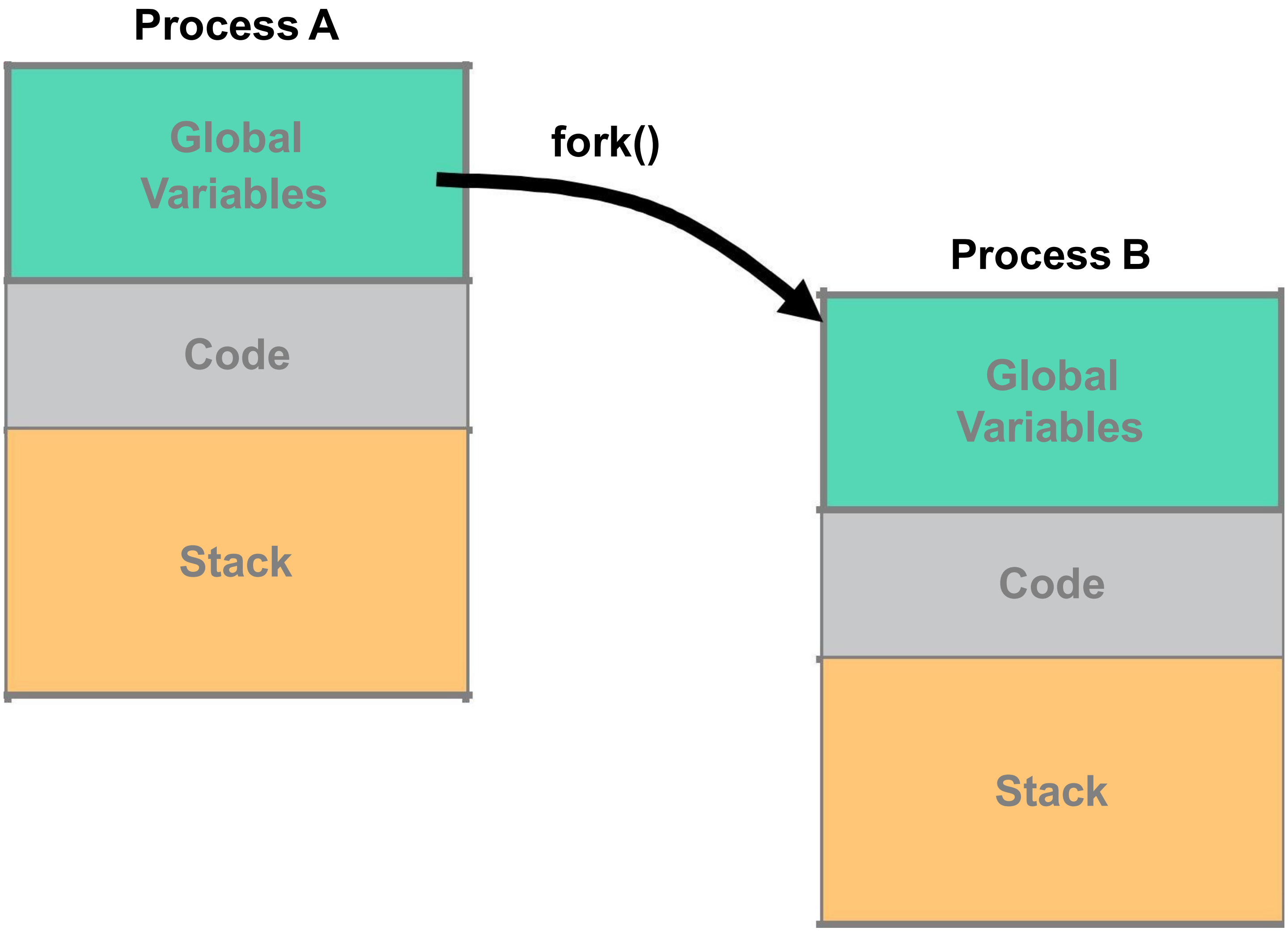
Returns: effective user ID of calling process gid_t getgid (void); Returns: real group ID of calling process. gid_t getegid (void);

Returns: effective group ID of calling process.

None of these functions has an error return.

Process structures contains: process state, pid, ppid, file table, signal table, threads, scheduling and other information.

fork()


Process A
Global Variables
Code
Stack
fork()
Process B
Global Variables
Code
Stack

PROCESS ATTRIBUTES

A process has a series of characteristics:

The process ID or PID: a unique identification number used to refer to the process.

The parent process ID or PPID: the number of the process (PID) that started this process.

Terminal or TTY: terminal to which the process is connected.

User name of the real and effective user (RUID and EUID): the owner of the process. The real owner is the user issuing the command, the effective user is the one determining access to system resources. RUID and EUID are usually the same, and the process has the same access rights the issuing user would have.

Process states in Linux:

Running: Process is either running or ready to run

Interruptible: a Blocked state of a process and waiting for an event or signal from another process

Uninterruptible: a blocked state. Process waits for a hardware condition and cannot handle any signal

Stopped: Process is stopped or halted and can be restarted by some other process

Zombie: process terminated, but information is still there in the process table.

The UNIX process hierarchy

every process in UNIX (except one) has a parent processes may create many children (via fork()) example: the UNIX boot procedure

- initially, a single parentless process called init runs
- init reads a file which logs the connected terminals
- init forks a login process for each terminal
- if a login process validates a user it forks a

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.

Eg:

```
int glob = 6;
char buf[] = "a write to stdout\n";
int main(void)
{
  int var; pid_t pid; var = 88;
   if (write(STDOUT_FILENO, buf, sizeof (buf)-1) !=
    sizeof (buf)-1)
   err_sys("write error");
   printf("before fork\n");
   if ((pid = fork()) < 0)
               {err_sys("fork error");
```

```
 else if (pid == 0)
{
 glob++; var++;
 }
 else
 {
 sleep(2);
  }
  printf("pid = %d, glob = %d, var = %d\n", getpid(),
 glob, var);
 exit(0);
}
```

```
     $ ./a.out
 a write to stdout
    before fork
pid = 430, glob = 7, var = 89
pid = 429, glob = 6, var = 88
```

uses for fork:

1. When a process wants to duplicate itself so that the parent and child can each execute different sections of code at the same time.
2. When a process wants to execute a different program. This is common for shells.

vfork :

The function vfork has the same calling sequence and same return values as fork.

The vfork function creates the new process, just like fork, without copying the address space of the parent into the child, as the child won't reference that address space.

vfork guarantees that the child runs first, until the child calls exec or exit. When the child calls either of these functions, the parent resumes.

exit :

A process can terminate normally in five ways:

1.Executing a return from the main function.

2.Calling the exit function.

3.Calling the _exit or _Exit function.

4.Executing a return from the start routine of the last thread in the process.

5.Calling the pthread_exit function from the last thread in the process.

Once the process terminates, the kernel closes all the open descriptors for the process, releases thememory that it wasusing.

Exit Functions

Three functions terminate a program normally:

_exit and _Exit, which return to the kernel immediately, and exit, which performs certain cleanup processing and then returns to the kernel.

```
#include      <stdlib.h>
void  exit(  int  status);
void _Exit( int status);
#include      <unistd.h>
void _exit( int status);
```

wait and waitpid Functions:

```
#include <sys/ wait.h>
pid_t wait(int *statloc);
pid_t waitpid( pid_t pid, int *statloc, int options);
```

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

The wait function can block the caller until a child process terminates, whereas waitpid has an option that prevents it from blocking.

The waitpid function doesn't wait for the child that terminates first; it has a number of options that control which process it waits for.

For both functions, the argument statloc is a pointer to an integer. If this argument is not a null pointer, the termination status of the terminated process is stored in the location pointed to by the argument.

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.

The options constants for waitpid:

WNOHANG-wait pid will not block if a child specified by pid is not immediately available.

WUNTRACED-Return status of child if child stopped and status has not already been returned (assumes job control support)

The waitpid function returns the process ID of the child that terminated and stores the child's termination status in the memory location pointed to by statloc.

The waitpid function provides these features that are not provided by the wait function are:

waitpid lets us to wait for one particular process

waitpid provides a non-blocking version of wait

waitpid supports job control (with the WUNTRACED option)

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.

Exec functions:

exec replaces the current process its text, data, heap, and stack segments with a brand new program from disk.

with the exec functions, we can initiate new programs.

There are six different exec functions.

```
 #include <unistd.h>
int execl (const char *pathname, const
   char *arg0, ... /* (char *)0 */ );
int execv (const char *pathname, char
   *const argv []);
int execle (const char *pathname, const char
   *arg0, ... /* (char *)0, char *const envp[] */ );
int execve (const char *pathname, char
   *const argv[], char *const envp []);
int execlp (const char *filename, const
   char *arg0, ... /* (char *)0 */ );
int execvp (const char *filename, char *const
   argv []);
```

The functions execl, execlp, and execle require each of the command-line arguments to the new program to be specified as separate arguments. We mark the end of the arguments with a null pointer.

For the other three functions (execv, execvp, and execve), we have to build an array of pointers to the arguments, and the address of this array is the argument to these three functions.

zombie process:

A **zombie process** or **defunct process** is a process that has completed execution but still has an entry in the process table. This entry is still needed to allow the parent process to read its child's exit status.

When a process terminates, the OS kernel:

_ Discards all memory used by the process

– closes all process‘ files

– Keeps some minimal info (PID, exit status, CPU time usage)

– Provides info to parent when parent calls wait

The ps – x command prints the status of a zombie process as

Z.

#sample progrom for zombies:

```
#include <stdio.h>
main()
{
    int pid;
    pid=fork(); /* Duplicate */
    if (pid!=0) /* Branch based on return value from fork() */
       {
          while (1) /* never terminate, and never execute a wait() */
      wait(&statloc);
      sleep(1000);
       }
    else
       {
          exit(42); /* Exit with a silly number */
       }
}
```

Input:

```
$ cc prog17.c
./a.out&   ... execute the program in the background.
[1] 13545
```

Output:

```
$ ps
  PID TT STAT TIME COMMAND
13535 p2 s    0:00 -ksh(ksh) ...the shell
13545 p2 s   0:00 aombie.exe...the parent process
13536 p2 z   0:00 <defunct> ...the zombie child process
13537 p2 R     0:00 ps
$ kill 13545                   ... kill the parent process.
[1] Terminated zombie.exe
$ ps                         ... notice the zombie is gone now.
    PID TT STAT TIME COMMAND
13535 p2 s    0:00 -csh(csh)
13548 p2 R     0:00 ps
```

Orphan Process:

An orphan process is a process that is still executing but whose parent died.

If a parent dies before its child, the child is automatically adopted by the original "init" process, PID 1.

To illustrate this, modify the previous program by inserting a sleep statement into the child's code. This ensured that the parent process terminated before the child.

```
#include <stdio.h>
main()
{
    int pid;
    printf("I'm the original process with PID %d and PPID
           %d.\n", getpid(),getppid());
    pid=fork(); /* Duplicate. Child and parent continue from here.*/
    if (pid!=0) /* Branch based on return value from fork() */
        { /* pid is non-zero, so I must be the parent */
           printf("I'm the parent process with PID %d and PPID
                   %d.\n", getpid(),getppid());
           printf("My child's PID is %d.\n", pid);
        }
    else
        { /* pid is zero, so I must be the child. */
           sleep(5); /*Make sure that the parent terminates first. */
           printf("I'm the child process with PID %d and PPID %d.\n",
                  getpid(),getppid());
        }
    printf("PID %d terminates.\n",pid); /* Both processes execute this */
}
```

```
$cc prog18.c
./a.out                    ... run the program.
I'm the original process with PID 13364 and PPID 13346.  I'm the
parent process with PID 13364 and PPID 13346.
PID 13364 terminates.
I'm the child process with PID 13365 and PPID 1.           ...orphaned!  PID
13365 terminates.                  ... child terminates.
```

SIGNALS:

A *signal* (software interrupt) is a notification to a process that an event has occurred.

Signals are software interrupts. Signals provide a way of handling asynchronous events.

Every signal has a name. These names all begin with the three characters SIG. These names are all defined by positive integer constants (the signal number) in the header <signal.h>.

No signal has a signal number of 0.

- Signals can be sent
  - by one process to another process(or itself)
  - by the kernel to a process

SIGCHLD signal: a signal sent by the kernel whenever a process terminates, to the parent of the terminating process

SIGNAL FUNCTION

The simplest interface to the signal features of the UNIX System is the signal function.

include <signal.h>

void (*signal(int signo, void (*func)(int)))(int);

Returns: previous disposition of signal if OK, SIG_ERR on error.

The signo argument is just the name of the signal. The value of func is (a) the constant SIG_IGN, (b) the constant SIG_DFL, or (c) the address of a function to be called when the signal occurs.

Signal Names:

- SIGABRT - Abnormal termination
- SIGALRM – Time out
- SIGILL – Illegal hardware instruction
- SIGINT – Terminal interrupt character
- SIGKILL – Termination of process (cant be caught or ignored)
- SIGPIPE – Write to pipe with no readers
- SIGSEGV – Invalid memory segment access.
- SIGTERM – TERMINATION
- SIGUSR1 – User defined signal1
- SIGUSR2 – User defined signal2
- SIGCHLD - Child process has stopped or exited
- SIGCONT - Continue executing, if stopped
- SIGSTOP - Stop executing (cant be caught or ignored)
- SIGTSTP - Terminal stop signal
- SIG_IGN - Ignore the signal
- SIG_DFL - Restore default behavior

The prototype for the signal function states that the function requires two arguments and returns a pointer to a function that returns nothing (void). The first argument, signo, is an integer. The second argument is a pointer to a function that takes a single integer argument and returns nothing. The function whose address is returned as the value of signal takes a single integer argument (the final (int)).

Simple program for Signal Handler:

```
 #include<signal.h>
     void abc();
       main()
{
printf(—press del to  stop‖);
signal(SIGINT,abc);
for(;;);
}
void abc()
{
printf(—signal received‖);
printf(—you pressed del key‖);
}
```

Unreliable Signals:

Unreliable signals mean that signals could get lost: a signal could occur and the process would never know about it.

Also, a process had little control over a signal: a process could catch the signal or ignore it. – RELIABLE SIGNAL

INTERRUPTED SYSTEM CALLS:

It is a system call within the kernel that is interrupted when a signal is caught.

To support this feature, the system calls are divided into two categories: the "slow" system calls and all the others.

The slow system calls are those that can block forever.

Reads that can block the caller forever if data isn't present with certain file types.

Writes that can block the caller forever if the data can't be accepted immediately by these same file types.

Opens that block until some condition occurs on certain file types.

The pause function and the wait function.

Some of the inter process communication 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 itself.

#include <signal.h>

int kill(pid_t pid, int

signo); int raise(int signo);

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

There are four different conditions for the pid argument to kill.

pid > 0The signal is sent to the process whose process ID is pid.

pid == 0The signal is sent to all processes whose process group ID equals the process group ID of the sender and for which the sender has permission to send the signal.

pid < 0The signal is sent to all processes whose process group ID equals the absolute value of pid and for which the sender has permission to send the signal.

pid == 1 The signal is sent to all processes on the system for which the sender has permission to send the signal.

The super user can send a signal to any process.

ALARM FUNCTION:

The alarm function allows us to set a timer that will expire at a specified time in the future.

When the timer expires, the SIGALRM signal is generated.

#include <unistd.h>

unsigned int alarm(unsigned int seconds);

Returns: 0 or number of seconds until previously set alarm.

The seconds value is no.of clock seconds in the future

when the signal should be generated.

If, when we call alarm, a previously registered alarm clock for the process has not yet expired, the number of seconds left for that alarm clock is returned as the value of this function.

If a previously registered alarm clock for the process has not yet expired and if the seconds value is 0, the previous alarm clock is canceled. The number of seconds left for that previous alarm clock is still returned as the value of the function.

pause function:

The pause function suspends the calling process until a signal is caught.

```
#include <unistd.h>

int pause(void);
```

Returns: 1 with errno set to EINTR

The only time pause returns is if a signal handler is executed and that handler returns.

Example 2:

```
#include<signal.h>
#include<unistd.h>
sig_alrm(int signo)
{
printf(―alarm time elapsed‖):
exit(0);
}
main()
{
signal(SIGALRM,sig_alrm);
alarm(20);
Abort();
}
```

abort Function:

The abort function causes abnormal program termination.

#include <stdlib.h>

void abort(void);

This function never returns.

This function sends the SIGABRT signal to the caller.

system Function:

It is convenient to execute a command string from within a program.

```
#include <stdlib.h>

int system(const char *cmdstring);
```

If cmdstring is a null pointer, *system* returns nonzero only if a command processor is available. This feature determines whether the *system* function is supported on a given operating system. Under the UNIX System, *system* is always available.

sleep Function:

```
#include <unistd.h>
unsigned int sleep(unsigned int seconds);
```

Returns: 0 or number of unslept seconds.

This function causes the calling process to be suspended until either

The amount of wall clock time specified by seconds has elapsed. The return value is 0.

A signal is caught by the process and the signal handler returns. Return value is the number of unslept seconds.

UNIT-IV
INTER PROCESS COMMUNICATION

CLOs	Course Learning Outcome
CLO 10	Design and implement inter process communication (IPC) in client server environment by using pipes and named pipes system calls.
CLO 11	Design and implement inter process communication (IPC) in client server environment by using message queues systems calls.
CLO 13	Illustrate client server authenticated communication in IPC through shared memory.

INTER PROCESS COMMUNICATION:

The communication of more than one process with an another process for making a program is known as the inter process communication.

IPC is divided into pipes,FIFOs, message queues, semaphores and shared memory.

PIPEs:

Pipes are the oldest form of UNIX System IPC and are provided by all UNIX systems. Pipes have two limitations.

1. They have been half duplex.
2. Pipes can be used only between processes that have a common ancestor.

A pipe is a one-way mechanism that allows two related processes to send data from one of them to the other one.

A pipe is created by calling the pipe function.

#include <unistd.h>

int pipe(int filedes[2]);

Returns: 0 if OK, 1 on error.

- Two file descriptors are returned through the filedes argument: filedes[0] is open for reading, and filedes[1] is open for writing. The output of filedes[1] is the input for filedes[0].

popen and pclose :

A common operation is to create a pipe to another process, to either read its output or send it input, the standard I/O library has historically provided the popen and pclose functions.

#include <stdio.h>

FILE *popen (const char *cmdstring, const char *type);

Returns: file pointer if OK, NULL on error.

int pclose(FILE *fp);

Returns: terminationstatusof cmdstring, or -1 on error

The function popen does a fork and exec to execute the cmdstring, and returns a standard I/O file pointer. If type is "r", the file pointer is connected to the standard output of cmdstring.

- If type is "w", the file pointer is connected to the standard input of cmdstring.
- The pclose function closes the standard I/O stream, waits for the command to terminate, and returns the termination status of the shell.

FIFOs

FIFOs are sometimes called named pipes. Pipes can be used only between related processes when a common ancestor has created the pipe. With FIFOs, unrelated processes can exchange data.

Creating a FIFO is similar to creating a file.

```
#include <sys/ stat.h>

int mkfifo( const char *pathname,
   mode_t mode);
```

Returns: 0 if OK,-1 on error.

Uses for FIFOs:

FIFOs are used by shell commands to pass data from one shell pipeline to another without creating intermediate temporary files.

FIFOs are used in client-server applications to pass data between the clients and the servers.

MESSAGE QUEUES:

A message queue is a linked list of messages stored within the kernel and identified by a message queue identifier. We'll call the message queue just a queue and its identifier a queue ID.

A new queue is created or an existing queue opened by *msgget*. New messages are added to the end of a queue by *msgsnd*.

Every message has a positive long integer type field, a non-negative length, and the actual data bytes all of which are specified to msgsnd when the message is added to a msgsnd.

Messages are fetched from a queue by *msgrcv.*

Each queue has the following msqid_ds structure associated with it:

```
struct msqid_ds
{
struct ipc_perm msg_perm;
msgqnum_t msg_qnum;
msglen_t msg_qbytes;  pid_t
msg_lspid;
pid_t msg_lrpid;
time_t msg_stime; time_t msg_rtime;}
```

msgget is used to either open an existing queue or create a new queue.

#include <sys/ msg.h>

int msgget(key_t key, int flag);

Returns: message queue ID if OK,-1 on error.

On success, msgget returns the non-negative queue ID. This value is then used with the other three message queue functions.

The *msgctl* function performs various operations on a queue.

#include <sys/msg.h>

int msgctl(int msqid, int cmd, struct msqid_ds *buf);

Returns: 0 if OK,-1 on error.

The cmd argument specifies the command to be performed on the queue specified by msqid.

IPC_STAT --- Fetch the msqid_ds structure for this queue, storing it in the structure pointed to by buf.

IPC_SET ---- Copy the following fields from the structure pointed to by buf to the msqid_ds structure associated with this queue: msg_perm.uid, msg_perm.gid, msg_perm.mode, and msg_qbytes.

IPC_RMID ---- Remove the message queue from the system and any data still on the queue. This removal is immediate.

Data is placed onto a message queue by calling *msgsnd*.

```
#include <sys/msg.h>

int msgsnd(int msqid, const void *ptr,
   size_t nbytes, int flag);
```

Returns: 0 if OK, -1 on error.

When msgsnd returns successfully, the msqid_ds structure associated with the message queue is updated to indicate the process ID that made the call (msg_lspid), the time that the call was made (msg_stime),
and that one more message is on the queue (msg_qnum).

Messages are retrieved from a queue by *msgrcv.*

```
#include <sys/msg.h>
```

```
ssize_t msgrcv(int msqid, void *ptr,
   size_t nbytes , long type, int flag);
```

Returns: size of data portion of message if OK, -1 on error.

When msgrcv succeeds, the kernel updates the msqid_ds structure associated with the message queue to indicate the caller's process ID (msg_lrpid), the time of the call (msg_rtime), and that one less message is on the queue (msg_qnum).

INTER PROCESS COMMUNICATION:

The communication of more than one process with an another process for making a program is known as the inter process communication.

IPC is divided into pipes,FIFOs, message queues, semaphores, and shared memory.

SEMAPHORES

A semaphore is a counter used to provide access to a shared data object for multiple processes.

To obtain a shared resource, a process needs to do the following:

1. Test the semaphore that controls the resource.

If the value of the semaphore is positive, the process can use the resource. In this case, the process decrements the semaphore value by 1, indicating that it has used one unit of the resource.

3.If the value of the semaphore is 0, the process goes to sleep until the semaphore value is greater than 0. When the process wakes up, it returns to step 1.

The kernel maintains a semid_ds structure for each semaphore set:

```
struct semid_ds{

    struct sem *sem_base;              /*ptr to first semaphore in set */
    ushort sem_nsems;                  /*# of semaphores in set */
    time_t sem_otime;                  /*last-semop() time */
    time_t sem_ctime;                  /*last-change time*/
};
```

The sem_base pointer is worthless to a user process, since it points to memory in the kernel. What it points to is an array of sem structure, containing sem_nsems elements, one element in the array for each semaphore value in the set.

```
struct sem{
   ushort semval; /*semaphore value, always>=0*/
   pid_t        sempid;      /*pid for last operation*/
   ushort semncnt;           /*# processes awaiting semval >
   curval*/
   ushort semcnt; /*processes awaiting semval = 0*/
};
```

The first function to call is semget to obtain a semaphore ID.

#include <sys/ sem.h>

int semget(key_t key, int nsems, int flag);

Returns: semaphore ID if OK, -1 on error.

The number of semaphores in the set is nsems. If a new set is being created (typically in the server), we must specify nsems. If we are referencing an existing set (a client), we can specify nsems as 0.

The semctl function is the catchall for various semaphore operations

```
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/semh>
 int semctl(int semid, int semnum, int cmd, union semun
 arg); Returns: as follows
```

Union is declared it as

```
union semun{
    int   val;                          /*for SETVAL*/
    struct semid_ds *buf;               /*for IPC_STAT and IPC_SET*/
    ushort *array;                      /*for GETALL and SETALL*/
  };
```

The *cmd* argument specified one of the following 10 commands to be performed on the set specified by *semid*. The five commands that refer to one particular semaphore value use *semnum* to specify one member of the set. The value of *semnum* is between 0 and *nsems-1*, inclusive.

IPC_STAT: Fetch the semid_ds structure for this set ,storing it in the structure pointed to by arg.buf

IPC_SET: Set the following three fields from the structure pointed by arg.buf in the structure associated with this set : sem_perm.uid, sem_perm.gid,and sem_perm.mode. This command can be executed by a process whose effective user ID equals sem_perm.cuid or sem_perm.uid, or by a process with super user privileges.

IPC_RMID: Remove the semaphore set from the system. This removal is immediate. Any other process still using the semaphore will get an error of EIDRM on its next attempted operation on the semaphore. This command can be executed only by a process whose effective user ID equals sem_perm.cuid or sem_perm.uid, or by a process with super user privileges.

GETVAL:

Return the value of *semval* for the member *semnum.*

SETVAL:

Set the value of *semval* for the member *semnum* . The valueis specified by arg.val.

GETPID:

Return the value of *sempid* for the member *semnum.*

GETNCNT:

Return the value of *semzcnt* for the member *semnum.*

GETALL:

Fetch all the semaphore values in the set . These values are stored in the array pointed by arg.array.

Set all semaphore values in the set to the values pointed to by arg.array.

The function *semop* atomically performs an array of opertions on a semaphore set.

```
include <sys/types.h>
include < sys/ipc.h>
include < sys/sem.h>
int semop(int semid, struct sembuf semoparray[], size_t nops);
```

Returns: 0 if OK, -1 on error

Semoparray is a pointer to array of semaphore operations.

```
Struct sembuf
{
    Ushort sem_num ;
        Short sem_op;
        Short sem_flg;
  };
```

nops specifies the number of operations (elements) in the array.

The operation on each member of the set is specified by the corresponding *sem_op* Value. This value can be negative , 0 positive. (In the following discussion we refer to the ―undo‖ flag fro a semaphore. This flag corresponds to the SEM_UNDO bit in the corresponding sem_flg member.)

The earliest case is when sem_op is positive.

UNIT-V
SHARED MEMEORY AND SOCKETS

CLOs	Course Learning Outcome
CLO 12	Illustrate client server authenticated communication in IPC through shared memory.
CLO 14	Demonstrate various client server applications on network using TCP or UDP protocols.
CLO 15	Design custom based network applications using the Sockets Interface in heterogeneous platforms.

SHARED MEMORY:

Shared Memory:

Shared memory allows two or more processes to share a given region of memory. This is the fastest form of IPC, because the data does not need to be copied between the client and the server.

The kernel maintains a structure with at least the following members for each shared memory segment:

```
  struct ipc_perm shm_perm;
    size_t shm_segsz; pid_t
            shm_lpid;
        pid_t shm_cpid;
shmatt_t shm_nattch;
time_t shm_atime;
time_t shm_dtime;
time_t shm_ctime;
}
```

The first function called is usually shmget, to obtain a shared memory identifier.

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

Returns: shared memory ID if OK, -1 on error.

The shmctl function is the catchall for various shared memory operations.

```
#include <sys/shm.h>
int shmctl(int shmid, int cmd, struct
   shmid_ds *buf);
```

Returns: 0 if OK, -1 on error.

Applications of IPC

ClientServer Properties:

Let's detail some of the properties of clients and servers that are affected by the various types of IPC used between them. The simplest type of relationship is to have the client fork and exec the desired server. Two half-duplex pipes can be created before the fork to allow data to be transferred in both directions.

The server that is executed can be a set-user-ID program, giving it special privileges. Also, the server can determine the real identity of the client by looking at its real user ID.

.

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.

- – The application creates a socket
- – The socket *type* dictates the style of communication

reliable vs. best effort

connection-oriented vs. connectionless

Once configured the application can

– pass data to the socket for network transmission –receive data from the socket (transmitted through

What is Socket ?

Endpoint of any connection

Two Types :

- TCP
- UDP

Identified by Two values

- An IP Address
- A Port Number

Two essential types of sockets

SOCK_STREAM

reliable delivery in-order guaranteed

connection-oriented

Bidirectional

Segment retransmission, ack

App segments

3 2 1 socket Dest.

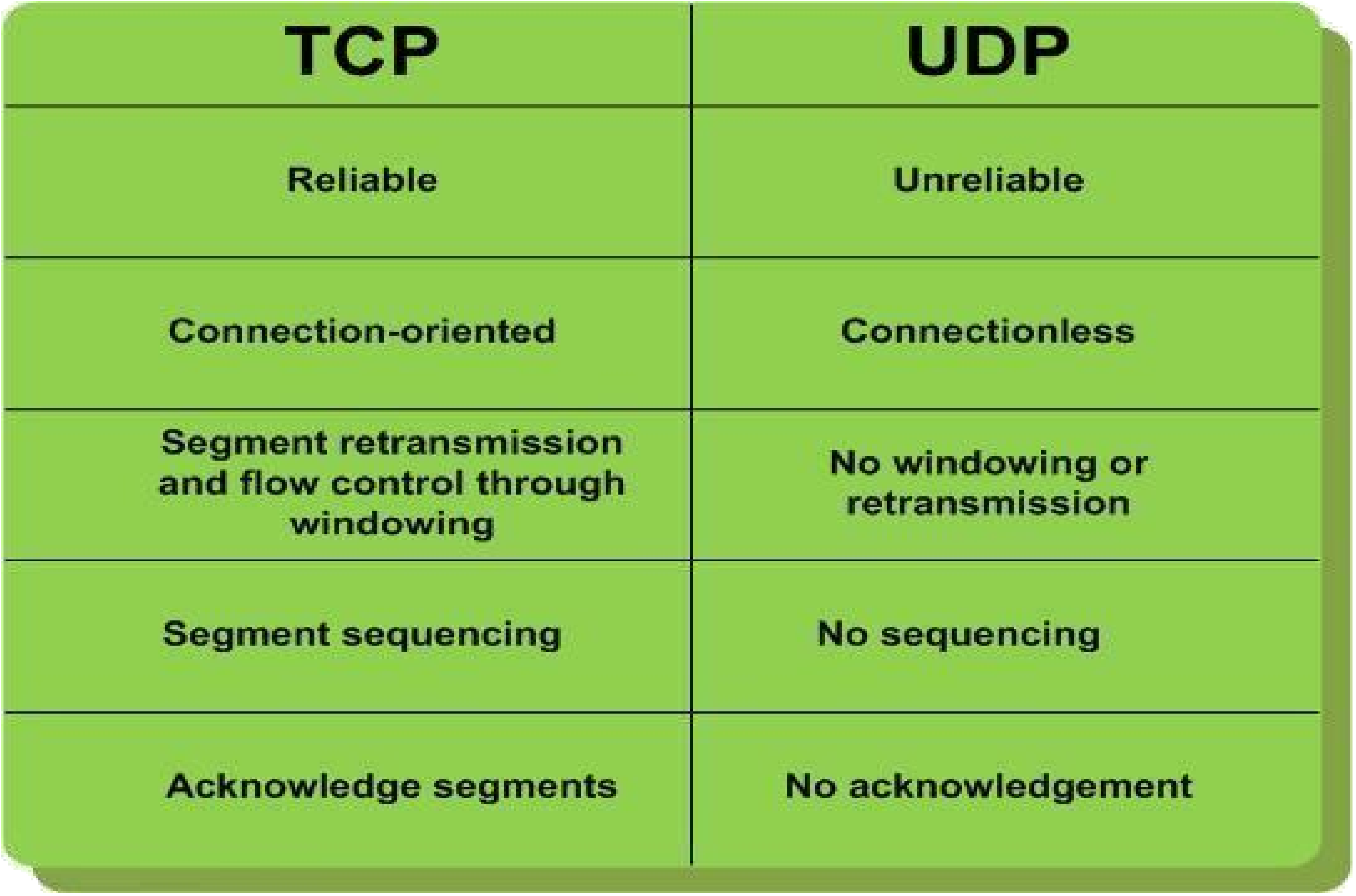
SOCK_DGRAM

- unreliable delivery
- no order guarantees
- no notion of ―connection‖–app indicates dest. for each packet
- can send or receive

App D1

3 2 1 socket D2

D3



TCP	UDP
Reliable	Unreliable
Connection-oriented	Connectionless
Segment retransmission and flow control through windowing	No windowing or retransmission
Segment sequencing	No sequencing
Acknowledge segments	No acknowledgement

SCOKET CREATION IN C

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 .service

  SOCK_DGRAM: unreliable, connectionless,

Socket Creation in C: socket

- – protocol: specifies protocol
- – usually set to –0 (zero) to select the system‘s default for the given combination of family and type.

NOTE: socket call does not specify where data will be coming from, nor where it will be going to – it just creates the interface!

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

Type of socket

Type	Description
SOCK_STREAM	STREAM socket
SOCK_DGRAM	Datagram socket
SOCK_SEQPACKET	Sequenced packet socket
SOCK_RAW	Raw socket

Protocol of socket

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

Addresses, Ports and Sockets

Like apartments and mailboxes

- You are the application
- Your apartment building address is the address
- Your mailbox is the port
- The post-office is the network
- The socket is the key that gives you access to the right mailbox (one difference: assume outgoing mail is placed by you in your mailbox)

IPv4 Socket Address Structure

Socket functions require a pointer to a socket address structure as an argument. Each supported protocol suite defines its own socket address structure. The names of these structures begins with sockaddr_ and end with a unique suffix for each protocol suite.

```
Struct in_addr{


    in_addr_t     s_addr;                              /*32bit IPv4 address*/
  };                                                  /*network byte ordered*/


struct sockaddr_in {
  uint8_t             sin_len;                         /*) */
  sa_family_t        sin_family;                       /* AF_INET */length of structure(16
  in_port_t          sin_port;                         /* 16bit TCP or UDP port number */
                                                      /*network byte ordered*/
  struct    in_addr sin_addr;                          /* 32bit IPv4 address */

                                                      /*network byte ordered*/
  char      sin_zero[8];                               /* unused */
```

SOCKET ADDRESS STRUCTURE:

- Length field simplifies the handling of variable-length socket address structures.
- Used with routing socket.
- In_addr_t datatype must be an unsigned integer type of at least 32 bits.

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 Address 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> */
```

.

New Generic Socket Address structure

Struct sockaddr

{

uint8_tsa_len;

sa_family_t   sa_family;

/* implementation dependent elements

to   provide

alignment

enough storage to hold any type of

socket

.

- If any socket address structures that the sockaddr_storage different from struct system supports have alignment sockaddr in two ways:

Requirements, the sockaddr_storage provides the strictest alignment requirement.

The sockaddr_storage is large enough to contain any socket address structure that the system supports.

Comparison of socket address structure

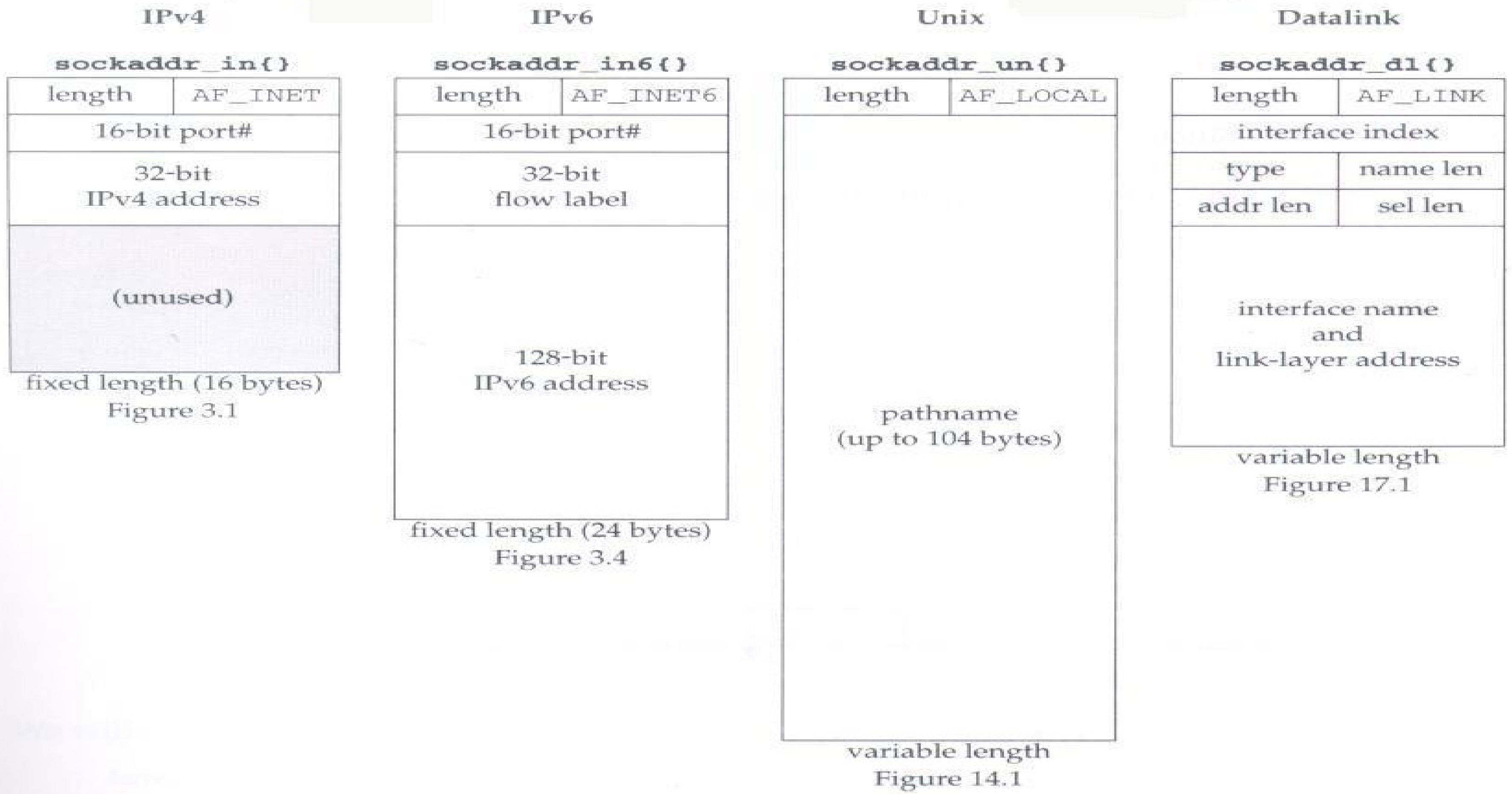


Figure 3.5 Comparison of various socket address structures.

Solution: Network Byte-Ordering

Defs:

- Host Byte-Ordering: the byte ordering used by a host (big or little)
- Network Byte-Ordering: the byte ordering used by the network – always big-endian

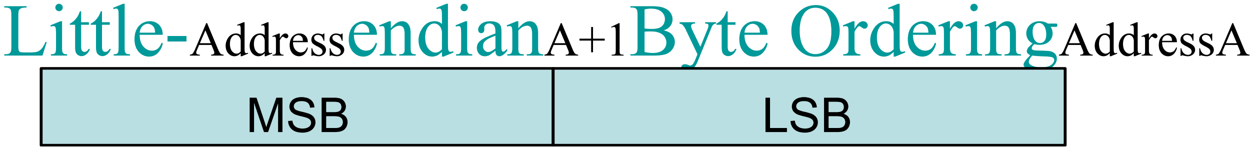
Any words sent through the network should be converted to Network Byte-Order prior to transmission (and back to Host Byte-Order once received)

BYTE ORDERING FUNCTIONS:

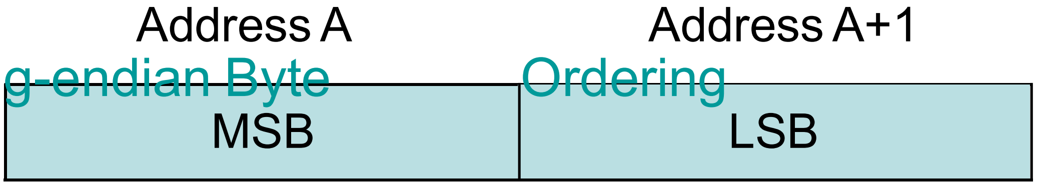
program

Two types of Byte Ordering

- – Little-endian Byte Ordering



- – Big-endian Byte Ordering



abstract

Socket function

connect function

bind function

listen function

accept function

fork and exec function

concurrent server close function

- getsockname and getpeername function

.

CONNECT FUNCTION:

Client program connect to server s by establishing a connection between an unnamed socket and the server listen socket.

```
#include <sys/socket.h>
 int connect(int sockfd, const struct sockaddr
  *servaddr, socklen_t addrlen);
```

- Returns : 0 if successful connect, -1 otherwise
- sockfd: integer, socket to be used in connection
- struct sockaddr: address of passive participant
- integer, sizeof(struct)

(If connect fails, the SYN_SENT socket is no longer

Connect function

Return error

- ETIMEOUT : no response from server
- RST : server process is not running
- EHOSTUNREACH : client‘s SYN unreachable from some intermediate router.

Error Return by Connect

If the client TCP receives no response to its SYN segment, ETIMEDOUT is returned.

If the server‘s response to the client‘s SYN is a reset (RST), this indicates that no process is waiting for connections on the server host at the port specified.

- Hard error
- ECONNREFUSED is returned to the client as soon as the RST is received.

Error Return by Connect

Three conditions that generate an RST are:

- When a SYN arrives for a port that has no listening server
- When TCP wants to abort an existing connection
- When TCP receives a segment for a connection that does not exist.

Error Return by Connect

ICMP destination unreachable received in response to client TCP‘s SYN (maybe due to transient routing problem), resend SYN timeout after 75 sec, returns EHOSTUNREACH or ENETUNREACH

connect Function: Three-Way Handshake

No bind before connect :The kernel chooses the source IP, if necessary, and an ephemeral port (for the client).

Hard error: RST received in response to client TCP's SYN (server not running) returns ECONNREFUSED

Soft error:

no response to client TCP's SYN, resend SYN, timeout after 75 sec (in 4.4BSD), returns ETIMEOUT

ICMP destination unreachable received in response to client TCP's SYN (maybe due to transient routing problem), retx SYN, timeout after 75 sec, returns EHOSTUNREACH)

The bind function

Assigns a local protocol address to a socket.

```
int bind(int sockid, const struct sockaddr *myaddr,
 socklen_t addrlen);
```

- status: error status, = -1 if bind failed, 0 if OK.
- sockid: integer, socket descriptor
- myaddr: struct sockaddr, the (IP) address and port of the machine (address usually set to INADDR_ANY – chooses a local address)
- addrlen: the size (in bytes) of the addr port structure

bind Function

Usually servers bind themselves to their well-known ports.

RPC servers let kernel choose ephemeral ports which are then registered with the *RPC port mapper*.

Normally, TCP client does not bind an IP address to its socket.

If a TCP server does not bind an IP address to its socket, the kernel uses the .destination IP addressof the client's SYN

LISTEN FUNCTION:

To accept incoming connections on a socket, a server program must create a queue to store pending requests.

```
#include <sys/socket.h>
int listen(int sockfd, int backlog);
```

Returns:0 if OK, -1 on error

==>This function is called only by a TCP server

The listen function converts an unconnected socket into a passive socket, indicating that the kernel should accept incoming connection requests directed to this socket.

backlog =>specify the maximum number of connections that the kernel should queue for this socket.

If the queues are full when client SYN arrives, TCP server ignore the SYN, it does not send RST.

listen function

- An incomplete connection queue, which contains an entry for each SYN that has arrived from a client for which the server is awaiting completion of the TCP three-way handshake
- A completed connection queue, which contains an entry for each client with whom the TCP three-way handshake has completed

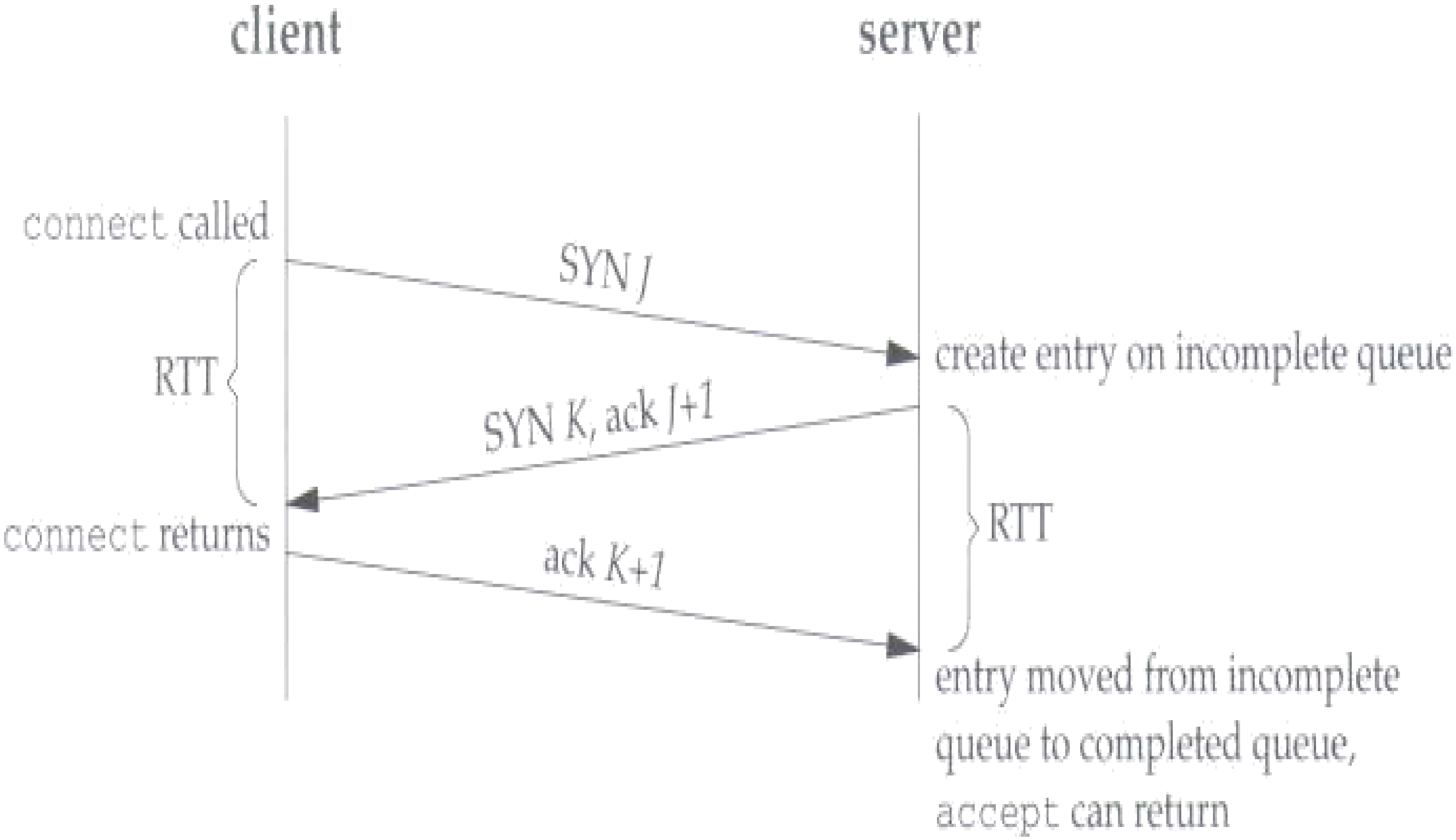


Figure 4.7 TCP three-way handshake and the two queues for a listening socket.

Figure 4.6 depicts these two queues for a given listening socket.

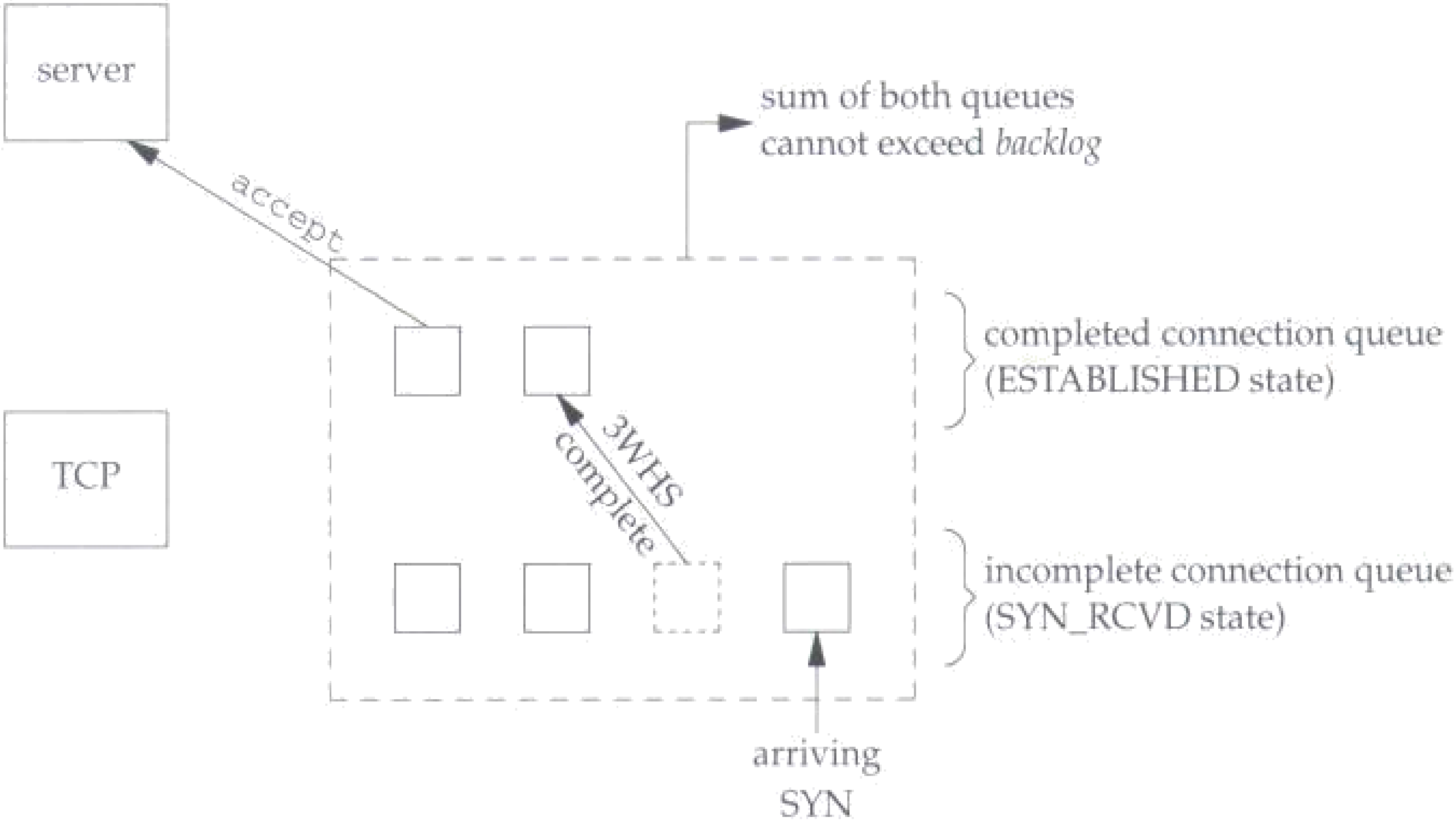

server
TCP
accept
sum of both queues cannot exceed *backlog*
completed connection queue (ESTABLISHED state)
3WHS complete
incomplete connection queue (SYN_RCVD state)
arriving SYN

Figure 4.6 The two queues maintained by TCP for a listening socket.

Backlog argument to the listen function has historically specified the maximum value for the sum of both queues

LISTEN FUNCTION:

Backlog argument to listen function has specified the maximum value for the *sum* of both queues

Berkeley derived : multipied by 1.5

Do not specify backlog of 0

What value should the application specify?

Allow Command line or an environment variable to override default.

Listen that allows an environment var to specify backlog

```
    Void Listen(int fd, int
   backlog) { char *ptr;
    if ((ptr=getenv(—LISTENQ‖))!=NULL)
                 backlog=atoi(ptr);
  if (listen(fd,backlog)<0)
          printf(—listen error‖);
}
```

listen Function

If the queues are full when a client SYN arrives, TCP ignores the arriving SYN : it does not send an RST.

Data that arrives after the three way handshake completes, but before the server call accept, should be queued by the server TCP, up to the size of the connected socket‘s receive buffer.

accept function

Once a server program has created and named a socket, it can wait for connections to be made to the socket by using the accept call.

```
#include <sys/socket.h>
int accept(int sockfd, struct sockaddr *cliaddr,
                                    socklen_t *addrlen);
```

Returns:nonnegative descriptor if OK, -1 on error

=> return the next completed connection from the front of the completed connection queue.

.If queue is empty, theprocess is put to sleep.

Return three values:

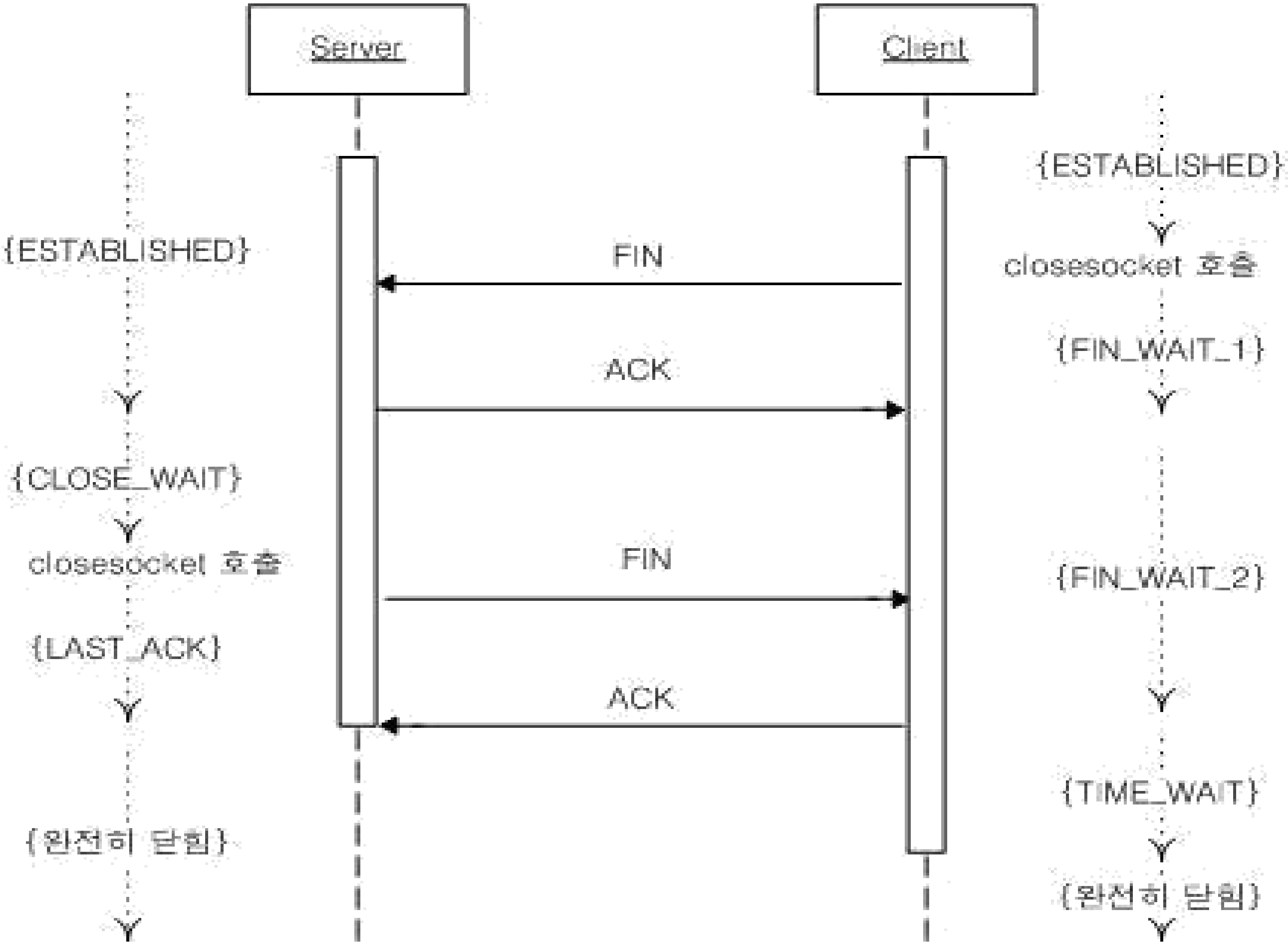
integer return code

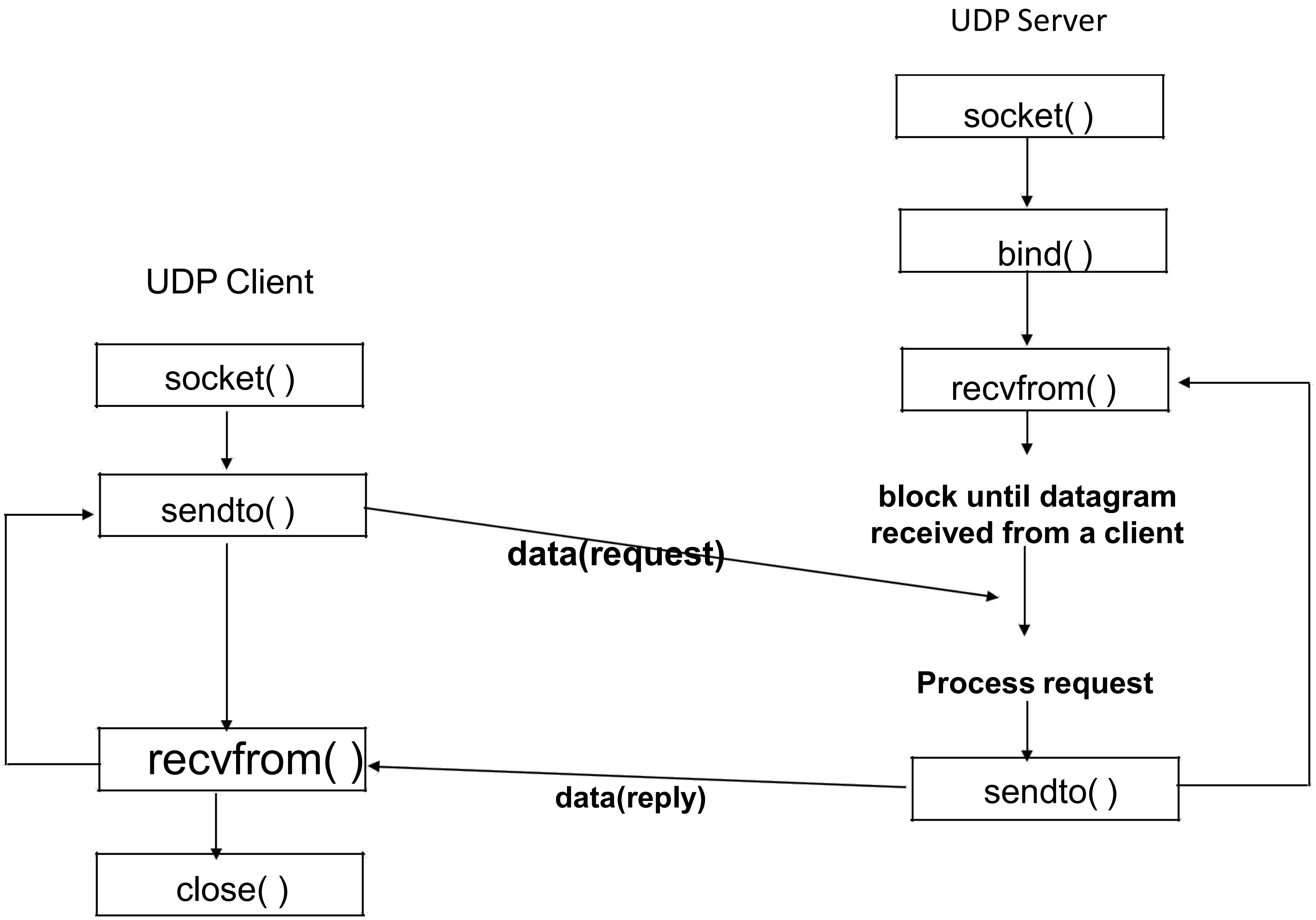
protocol address of the client process

size of this address :

This integer value contains the actual number of bytes stored by the kernel in the socket address structure.

If we are not interested in having the protocol address of the client returned, we set both *cliaddr* and *addrlen* to null pointers.


Server
Client
{ESTABLISHED}
{ESTABLISHED}
closesocket 호출
FIN
{FIN_WAIT_1}
ACK
{CLOSE_WAIT}
closesocket 호출
FIN
{FIN_WAIT_2}
{LAST_ACK}
ACK
{TIME_WAIT}
{완전히 닫힘}
{완전히 닫힘}



Socket functions for UDP client-server

recvfrom and sendto Functions

```
#include<sys/socket.h>
```

The recvfrom function fills in the socket address structure pointed to by from with the protocol address of who sent the datagram.

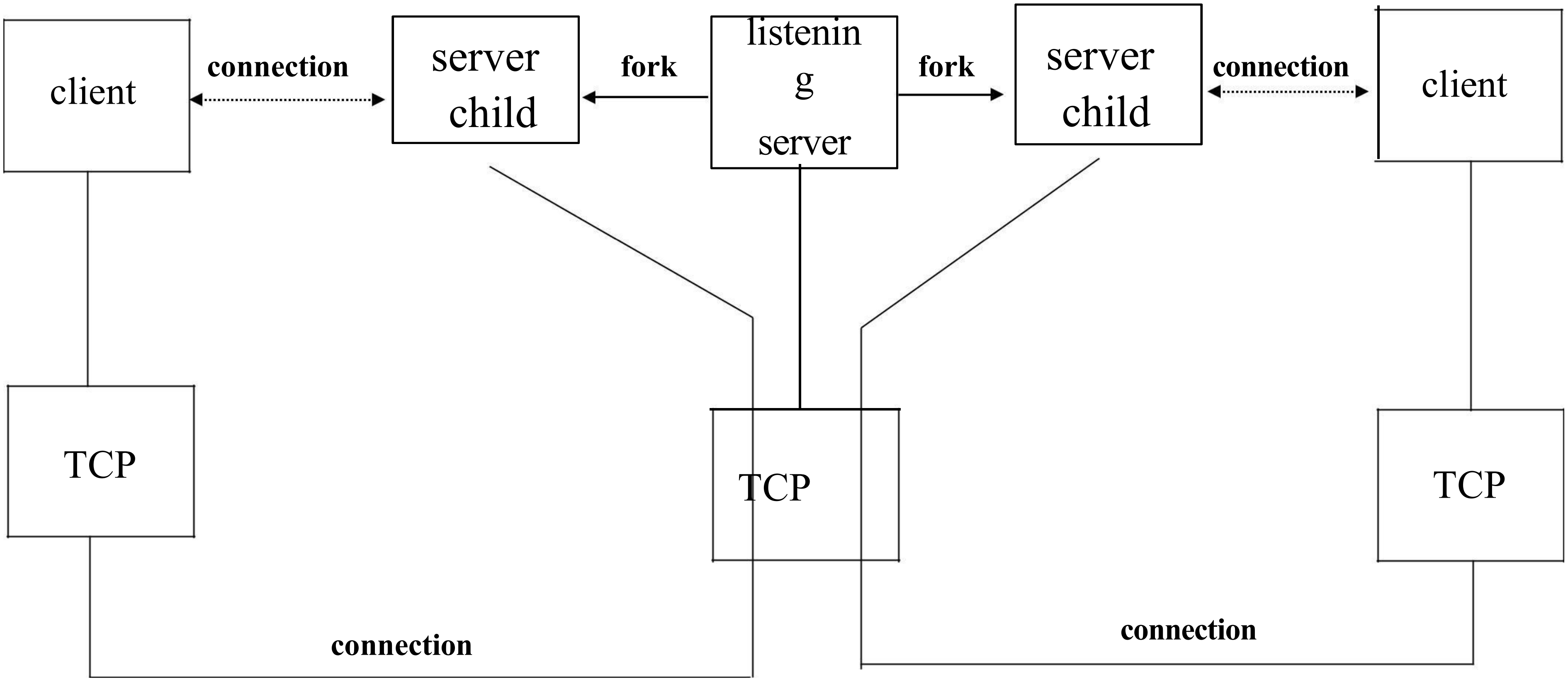
```
ssize_t recvfrom(int sockfd, void *buff, size_t nbyte, int flag,
                                    struct sockaddr *from, socklen_t *addrlen);


ssize_t sendto(int sockfd, const void *buff, size_t nbyte, int flag,
                                const struct sockaddr *to, socklen_t addrlen);
```

The to argument for sendto is a socket address structure containing the protocol address of where the data is to be sent.

The size of the socket address structure is specified by addrlen.

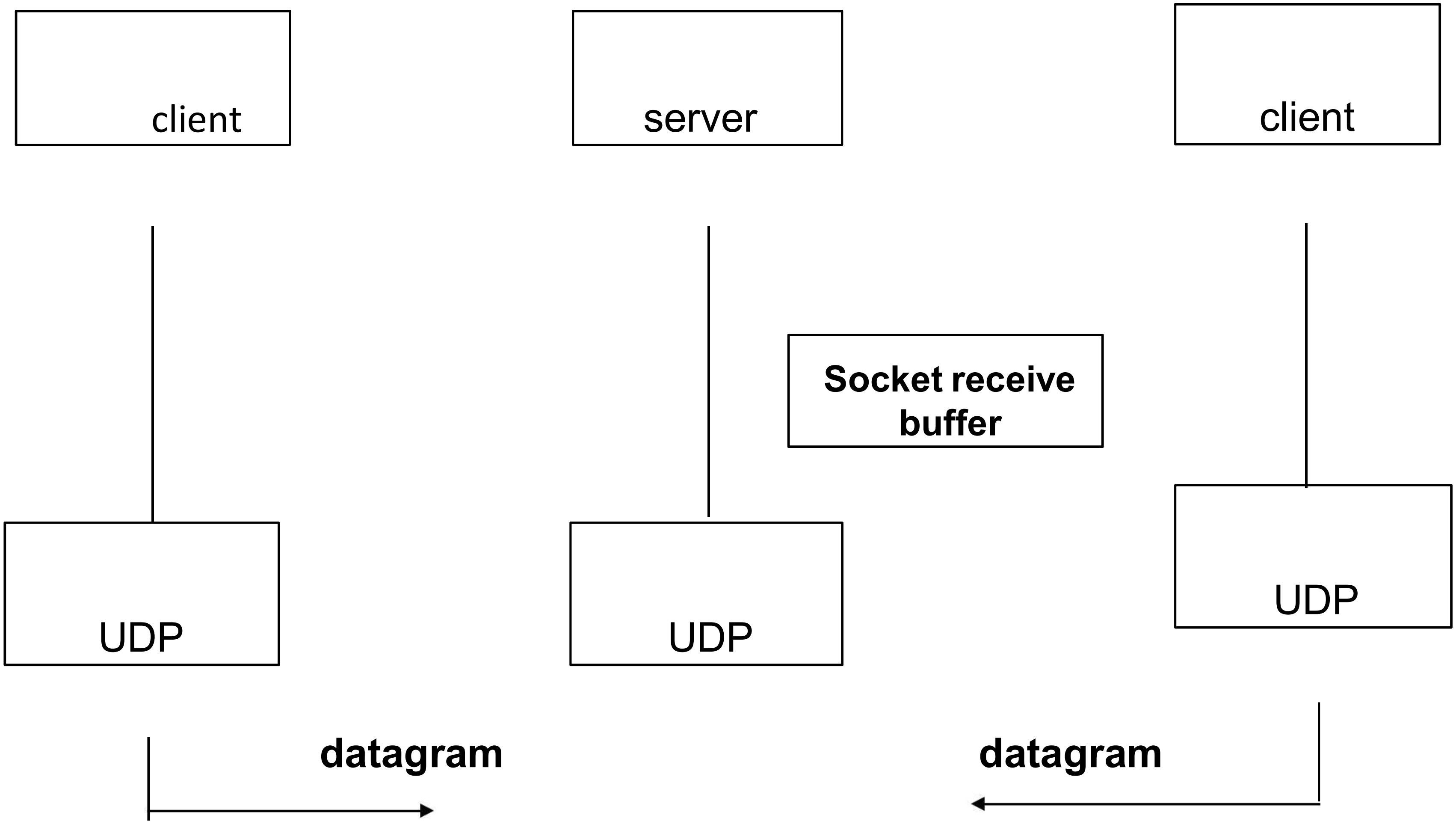
Both return: number of bytes read or written if OK,-1 on error

concurrent servers with TCP:



Summary of TCP client-server with two clients .

ITERATIVE SERVERS:



client

server

client

Socket receive buffer

UDP

UDP

UDP

datagram

datagram

Summary of UDP client-server with two clients

THE END