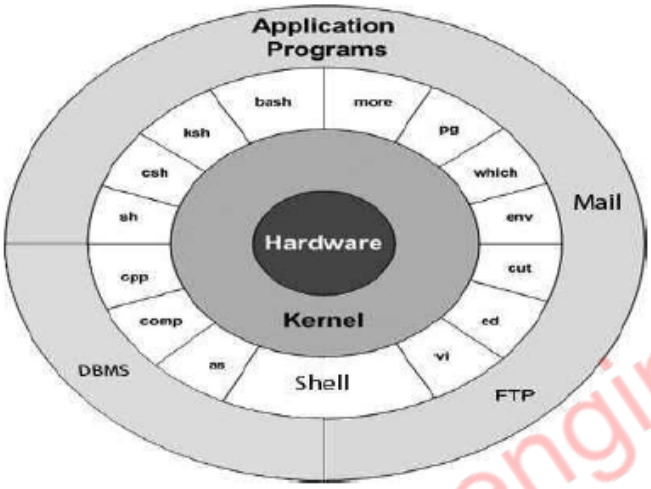
UNIX PROGRAMMING

1(A) - Explain with a figure ,the kernel and shell relationship in unix operating system.



Kernel: is the heart of UNIX system. It contains two basic parts of the OS: process control and resource management. All other components of the system call on the kernel to perform these services for them.

Shell: interface between Kernel and User. It functions as command interpreter i,e it receives and interprets the command from user and interacts with the hardware. There is only one kernel running on the system, there could be several shells in action- one for each user who is logged in. Shell has two major parts.

a. Interpreter: reads your commands and works with the kernel to execute them.

b. Shell Programming: is a programming capability that allows you to write a shell scripts.

A shell script is a file that contains the shell commands that perform a useful function. It is also known as shell program.

There are three standard shells used in UNIX today.

- Bourne shell: developed by steve bourne at AT&T labs, is the oldest.
- Bash(Bourne Again shell): An enhanced version of the Bash shell.
- C shell: developed in the Berkeley by Bill joy, Its commands look like C statements.
- Tesh: A compatible version of C shell.
- Korn shell: developed by David Korn, also of AT&T Labs is the newest and powerful.

1(B) - List and explain the salient features of Unix operating system.

FEATURES OF UNIX

Several features of UNIX have made it popular. Some of them are:

• **Portable:** UNIX can be installed on many hardware platforms. Its widespread use can be traced to the decision to develop it using the C language. Because C programs are easily moved from one hardware environment to another, it is relatively simple to port it to different environments.

• **Multiuser:** The UNIX design allows multiple users to concurrently share hardware and software

• **Multitasking:** UNIX allows a user to run more than one program at a time. In fact more than one program can be running in the background while a user is working foreground.

• **Networking:** While UNIX was developed to be an interactive, multiuser, multitasking system, networking is also incorporated into the heart of the operating system. Access to another system uses a standard communications protocol known as Transmission Control Protocol/Internet Protocol (TCP/IP).

• Organized File System: UNIX has a very organized file and directory system that allows users to organize and maintain files.

• Device Independence: UNIX treats input/output devices like ordinary files. Input or output to a program can be from any device or file.The source or destination for file input and output is easily controlled through a UNIX design feature called redirection.

• Utilities: UNIX provides a rich library of utilities that can be use to increase user productivity.

• Services: UNIX also includes the support utilities for system administration and control.

2(B) – With suitable example bring out the differences between absolute and relative pathnames.

RELATIVE AND ABSOLUTE PATHNAMES

THE dot(.) and double dots(..) notation to represent present and parent directories and their usage in relative pathnames

RELATIVE PATHNAMES

- Pathnames that don't begin with / specify locations relative to your current working directory.
- Uses either the current or parent directory as reference and specifies path relative to it.
- A relative pathname uses one of these cryptic symbols.
  - . (a single dot)-→ this represents the current directory.
  - .. (two dots)→this represents the parent directory

Command	Function
cd	Returns you to your login directory
cd ~	Also returns you to your login directory
cd /	Takes you to the entire system's root directory
cd /root	Takes you to the home directory of the root or superuser,account created at installation, you must be root user to access this directory.
cd /home	Takes you to the home directory where user login directories are usually stored
cd ..	Moves you up one directory
cd ~otheruser	Takes you to the otheruser's login directory
cd /dir/subdirfoo	Regardless of which directory you are in, the absolute path takes you directly to subdirfoo, a subdirectory of dir.

Ex .1: Assume the current directory is /home/kumar/progs/data/text, using cd .. will move one level up

$pwd
/home/kumar/progs/data/text
$ cd ..
$pwd
/home/kumar/progs/data

Ex 2 : To move two levels up

$pwd
/home/kumar/progs
$ cd ../..
$pwd
/home

Ex 3: My present location is /etc/samba and now I want to change directory to /etc.
Using relative path: $ **cd ..**
Using absolute path: $**cd /etc**

Ex 4: My present location is /var/ftp/ and I want to change the location to /var/log
Using relative path: **cd ../log**
Using absolute path: **cd /var/log**

Ex 5: My present location is /etc/lvm and I want to change my location to /opt/oradba
Using relative path: **cd ../../opt/oradba**
Using absolute path: **cd /opt/oradba**

ABSOLUTE PATHNAMES:

- If the first character of a pathname is / the files location must be determined with respect to root(/) . Such a pathname is called absolute pathname.

  cat /home/kumar

- When you have more than one / in a pathname for such / you have to descend one level in the file system. Thus Kumar is one level below home and two levels below root.
- When you specify a file y using frontslashes to demarcate the various levels,you have a mechanism of identifying a file uniquely.No two files in a UNIX system can have same absolute pathnames.
- When you specify the date command, the system has to locate the file date from a list of directories specified in the PATH variable and then execute it.
- However if you know the location of a command in prior, for example date is usually located in /bin or /usr/bin . Use absolute pathname i,e precede its name with complete path

  $/bin/date

  For example if you need to execute program **less** residing in /usr/local/bin you need to enter the absolute pathname

  $/usr/local/bin/less

2(C) - Explain the basic file categories in Unix operating system?

a.Ordinary Files or regular files

It contains only data as a stream of characters.

An ordinary files itself divided into 2 types

Text file: contains only printable characters, and you can often view the contents and make sense out of them. All C and Java files are example of text file. A text file contains lines of characters where every line is terminated with the newline character, also known as **linefeed** (LF) when you press Enter while inserting text, the LF character is appended to every line. You won't see this character normally, but there is command (od) which can make it visible.

Binary file: it contains both printable and unprintable characters that cover the entire ASCII range(0 to 255).most UNIX commands are example of binary files.

b. Directory files

i. Contains no data, but keeps some details of the files and subdirectories that it contains.

ii. A directory file contains an entry for every file and sub directory that it houses. Each entry has two components

- The filename
- A unique Identification number for the file or directory(called the inode number)

iii. A directory contains the filename but not the contents of file.

iv. When you create or remove a file the kernel automatically updates its corresponding directory by adding or removing the enter i.e inode number associated with that file.

c. Device files

i. Used to represent a real physical device such as a printer, tape drive or terminal, used for Input/Ouput (I/O) operations

ii. Unix considers any device attached to the system to be a file - including your terminal:

iii. By default, a command treats your terminal as the standard input file (stdin) from which to read its input

iv. Your terminal is also treated as the standard output file (stdout) to which a command's output is sent.

v. stdin and stdout will be discussed in more detail later

3(A) – Which command is for is used for listing file attributes ?explain the significance of each field in the attributes?

The ls command with options

ls –l: LISTING FILE ATTRIBUTES

ls command is used to obtain a list of all filenames in the current directory. The output in UNIX lingo is often referred to as the listing. Sometimes we combine this option with other options for displaying other attributes, or ordering the list in a different sequence. ls look up the file's inode to fetch its attributes. It lists seven attributes of all files in the current directory and they are:

File type and Permissions

The file type and its permissions*:* The first column shows the type and permissions associated with each file.The first character in this column is mostly a – which indicates that the file is an ordinary one. In unix, file system has three types of permissions- read, write and execute.

Links: The second column indicates the number of links associated with the file. This is actually the number of filenames maintained by the system of that file.

Ownership: The third column shows the owner of files. The owner has full authority to tamper with files content and permissions. Similarly, you can create, modify or remove files in a directory if you are the owner of the directory.

Group ownership: The fourth column represents the group owner of the file. When opening a user account, the system admin also assigns the user to some group. The concept of a group of users also owning a file has acquired importance today as group members often need to work on the same file.

File size: The fifth column shows the size of the file in bytes. The important thing to remember here is that it only a character count of the file and not a measure of the disk space that it occupies.

Last modification time: The sixth, seventh and eighth columns indicate the last modification time of the file, which is stored to the nearest second. A file is said to be modified only if its content have changed in any way.If the file is less than a year old since its last modification time, the year won't be displayed.

Filename: The last column displays the filename arranged in ASCII collating sequence.

3(B) - What are file permissions? Explain the use of chmod to change file permissions using both absolute and relative methods?

FILE PERMISSIONS

- UNIX has a simple and well defined system of assigning permissions to files.
- Lets issue the ls –l command once again to view the permissions of a few lines .

```
$ls   -l    chap02    dept.lst      dateval.sh
-rwxr-xr--   1   kumar     metal     25000        May 10  19:21       chap02
-rwxr-xr-x   1   kumar     metal     890           Jan 10  23:17       dept.lst
-rw-rw-rw-       1    kumar     metal     84            Feb  18   12:20       dateval.sh
```

Consider the first column.

- rwx r-x r—

Each group here represents the category and contain three slots representing the read, write and execute permissions of the file.

r indicates the read permission; **w** indicates write permission; **x** indicates execute permission

- (hyphen) indicates the absence of the corresponding permission.

In the above example, the file permissions of chap02 file is

-	rwx	r-x	r - -
File	**owner/user**	**group**	**others**

First group(rwx) has all the three permissions.

- The file is readable, writable and executable by the **owner** of the file, kumar.
- The third column shows the owner of the file.
- The first permissions group applies to kumar.
- You have to login with the name kumar for the privileges to apply to you.

Second group(r-x):

- has a hyphen in the middle slot, which indicates the absence of write permissions by the **group** owner of the file.
- The group owner is metal and all users belonging to group metal has only read and execute permissions.

CHANGING FILE PERMISSIONS:RELATIVE AND ABSOLUTE PERMISSIONS

chmod command.

A file or a directory is created with a default set of permissions, which can be determined by umas

us assume that the file permission for the created file is -rw-r--r--. Using chmod command, we can c

the file permissions and allow the owner to execute his file. The command can be used in two way

In a relative manner by specifying the changes to the current permissions

In an absolute manner by specifying the final permissions

RELATIVE PERMISSIONS

chmod only changes the permissions specified in the command line and leaves the other permissic

unchanged.

Its syntax is:

chmod category operation permission filename(s)

chmod takes an expression as its argument which contains:

user category (user, group, others)

operation to be performed (assign or remove a permission)

type of permission (read, write, execute)

The below shows the abbreviations used by **chmod** command

Category	**operation**	**permission**
u - user	+ assign	r - read
g – group	- remove	w - write
o - others	= absolute	x - execute
a - all (ugo)		

Ex 1:

$ls –l xstart

-rw-r--r-- 1 kumar metal 1906 sep 23:38 xstart

Here user is having the only read and execute permission .

Using relative file permission need to add the execute permission to user

chmod	category	operation(+,-)	permission	filename.
$chmod	u	+	x	xstart

$chmod u+x xstart

$ ls –l xstart

-rwxr--r-- 1 kumar metal 1906 sep 23:38 xstart

After executing the **chmod** command, the command assigns (+) execute (x) permission to the user (u), other permissions remain unchanged.

Ex 2: To remove execute permission from all and assign read permission to group and others

Schmod a-x, go+r xstart /*to remove execute permission from all(a)ie user, group, others

/*to assign read permission to group and others (go+r)

Ex 3: To assign write and execute permissions for others.

Schmod o+wx xstart

ABSOLUTE PERMISSIONS

A string of three octal digits is used as an expression. The permission can be represented by one octal digit for each category. For each category, we add octal digits. If we represent the permissions of each category by one octal digit, this is how the permission can be represented:

- Read permission – 4 (octal 100)
- Write permission – 2 (octal 010)
- Execute permission – 1 (octal 001)

2	- w -	write only
3	- w x	write and execute
4	r - -	read only
5	r - x	read and execute
6	r w -	read and write
7	r w x	read, write and execute

We have three categories and three permissions for each category, so three octal digits can des file's permissions completely. The most significant digit represents user and the least one rep others. chmod can use this three-digit string as the expression.

Ex 1: To assign read,write permissions to all .Using **relative** permission, we have,

$chmod a+rw xstart

Using **absolute** permission, we have,

$chmod 666 xstart /* 6 for r-w

/* first digit 6 for user, second 6 for group and third 6 for ot

Ex 2:

To assign read and write for user and remove write, execute permissions from group and others

- Here to assign rw- corresponds to digit 6
- Remove write , execute permissions is nothing but assigning only read option to group ar ers
- Only read permission is r—corresponds to 4

$chmod 644 xstart

Ex 3:

To assign all permissions to the owner, read and write to group and only execute for others.

$chmod 761 xstart

Ex 4

To assign all permissions to all categories.

$chmod 777 xstart

3(C) - Explain grep command?List its options with its significance.

The grep,egrep

grep(globally search regular expression and print)

Unix has a special family of commands for handling search requirements and the principal member of the family is the **grep command**.

grep scans its input for a pattern and displays lines containing the pattern, the line numbers or filename where the pattern occurs.

Syntax:

grep options pattern filename(s)

grep searches for pattern in one or more filename or the standard input if no filename is specified. The first argument(barring the options) is the pattern and the remaining arguments are filenames.

Consider three files emp.lst, emp1.lst, emp2.lst

$cat emp.lst

```
101|sharma|general manager|sales|10/09/61|6700
102|kumar|director|Sales|09/09/63|7700
103|aggarwal|manager|sales|03/05/70|5000
104|rajesh|manager|marketing|12/04/72|5800
105|adarsh|executive|sales|07/09/57|5300
```

$cat emp1.lst

```
201|anil|director|sales|05/01/59|5000
202|sunil|director|marketing|12/06/51|6000
203|gupta|director|production|09/08/55|7000
```

$cat emp2.lst

```
301|anil Agarwal|manager|sales
302|sudhir agarwal|director|production
303|rajath Agarwal|manager|sales
304|v.k.agrawal|director|marketing
305|v.s.agrawal|deputy manager|sales
306|agrawal|executive|sales
401|sumith|general manager|marketing
402|sudhir agarwal|director|production
```

```
$ grep "sales" emp.lst
101|sharma|general manager|sales|10/09/61|6700
103|aggarwal|manager|sales|03/05/70|5000
105|adarsh|executive|sales|07/09/57|5300
```

Ex 2: To search the pattern director from 2 files i,e emp.lst and emp2.lst

```
grep "director" emp.lst emp2.lst
emp.lst:102|kumar|director|Sales|09/09/63|7700
emp2.lst:302|sudhir agarwal|director|production
emp2.lst:304|v.k.agrawal|director|marketing
emp2.lst:402|sudhir agarwal|director|production
```

grep along with options

Table 13.1 Options Used by **grep**

Option	*Significance*
`-i`	Ignores case for matching
`-v`	Doesn't display lines matching expression
`-n`	Displays line numbers along with lines
`-c`	Displays count of number of occurrences
`-l`	Displays list of filenames only
`-e` *exp*	Specifies expression with this option. Can use multiple times. Also used for matching expression beginning with a hyphen.
`-x`	Matches pattern with entire line (doesn't match embedded patterns)
`-f` *file*	Takes patterns from *file*, one per line
`-E`	Treats pattern as an extended regular expression (ERE)
`-F`	Matches multiple fixed strings (in **fgrep**-style)

4(A) - Explain the concept of escaping and quoting with suitable example?

ESCAPING and QUOTING

Escaping: providing a \(backslash character) before the wild card to remove or escape its special meaning.

Quoting: enclosing the wild card or even the entire pattern within quotes ('chap* '). Anyting within the quotes are left alone by the shell and not interpreted.

ESCAPING

- Placing a \ immediately before a metacharacter turns off its special meaning.
- For instance * , matches * itself. Its special meaning of matching zero or more occurrencesof character is turned off.

Ex 1:

Srm chap*

removes all the filenames starting with chap. Chap, chap01,chap02 and chap03 are removed.

Srm chap* // * metacharacter meaning is turned off

removes the filename with chap* /*name of the file itself is chap*.

- If there are files with names chap01, chap02,chap03.
- To list the filenames starting with chap0

S ls chap0[1-3]
Output:
chap01
chap02
chap03

Ex 3:
To match the file named as chap0[1-3]
S ls chap0\[1-3\]
Output:
chap0[1-3]

Escaping the space: To remove the file My document.doc, which has space embedded,
Srm My\ document.doc

Escaping the newline character.
Secho -e "The newline charcter is \n Enter the command"
Output:
The newline character /n – newline character is interpreted and cursor moves to next line
Enter the command

S echo "the newline character is \\n enter the command"
Output:

the newline character is \\n enter the command /* here newline character interpretation is turned off and the \n is printed as it is*/

Escaping the \ itself
**Secho **
Output
\

QUOTING:

- This is the another way of turning off the meaning of metacharacter.
- When a command argument is enclosed within quotes, the meaning of all enclosed special characters are turned off

$rm 'chap*' // * metacharacter meaning is turned off

removes the filename with chap* /*name of the file itself is chap*.

$rm "My\ document.doc" /* To remove the file My document.doc, which has space embedded.

$echo '\'

output

****

4(C) – What are wild card characters? Explain shell wild card characters with example?

WILD CARDS

The metacharacters taht are used to construct the generalized pattern for matching filenames belong to the category called **wild cards.**

The * and ?

- **The metacharacter** * is one of the characters of the shell wild card set. It matches any number of characters including none.
- For example : to match filenames chap chap01 chap02 chap03 chap04

  $ ls chap*

  Output : // the * matches all strings along with none

  chap
  chap01
  chap02
  chap03
  chap04

- **The metacharacter ?** matches a single character
  For example: to match filenames chapx chapy chapz
  $ ls chap?
  Output: //the ? replaces single character i,e ? is replaced by x, y,z
  chapx
  chapy
  chapz

Matching the dot(.)

- The **.** dot metachacter can be used to match all the hidden files in your directory.
- Example: To list all hidden files in your directory having atleast three characters after the dot.
  $ ls .???*
  Output:
  .bash_profile
  .exrc
  .netscape
  .profile

The character class [ ]

- This class comprises a set of charcters enclosed by the rectangular brackets [and],but it matches only a single character in the class.
- The **pattern [abcd]** is a character class and it matches a single character a, b, c or d
- For example to match chap01 chap02 chap03 chap04
  $ ls chap0[1234]
  output:
  chap01
  chap02
  chap03
  chap04

Negating the character class(!)

- It is used to reverse the matching criteria.
- For example:
  To match all the filenames with single character extensions but not .c or .o files
  $ ls *.[!co]
  *** to match any filename**
  . extension
  ! except
  [!co]---> .c extension or .o extension
- To match filenames that does not begin with a digit
  $ ls [!0-9]*

- To match filename with 3 character that does not begin with an Upper Case letter
  $ ls [!A-Z]??

4(B) - Explain three standard files supported by unix? Explain about special files used for output redirection?

standard files.

- With input files, we call it input redirection;
- With output file, we call it output redirection
- With the error file, we call it error redirection.

Standard input: The file representing the input, which is connected to the keyboard

Standard output: The file representing the output, which is connected to the display.

Standard error: the file representing the error messages that emanate from the command or shell. This is also connected to display.

Each of the three standard files are represented by a number called a **file descriptor.**

The first three slots are generally allocated to three standard streams in this manner

0: standard input

1: standard output

2: standard error

STANDARD INPUT

This file is indeed special

- The keyboard, the default source
- **a file using redirection with the < symbol**
- another program using the pipeline

- The input redirection operator is less than character (<).
- When you use wc without an argument , it prompts you to provide the input from standard input keyboard

  $ wc

  Unix is a multiuser multitasking OS

  [ctrl-d]

- When wc is used with argument. Filename is passed as an argument i,e wc takes the input from the filename we have specified

  For example: Create a file with name sample.txt

  $ vi sample.txt

  Unix is a multiuser multitasking OS

[ctrl-d]

- When wc is used with argument. Filename is passed as an argument i,e wc takes the input from the filename we have specified

For example: Create a file with name sample.txt
$ vi sample.txt
Unix is a multiuser multitasking OS

:wq /*saving and quitting from the file

$ wc < sample.txt /*wc command takes input from the file sample.txt
output
1 6 36 /* count of characters, words, lines of file sample.txt

STANDARD OUTPUT

- All commands displaying the output on the terminal actually write to the standard output file as a stream of characters and not directly to the terminal as such.
- There are three possible destinations of this stream
- The terminal, the default destination
- **A file using the redirection symbol > and >>**
- As input to another program using a pipeline
- There are two basic redirection operators for standard output.

a. greater than character(>):

If you want the file to contain only the output from this execution of the command, you can use greater than token.
Ex: consider the file **sample.txt**
$ cat sample.txt /* to display the content of sample.txt
Unix is a multiuser multitasking OS
$ wc sample.txt > newfile
> symbol redirects the output of wc command to a file named newfile
The output is now stored in newfile

S cat newfile /* To view the content of newfile

1 6 37

b Two greater than characters>> (append)

If you want the output of the command to be appended without overwriting the existing content.

$who >> newfile
The output of who command is appended to newfile without overwriting the existing content.
$cat newfile
1 6 37 /* output of wc command executed previously

root console aug 1 07:51 (:0) /* output of who command
kumar pts/10 aug 1 02:51 (:0)
sharma pts/6 aug 1 03:51 (:0)

STANDARD ERROR

When you enter an incorrect command or try to open a non existent file, certain diagnostic messages show up on the screen. This is the standard error stream whose default destination is the terminal.

Standard output and error on monitor

Ex 1: Consider two files file1 and file2 , where file1 exist and file2 do not exist

```
S ls –l   file1 file2
-rwxr—r--    1      gilberg          staff   1234   oct     file1
Cannot access file2: no such file or directory                /*error because file2 do not exist
```

Ex 2: To redirect standard output to same file

Sls -l file1 file2 1>filelist 2>filelist

The 1st argument is file1 and 2nd argument is file2.

The output and errors are sent to file named filelist

Scat filelist

```
-rwxr—r--    1      gilberg          staff   1234   oct     file1
```


5(B) - With neat sketch, explain memory layout of C program.

MEMORY LAYOUT OF A C PROGRAM

Historically, a C program has been composed of the following pieces:

- **Text segment**, the machine instructions that the CPU executes. Usually, the text segment is sharable so that only a single copy needs to be in memory for frequently executed programs, such as text editors, the C compiler, the shells, and so on. Also, the text segment is often read-only, to prevent a program from accidentally modifying its instructions.
- **Initialized data segment**, usually called simply the data segment, containing variables that are specifically initialized in the program. For example, the C declaration

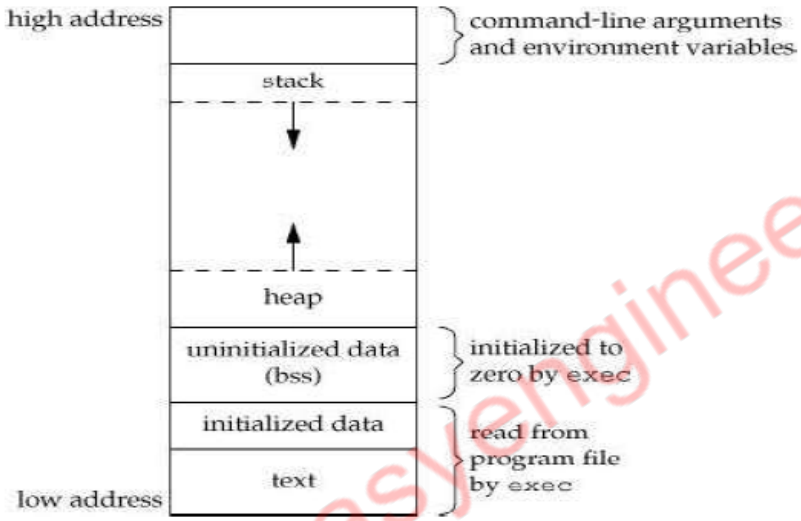
  int maxcount = 99;

  appearing outside any function causes this variable to be stored in the initialized data segment with its initial value.
  - **Uninitialized data segment**, often called the "bss" segment, named after an ancient assembler operator that stood for "block started by symbol." Data in this segment is initialized by the kernel to arithmetic 0 or null pointers before the program starts executing. The C declaration

    long sum[1000];

appearing outside any function causes this variable to be stored in the uninitialized data segment.

- **Stack**, where automatic variables are stored, along with information that is saved each time a function is called. Each time a function is called, the address of where to return to and certain information about the caller's environment, such as some of the machine registers, are saved on the stack. The newly called function then allocates room on the stack for its automatic and temporary variables. This is how recursive functions in C can work. Each time a recursive function calls itself, a new stack frame is used, so one set of variables doesn't interfere with the variables from another instance of the function.
- **Heap**, where dynamic memory allocation usually takes place. Historically, the heap has been located between the uninitialized data and the stack.



6(A) - With related data structures explain UNIX kernel support for a process.

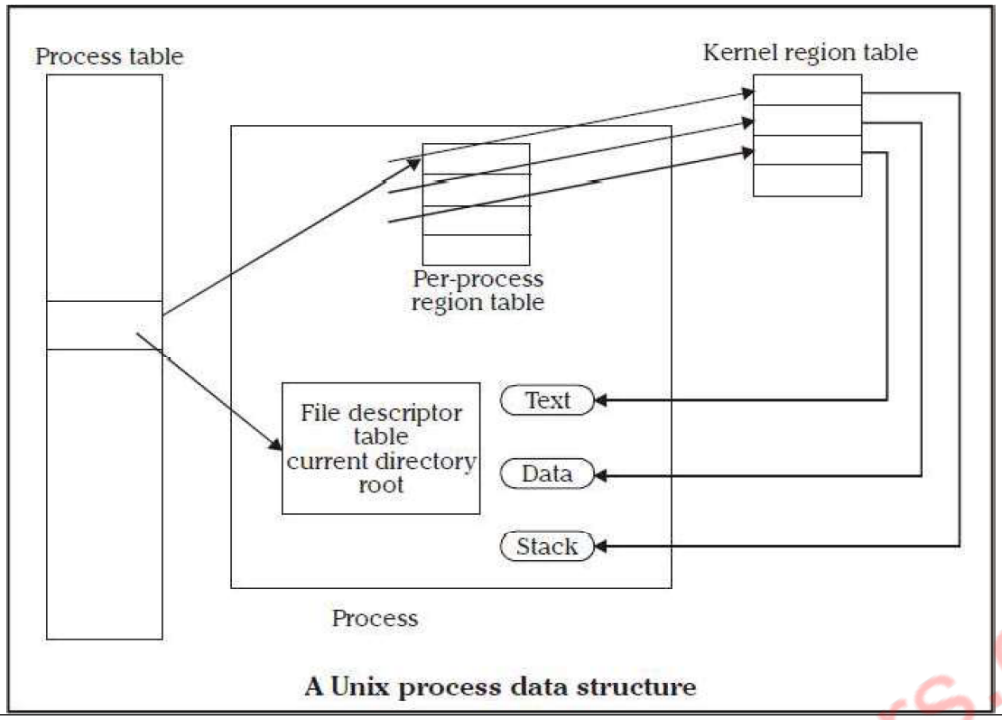
UNIX KERNEL SUPPORT FOR PROCESS

The data structure and execution of processes are dependent on operating system implementation.

A UNIX process consists minimally of a text segment, a data segment and a stack segment. A segment is an area of memory that is managed by the system as a unit.

- A text segment consists of the program text in machine executable instruction code format.
- The data segment contains static and global variables and their corresponding data.
- A stack segment contains runtime variables and the return addresses of all active functions for a process.

UNIX kernel has a process table that keeps track of all active process present in the system. Some of these processes belongs to the kernel and are called as "system process". Every entry in the process table contains pointers to the text, data and the stack segments and also to U-area of a process. U-area of a process is an extension of the process table entry and contains other process specific data such as the file descriptor table, current root and working directory inode numbers and set of system imposed process limits.



A Unix process data structure

All processes in UNIX system expect the process that is created by the system boot code, are created by the fork system call. After the fork system call, once the child process is created, both the parent and child processes resumes execution. When a process is created by fork, it contains duplicated copies of the text, data and stack segments of its parent as shown in the Figure below. Also it has a file descriptor table, which contains reference to the same opened files as the parent, such that they both share the same file pointer to each opened files.

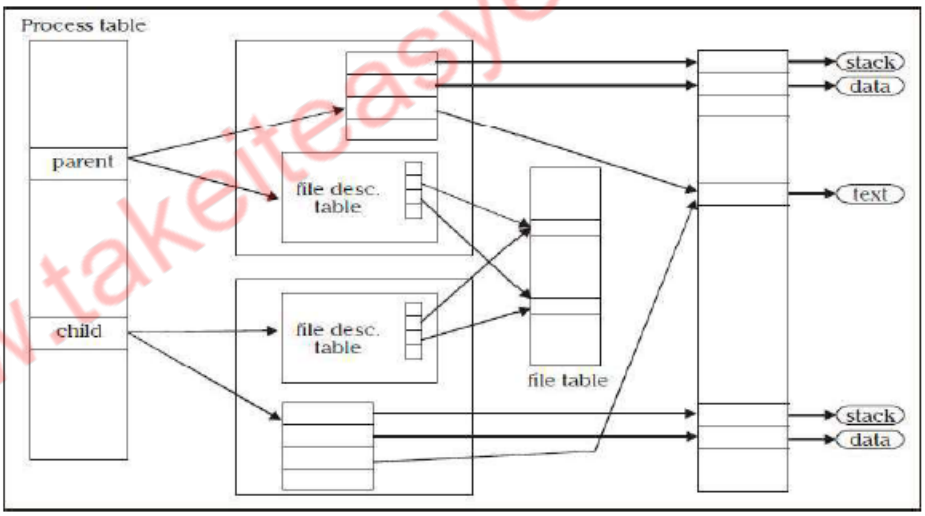


Figure: Parent & child relationship after fork

The process will be assigned with attributes, which are either inherited from its parent or will be set by the kernel.

- **A real user identification number (rUID)**: the user ID of a user who created the parent process.
- **A real group identification number (rGID)**: the group ID of a user who created that parent process.
- **An effective user identification number (eUID)**: this allows the process to access and create files with the same privileges as the program file owner.
- **An effective group identification number (eGID)**: this allows the process to access and create files with the same privileges as the group to which the program file belongs.
- **Saved set-UID and saved set-GID**: these are the assigned eUID and eGID of the process respectively.
- **Process group identification number (PGID) and session identification number (SID):** these identify the process group and session of which the process is member.
- **Supplementary group identification numbers:** this is a set of additional group IDs for a user who created the process.

- **Current directory:** this is the reference (inode number) to a working directory file.
- **Root directory**: this is the reference to a root directory.
- **Signal handling**: the signal handling settings.
- **Signal mask**: a signal mask that specifies which signals are to be blocked.
- **Unmask**: a file mode mask that is used in creation of files to specify which accession rights should be taken out.
- **Nice value**: the process scheduling priority value.
- **Controlling terminal**: the controlling terminal of the process.

In addition to the above attributes, the following attributes are different between the parent and child processes:

- **Process identification number (PID)**: an integer identification number that is unique per process in an entire operating system.
- **Parent process identification number (PPID)**: the parent process PID.
- **Pending signals**: the set of signals that are pending delivery to the parent process.
- **Alarm clock time**: the process alarm clock time is reset to zero in the child process.
- **File locks**: the set of file locks owned by the parent process is not inherited by the chid process.

fork and ***exec*** are commonly used together to spawn a sub-process to execute a different program. The advantages of this method are:

- A process can create multiple processes to execute multiple programs concurrently.
- Because each child process executes in its own virtual address space, the parent process is not affected by the execution status of its child process.

6(B) - What do you mean by fork and vfork functions. Explain both functions with example programs.

fork FUNCTION

An existing process can create a new one by calling the forkfunction.

```
#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 forkis 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.
- The reason the child's process ID is returned to the parent is that a process can have more than one child, and there is no function that allows a process to obtain the process IDs of its children.
- The reason fork returns 0 to the child is that a process can have only a single parent, and the child can always call getppid to obtain the process ID of its parent. (Process ID 0 is reserved for use by the kernel, so it's not possible for 0 to be the process ID of a child.)

- Both the child and the parent continue executing with the instruction that follows the call to fork.
- The child is a copy of the parent.
- For example, the child gets a copy of the parent's data space, heap, and stack.
- Note that this is a copy for the child; the parent and the child do not share these portions of memory.
- The parent and the child share the text segment .

Example programs:

Program 1

/* Program to demonstrate fork function Program name – fork1.c */

```
#include<sys/types.h>
#include<unistd.h>
int main( )
{
        fork( );
        printf("\n hello USP");
}
```

Output :

$ cc fork1.c

```
$    ./a.out
hello USP
hello USP
```

Note : The statement hello USP is executed twice as both the child and parent have executed that instruction.

vfork FUNCTION

- ✓ The function vforkhas the same calling sequence and same return values as fork.
- ✓ The vfork function is intended to create a new process when the purpose of the new process is to exec a new program.
- ✓ 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; the child simply calls exec (or exit) right after the vfork.
- ✓ Instead, while the child is running and until it calls either exec or exit, the child runs in the address space of the parent. This optimization provides an efficiency gain on some paged virtual-memory implementations of the UNIX System.
- ✓ Another difference between the two functions is that vfork guarantees that the child runs first, until the child calls execor exit. When the child calls either of these functions, the parent resumes.

Example of vforkfunction

```
#include "apue.h"
int        glob = 6;          /* external variable in initialized data */

int main(void)
{
    int        var;           /* automatic variable on the stack */
    pid_t      pid;

    var = 88;
    printf("before vfork\n");      /* we don't flush stdio */ if
    ((pid = vfork()) < 0) {
        err_sys("vfork error");
    } else if (pid == 0) {         /* child */
        glob++;                    /* modify parent's variables */
        var++;
        _exit(0);                  /* child terminates */
    }
    /*
     * Parent continues here.
     */
    printf("pid = %d, glob = %d, var = %d\n", getpid(), glob, var);
    exit(0);
}
```

7(A) - What are Pipes? Explain different ways to view a half-duplex pipe. Write a program to send data from parent process to child process using pipes.

PIPES

Pipes are the oldest form of UNIX System IPC. Pipes have two limitations.

- Historically, they have been half duplex (i.e., data flows in only one direction).
- Pipes can be used only between processes that have a common ancestor. Normally, a pipe is created by a process, that process calls fork, and the pipe is used between the parent and the child.

A pipe is created by calling the pipefunction.

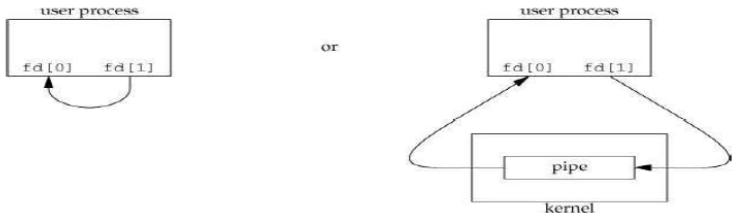
```
#include <unistd.h>

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].

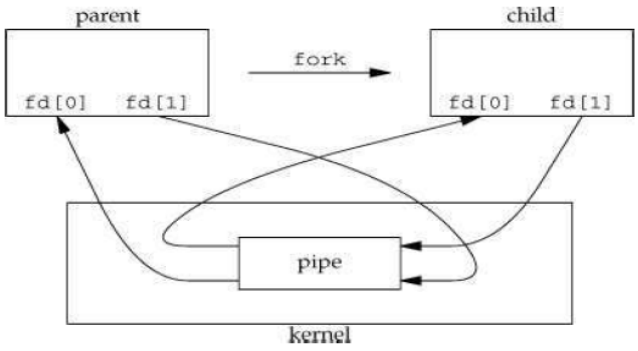
Two ways to picture a half-duplex pipe are shown in Figure 15.2. The left half of the figure shows the two ends of the pipe connected in a single process. The right half of the figure emphasizes that the data in the pipe flows through the kernel.

Figure 15.2. Two ways to view a half-duplex pipe



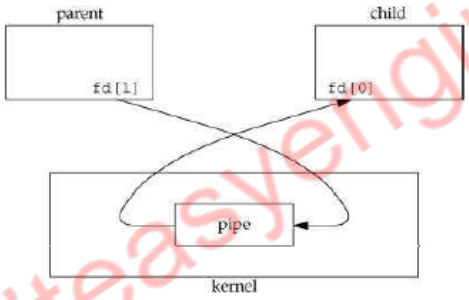
A pipe in a single process is next to useless. Normally, the process that calls pipe then calls fork, creating an IPC channel from the parent to the child or vice versa. Figure 15.3 shows this scenario.

Figure 15.3 Half-duplex pipe after a fork



What happens after the fork depends on which direction of data flow we want. For a pipe from the parent to the child, the parent closes the read end of the pipe (fd[0]), and the child closes the write end (fd[1]). Figure 15.4 shows the resulting arrangement of descriptors.

Figure 15.4. Pipe from parent to child



For a pipe from the child to the parent, the parent closes fd[1], and the child closes fd[0]. When one end of a pipe is closed, the following two rules apply.

- If we read from a pipe whose write end has been closed, read returns 0 to indicate an end of file after all the data has been read.
- If we write to a pipe whose read end has been closed, the signal SIGPIPEis generated. If we either ignore the signal or catch it and return from the signal handler, writereturns 1 with errnoset to EPIPE.

PROGRAM: shows the code to create a pipe between a parent and its child and to send data down the pipe.

```
#include "apue.h"

int
main(void
)
{
    int     n;
    int     fd[2];
    pid_t   pid;
    char    line[MAXLINE];

    if (pipe(fd) < 0) err_sys("pipe
```

```
            error");
        if ((pid = fork()) < 0) {
            err_sys("fork error");
        } else if (pid > 0) {                /* parent */
            close(fd[0]);
            write(fd[1], "hello world\n", 12);
        } else {                             /* child */
            close(fd[1]);
            n = read(fd[0], line, MAXLINE);
            write(STDOUT_FILENO, line,
            n);
        }
        exit(0);
}
```

7(B) - What is fifo? With a neat diagram explain the client server communication using fifo?

FIFOs

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

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

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

Returns: 0 if OK, 1 on error

Once we have used mkfifo to create a FIFO, we open it using open. When we open a FIFO, the nonblocking flag (O_NONBLOCK) affects what happens.

- In the normal case (O_NONBLOCK not specified), an open for read-only blocks until some other process opens the FIFO for writing. Similarly, an open for write-only blocks until some other process opens the FIFO for reading.
- If O_NONBLOCKis specified, an openfor read-only returns immediately. But an openfor write-only returns 1 with errnoset to ENXIOif no process has the FIFO open for reading.

There are two 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 as rendezvous points in client-server applications to pass data between the clients and the servers.

Example Client-Server Communication Using a FIFO

- FIFO's can be used to send data between a client and a server. If we have a server that is contacted by numerous clients, each client can write its request to a well-known FIFO that the server creates. Since there are multiple writers for the FIFO, the requests sent by the clients to the server need to be less than PIPE_BUF bytes in size.
- This prevents any interleaving of the client writes. The problem in using FIFOs for this type of client server communication is how to send replies back from the server to each client.
- A single FIFO can't be used, as the clients would never know when to read their response versus responses for other clients. One solution is for each client to send its process ID with the request. The server then creates a unique FIFO for each client, using a pathname based on the client'sprocess ID.
- For example, the server can create a FIFO with the name /vtu/ ser.XXXXX, where XXXXX is replaced with the client's process ID. This arrangement works, although it is impossible for the server to tell whether a client crashes. This causes the client-specific FIFOs to be left in the file system.
- The server also must catch SIGPIPE, since it's possible for a client to send a request and terminate before reading the response, leaving the client-specific FIFO with one writer (the server) and no reader.

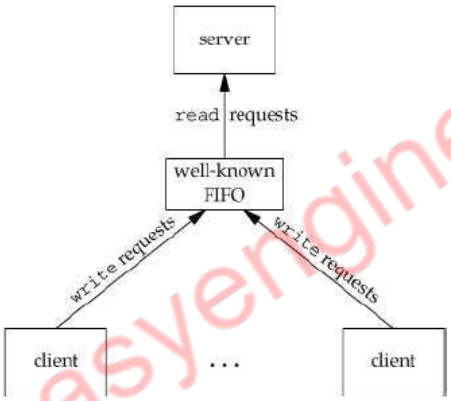


Figure 15.22. Clients sending requests to a server using a FIFO

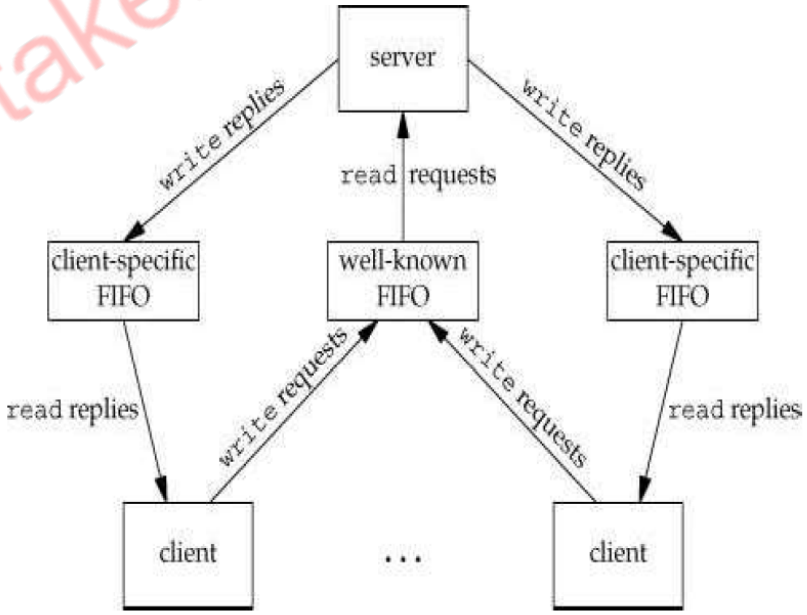


Figure 15.23. Client-server communication using FIFOs

8(A) - Explain briefly with example a) Message queue b) Semaphores.

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 (corresponding to the length), all of which are specified to msgsndwhen the message is added to a queue. Messages are fetched from a queue by msgrcv. We don't have to fetch the messages in a first-in, first-out order. Instead, we can fetch messages based on their type field.

Each queue has the following msqid_dsstructure associated with it:

```
struct msqid_ds
{
```

struct ipc_perm	msg_perm;	/* see Section 15.6.2 */
msgqnum_t	msg_qnum;	/* # of messages on queue */
msglen_t	msg_qbytes;	/* max # of bytes on queue */
pid_t	msg_lspid;	/* pid of last msgsnd() */
pid_t	msg_lrpid;	/* pid of last msgrcv() */

```
    time_t      msg_stime;  /* last-msgsnd()  time */ time_t
                msg_rtime;  /* last-msgrcv()  time */ time_t
                msg_ctime;  /* last-change time */
    .
    .
    .
};
```

This structure defines the current status of the queue.

The first function normally called is msggetto 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

When a new queue is created, the following members of the msqid_dsstructure are initialized.

- ✓ The ipc_perm structure is initialized. The mode member of this structure is set to the corresponding permission bits of flag.
- ✓ msg_qnum, msg_lspid, msg_lrpid, msg_stime, and msg_rtimeare all set to 0.
- ✓ msg_ctimeis set to the current time.
- ✓ msg_qbytesis set to the system limit.

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

The msgctlfunction 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.

Table 9.7.2 POSIX:XSI values for the cmd parameter of msgctl.

cmd	description
IPC_RMID	remove the message queue msqid and destroy the corresponding msqid_ds
IPC_SET	set members of the msqid_ds data structure from buf
IPC_STAT	copy members of the msqid_ds data structure into buf

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.

Each message is composed of a positive long integer type field, a non-negative length (nbytes), and the actual data bytes (corresponding to the length). Messages are always placed at the end of the queue.

The ptr argument points to a long integer that contains the positive integer message type, and it is immediately followed by the message data. (There is no message data if nbytes is 0.) If the largest message we send is 512 bytes, we can define the following structure:

```
struct mymesg
{
   long    mtype;       /* positive message type */
   char    mtext[512]; /* message data, of length nbytes */
};
```

The ptr argument is then a pointer to a mymesg structure. The message type can be used by the receiver to fetch messages in an order other than first in, first out.

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.

The type argument lets us specify which message we want.

type == 0	The first message on the queue is returned.
type > 0	The first message on the queue whose message type equals type is returned.
type < 0	The first message on the queue whose message type is the lowest value less than or equal to the absolute value of type is returned.

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.
2. 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. Otherwise, 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.

When a process is done with a shared resource that is controlled by a semaphore, the semaphore value is incremented by 1. If any other processes are asleep, waiting for the semaphore, they are awakened.

A common form of semaphore is called a ***binary semaphore***. It controls a single resource, and its value is initialized to 1. In general, however, a semaphore can be initialized to any positive value, with the value indicating how many units of the shared resource are available for sharing.

XSI semaphores are, unfortunately, more complicated than this. Three features contribute to this unnecessary complication.

1. A semaphore is not simply a single non-negative value. Instead, we have to define a semaphore as a set of one or more semaphore values. When we create a semaphore, we specify the number of values in the set.

2. The creation of a semaphore (semget) is independent of its initialization (semctl). This is a fatal flaw, since we cannot atomically create a new semaphore set and initialize all the values in the set.
3. Since all forms of XSI IPC remain in existence even when no process is using them, we have to worry about a program that terminates without releasing the semaphores it has been allocated. The undo feature that we describe later is supposed to handle this.

The kernel maintains a semid_dsstructure for each semaphore set:

```
struct semid_ds {
    struct ipc_perm   sem_perm; /* see Section 15.6.2 */
    unsigned short    sem_nsems; /* # of semaphores in set */
    time_t            sem_otime; /* last-semop() time */
    time_t            sem_ctime; /* last-change time */
    .
    .
    .
};
```

Each semaphore is represented by an anonymous structure containing at least the following members:

```
struct {
    unsigned short   semval;   /* semaphore value, always >= 0 */
    pid_t            sempid;   /* pid for last operation */
    unsigned short   semncnt;  /* # processes awaiting semval>curval */
    unsigned short   semzcnt;  /* # 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

When a new set is created, the following members of the semid_ds structure are initialized.

- The ipc_perm structure is initialized. The mode member of this structure is set to the corresponding permission bits of flag.
- sem_otime is set to 0.
- sem_ctime is set to the current time.
- sem_nsems is set to nsems.

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/sem.h>

int semctl(int semid, int semnum, int  cmd,... /* union semun arg */);
```

The fourth argument is optional, depending on the command requested, and if present, is of type semun, a union of
various command-specific arguments:

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


Table 9.8.1 POSIX:XSI values for the cmd parameter of semctl.

cmd	description
GETALL	return values of the semaphore set in arg.array
GETVAL	return value of a specific semaphore element
GETPID	return process ID of last process to manipulate element
GETNCNT	return number of processes waiting for element to increment
GETZCNT	return number of processes waiting for element to become 0
IPC_RMID	remove semaphore set identified by semid
IPC_SET	set permissions of the semaphore set from arg.buf
IPC_STAT	copy members of semid_ds of semaphore set semid into arg.buf
SETALL	set values of semaphore set from arg.array
SETVAL	set value of a specific semaphore element to arg.val

The *cmd* argument specifies one of the above ten commands to be performed on the set specified by semid. The function semopatomically performs an array of operations on a semaphore set.

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

```
int semop(int semid, struct sembuf semoparray[], size_t nops);
```

Returns: 0 if OK, 1 on error.

The semoparray argument is a pointer to an array of semaphore operations, represented by sembufstructures:

```
struct sembuf {
  unsigned short sem_num; /* member # in set (0, 1, ..., nsems-1) */ short
                  sem_op; /* operation (negative, 0, or positive) */ short
                  sem_flg; /* IPC_NOWAIT, SEM_UNDO */
};
```

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

The sem_op element operations are values specifying the amount by which the semaphore value is to be changed.

- If sem_op is an integer ***greater than zero***, semop adds the value to the corresponding semaphore element value and awakens all processes that are waiting for the element to increase.
- If sem_op is ***0*** and the semaphore element value is not 0, semop blocks the calling process (waiting for 0) and increments the count of processes waiting for a zero value of that element.

- If sem_op is a ***negative*** number, semop adds the sem_op value to the corresponding semaphore element value provided that the result would not be negative. If the operation would make the element value negative, semop blocks the process on the event that the semaphore element value increases. If the resulting value is 0, semop wakes the processes waiting for 0.

8(B) - Write a note on (i) Process accounting (ii) Process Times.

PROCESS ACCOUNTING

- Most UNIX systems provide an option to do process accounting. When enabled, the kernel writes an accounting record each time a process terminates.
- These accounting records are typically a small amount of binary data with the name of the command, the amount of CPU time used, the user ID and group ID, the starting time, and so on.
- A superuser executes accton with a pathname argument to enable accounting.
- The accounting records are written to the specified file, which is usually /var/account/acct. Accounting is turned off by executing accton without any arguments.
- The data required for the accounting record, such as CPU times and number of characters transferred, is kept by the kernel in the process table and initialized whenever a new process is created, as in the child after a fork.
- Each accounting record is written when the process terminates.
- This means that the order of the records in the accounting file corresponds to the termination order of the processes, not the order in which they were started.
- The accounting records correspond to processes, not programs.
- A new record is initialized by the kernel for the child after a fork, not when a new program is executed. The structure of the accounting records is defined in the header <sys/acct.h>and looks something like

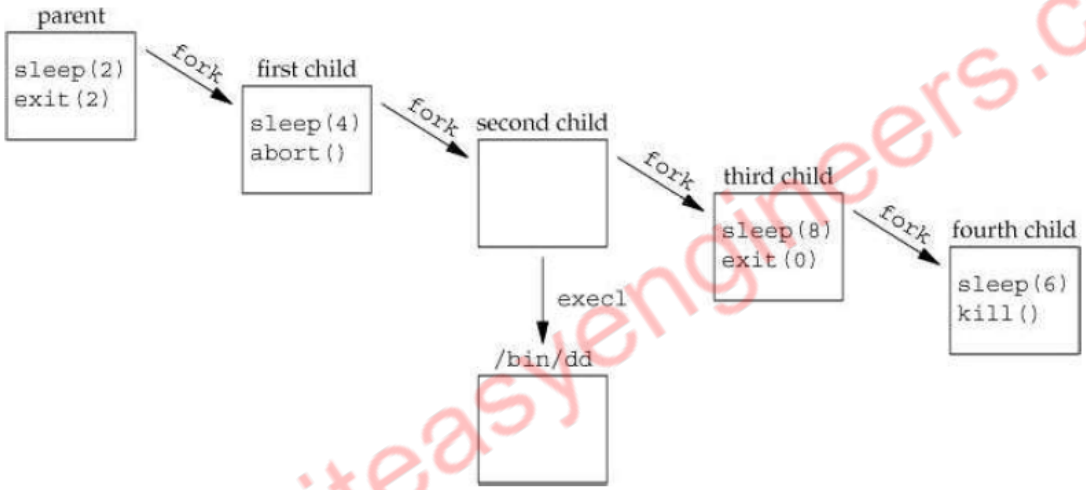
Values for ac_flagfrom accounting record

ac_flag	Description
AFORK	process is the result of fork, but never called exec
ASU	process used superuser privileges
ACOMPAT	process used compatibility mode
ACORE	process dumped core
AXSIG	process was killed by a signal
AEXPND	expanded accounting entry

Program to generate accounting data

```
#include "apue.h"

Int main(void)
{
    pid_t    pid;

    if ((pid = fork()) < 0)
        err_sys("fork error");
    else if (pid != 0) {              /* parent */
        sleep(2);
        exit(2);                      /* terminate with exit status 2 */
    }

                                      /* first child */
    if ((pid = fork()) < 0)
        err_sys("fork error");
    else if (pid != 0) {
        sleep(4);
        abort();                      /* terminate with core dump */
    }

                                      /* second child */
    if ((pid = fork()) < 0)
        err_sys("fork error");
    else if (pid != 0) {
        execl("/bin/dd", "dd", "if=/etc/termcap", "of=/dev/null", NULL);
        exit(7);                      /* shouldn't get here */
    }
```

```
                                /* third child */

    if ((pid = fork()) < 0)
        err_sys("fork error");
    else if (pid != 0) {
        sleep(8);
        exit(0);                /* normal exit */
    }

                                /* fourth child */
    sleep(6);
    kill(getpid(), SIGKILL);    /* terminate w/signal, no core dump */
    exit(6);                    /* shouldn't get here */
}
```



Process structure for accounting example

PROCESS TIMES

We describe three times that we can measure: wall clock time, user CPU time, and system CPU time. Any process can call the timesfunction to obtain these values for itself and any terminated children.

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

clock_t times(struct tms *buf);
```

Returns: elapsed wall clock time in clock ticks if OK, 1 on error

This function fills in the tmsstructure pointed to by buf:

```
struct tms {
  clock_t   tms_utime;  /* user CPU time */
  clock_t   tms_stime;  /* system CPU time */
  clock_t   tms_cutime; /* user CPU time, terminated children */
  clock_t   tms_cstime; /* system CPU time, terminated children */
};
```

Note that the structure does not contain any measurement for the wall clock time. Instead, the function returns the wall clock time as the value of the function, each time it's called. This value is measured from some arbitrary point in the past, so we can't use its absolute value; instead, we use its relative value.

9(A) - What are signals? Mention different source of signals? Write program to setup signal handlers for SIGINIT and SIGALRM.

THE UNIX KERNEL SUPPORT OF SIGNALS

- When a signal is generated for a process, the kernel will set the corresponding signal flag in the process table slot of the recipient process.
- If the recipient process is asleep, the kernel will awaken the process by scheduling it.
- When the recipient process runs, the kernel will check the process U-area that contains an array of signal handling specifications.
- If array entry contains a zero value, the process will accept the default action of the signal.
- If array entry contains a 1 value, the process will ignore the signal and kernel will discard it.
- If array entry contains any other value, it is used as the function pointer for a user-defined signal handler routine.

SIGNAL

The function prototype of the signal API is:

```
#include <signal.h>


void (*signal(int sig_no, void (*handler)(int)))(int);
```

The formal argument of the API are: sig_no is a signal identifier like SIGINT or SIGTERM. The handler argument is the function pointer of a user-defined signal handler function.

The following example attempts to catch the SIGTERM signal, ignores the SIGINT signal, and accepts the default action of the SIGSEGV signal. The pause API suspends the calling process until it is interrupted by a signal and the corresponding signal handler does a return:

```
#include<iostream.h>
#include<signal.h>
/*signal handler function*/ void
catch_sig(int sig_num)
{
        signal (sig_num,catch_sig);
        cout<<"catch_sig:"<<sig_num<<endl;
}


/*main function*/
int main()
{
        signal(SIGTERM,catch_sig);
        signal(SIGINT,SIG_IGN);
        signal(SIGSEGV,SIG_DFL);
        pause( );                       /*wait for a signal interruption*/
}
```

The following example checks whether the SIGINT signal is present in a process signal mask and adds it to the mask if it is not there.

```
#include<stdio.h>
#include<signal.h>
int main()
{
        sigset_t        sigmask;
        sigemptyset(&sigmask);              /*initialise set*/

        if(sigprocmask(0,0,&sigmask)==-1)       /*get current signal mask*/
        {
                perror("sigprocmask")
                ; exit(1);
        }
        else sigaddset(&sigmask,SIGINT); /*set SIGINT flag*/

        sigdelset(&sigmask, SIGSEGV);           /*clear SIGSEGV
        flag*/ if(sigprocmask(SIG_SETMASK,&sigmask,0)==-1)
                perror("sigprocmask");
}
```

A process can query which signals are pending for it via the sigpending API:

```
#include<signal.h>
int sigpending(sigset_t* sigmask);
```

Returns 0 if OK, -1 if fails.

The sigpending API can be useful to find out whether one or more signals are pending for a process and to set up special signal handling methods for these signals before the process calls the sigprocmask API to unblock them.

The following example reports to the console whether the SIGTERM signal is pending for the process:

```
#include<iostream.h>
#include<stdio.h>
#include<signal.h> int
main()
{
        sigset_t        sigmask;
        sigemptyset(&sigmask);
        if(sigpending(&sigmask)==-1)
                perror("sigpending");
        else cout << "SIGTERM signal is:"
                << (sigismember(&sigmask,SIGTERM) ? "Set" : "No Set")       << endl;
}
```

In addition to the above, UNIX also supports following APIs for signal mask manipulation:

```
#include<signal.h>

int sighold(int signal_num);
int sigrelse(int signal_num);
int sigignore(int signal_num);
int sigpause(int signal_num);
```

9(B) - Explain daemon characteristics and basic coding rules.

DAEMON CHARACTERISTICS

The characteristics of daemons are:

- Daemons run in background.
- Daemons have super-user privilege.
- Daemons don't have controlling terminal.
- Daemons are session and group leaders.

CODING RULES

- **Call umask to set the file mode creation mask to 0**. The file mode creation mask that's inherited could be set to deny certain permissions. If the daemon process is going to create files, it may want to set specific permissions.
- **Call fork and have the parent exit**. This does several things. First, if the daemon was started as a simple shell command, having the parent terminate makes the shell think that the command is done. Second, the child inherits the process group ID of the parent but gets a new process ID, so we're guaranteed that the child is not a process group leader.
- **Call setsid to create a new session**. The process (a) becomes a session leader of a new session, (b) becomes the process group leader of a new process group, and (c) has no controlling terminal.
- **Change the current working directory to the root directory**. The current working directory inherited from the parent could be on a mounted file system. Since daemons normally exist until the system is rebooted, if the daemon stays on a mounted file system, that file system cannot be unmounted.
- **Unneeded file descriptors should be closed.** This prevents the daemon from holding open any descriptors that it may have inherited from its parent.
- **Some daemons open file descriptors 0, 1, and 2 to /dev/null so that any library routines that try to read from standard input or write to standard output or standard error will have no effect**. Since the daemon is not associated with a terminal device, there is nowhere for output to be displayed; nor is there anywhere to receive input from an interactive user. Even if the daemon was started from an interactive session, the daemon runs in the background, and the login session can terminate without affecting the daemon. If other users log in on the same terminal device, we wouldn't want output from the daemon showing up on the terminal, and the users wouldn't expect their input to be read by the daemon.

10(A) - What is signal mask of a process? WAP to check whether the SIGINT signal present in signal mask.

SIGNAL MASK

A process initially inherits the parent's signal mask when it is created, but any pending signals for the parent process are not passed on. A process may query or set its signal mask via the sigprocmask API:

```
#include <signal.h>
int sigprocmask(int cmd, const sigset_t *new_mask, sigset_t *old_mask);
```

Returns: 0 if OK, 1 on error

The new_mask argument defines a set of signals to be set or reset in a calling process signal mask, and the cmd argument specifies how the new_mask value is to be used by the API. The possible values of cmd and the corresponding use of the new_mask value are:

Cmd value	Meaning
SIG_SETMASK	Overrides the calling process signal mask with the value specified in the new_mask argument.
SIG_BLOCK	Adds the signals specified in the new_mask argument to the calling process signal mask.
SIG_UNBLOCK	Removes the signals specified in the new_mask argument from the calling process signal mask.

- ✓ If the actual argument to new_mask argument is a NULL pointer, the cmd argument will be ignored, and the current process signal mask will not be altered.
- ✓ If the actual argument to old_mask is a NULL pointer, no previous signal mask will be returned.
- ✓ The sigset_t contains a collection of bit flags.

The BSD UNIX and POSIX.1 define a set of API known as sigsetops functions:

```
#include<signal.h>

int sigemptyset (sigset_t* sigmask);
int sigaddset (sigset_t* sigmask, const int sig_num);
int sigdelset (sigset_t* sigmask, const int sig_num);
int sigfillset (sigset_t* sigmask);
int sigismember (const sigset_t* sigmask, const int sig_num);
```

- The sigemptyset API clears all signal flags in the sigmask argument.
- The sigaddset API sets the flag corresponding to the signal_num signal in the sigmask
- argument. The sigdelset API clears the flag corresponding to the signal_num signal in the
- sigmask argument. The sigfillset API sets all the signal flags in the sigmask argument.
  [all the above functions return 0 if OK, -1 on error]
- The sigismember API returns 1 if flag is set, 0 if not set and -1 if the call fails.

The following example checks whether the SIGINT signal is present in a process signal mask and adds it to the mask if it is not there.

```
#include<stdio.h>
#include<signal.h>
int main()
{
```

```
sigset_t        sigmask;
sigemptyset(&sigmask);              /*initialise set*/


if(sigprocmask(0,0,&sigmask)==-1)       /*get current signal mask*/
{
        perror("sigprocmask")
        ; exit(1);
}
else sigaddset(&sigmask,SIGINT); /*set SIGINT flag*/

sigdelset(&sigmask, SIGSEGV);           /*clear SIGSEGV
flag*/ if(sigprocmask(SIG_SETMASK,&sigmask,0)==-1)
        perror("sigprocmask");
}
```

10(B) - Explain The sigsetjmp and siglongjmp Functions with examples.

THE sigsetjmp AND siglongjmp APIs

The function prototypes of the APIs are:

```
#include <setjmp.h>

int sigsetjmp(sigjmp_buf env, int savemask);
int siglongjmp(sigjmp_buf env, int val);
```

The sigsetjmp and siglongjmp are created to support signal mask processing. Specifically, it is implementation-dependent on whether a process signal mask is saved and restored when it invokes the setjmp and longjmp APIs respectively.

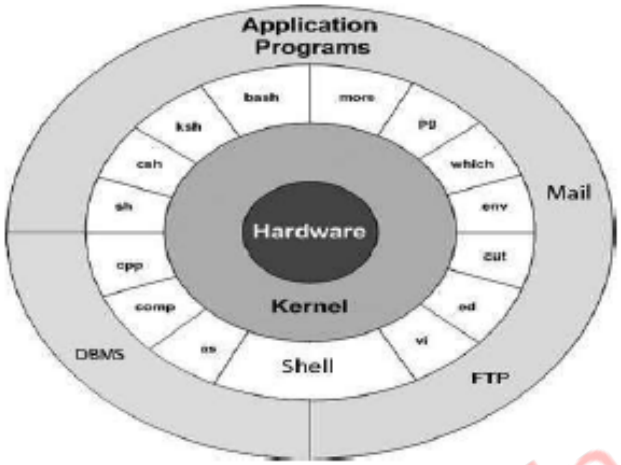
The only difference between these functions and the setjmp and longjmp functions is that sigsetjmp has an additional argument. If savemask is nonzero, then sigsetjmp also saves the current signal mask of the process in env. When siglongjmp is called, if the env argument was saved by a call to sigsetjmp with a nonzero savemask, then siglongjmp restores the saved signal mask. The siglongjmp API is usually called from user-defined signal handling functions. This is because a process signal mask is modified when a signal handler is called, and siglongjmp should be called to ensure the process signal mask is restored properly when "jumping out" from a signal handling function.

The following program illustrates the uses of sigsetjmp and siglongjmp APIs.

```
#include<iostream.h
> #include<stdio.h>
#include<unistd.h>
#include<signal.h>
#include<setjmp.h>

sigjmp_buf    env;

void callme(int sig_num)

{

        cout<< "catch signal:" <<sig_num
        <<endl; siglongjmp(env,2);
}

int main()

{

        sigset_t        sigmask;

        struct sigaction action,old_action;


        sigemptyset(&sigmask);


        if(sigaddset(&sigmask,SIGTERM)==-1) ||
                sigprocmask(SIG_SETMASK,&sigmask,0)==-1) perror("set signal mask");
        sigemptyset(&action.sa_mask);
        sigaddset(&action.sa_mask,SIGSEGV);
        action.sa_handler=(void(*)())callme;
        action.sa_flags=0;
        if(sigaction(SIGINT,&action,&old_action)==-
        1)
                perror("sigaction");
        if(sigsetjmp(env,1)!=0)
        {
                cerr<<"return from signal interruption";
                return 0;

        }
        else
                cerr<<"return from first time sigsetjmp is called";
```

1(A) - Illustrate unix architecture with neat diagram.

UNIX ARCHITECTURE



Kernel: is the core of operating system. A collection of routines mostly written in C.

It is loaded into memory when the system is booted and communicates directly with the hardware. The kernel manages system memory, processes, decides priorities.

Shell: interface between Kernel and User. It functions as command interpreter i,e it receives and interprets the command from user and interacts with the hardware. There is only one kernel running on the system, there could be several shells in action- one for each user who is logged in.

Files and Process: file is an array of bytes and it contain virtually anything. Unix considers even the directories and devices as members of file system. The dominant file type is text and behavior of system is mainly controlled by text files.

The second entity is the process, which is the name given to a file when it is executed as a program. Process is simply a time image of an executable file.

1.1 System Calls: Though there are thousands of commands in the unix system, they all use a handful of functions called system calls. User programs that need to access the hardware use the services of the kernel, which performs the job on users behalf. These programs access the kernel through a set of functions called system calls.

Ex: open()-- system call to access both file and device. Write()—system call to write a file.

1(C) - What are internal and external commands in UNIX? Explain them with suitable example.

MEANING OF INTERNAL AND EXTERNAL COMMANDS

UNIX commands are classified into two types

- Internal Commands - Ex: echo
- External Commands - Ex: ls, cat

Internal Command:

Internal commands are something which is built into the shell. For the shell built in commands, the execution speed is really high. It is because no process needs to be spawned for executing it.

- For example, when using the "cd" command, no process is created. The current directory sim- ply gets changed on executing it.

External Command:

External commands are not built into the shell. These are executable present in a separate file. When an external command has to be executed, a new process has to be spawned and the command gets exe- cuted.

- For example, when you execute the "cat" command, which usually is at /usr/bin, the executable /usr/bin/cat gets executed.

type command:

```
$ type cd
cd is a shell builtin
$ type cat
cat is /bin/cat
```

For the internal commands, the type command will clearly say its shell built-in, however for the external commands, it gives the path of the command from where it is executed.

THE TYPE COMMAND: knowing the type of a command and locating it.

type - Display information about command type.

The type command is a shell built-in that displays the kind of command the shell will execute, given a particular command name. It works like this: type command where "command" is the name of the command you want to examine. Here are some examples:

$type type

Output: type is a shell built-in

$type ls

Output: ls is aliased to 'ls –color=tty'

$type cp

Output: cp is /bin/cp

Here we see the results for three different commands. Notice that the one for ls (taken from a Fedora system) and how the ls command is actually an alias for the ls command with the "--color=tty" option added. Now we know why the output from ls is displayed in color!

2(A) - Illustrate command structure usage and behavior with respect to absolute and relative pathnames of following commands with suitable examples. i). mkdir ii). Rmdir.

mkdir: "making directory".

- **mkdir** is used to create directories on a file system.
- If the specified *DIRECTORY* does not already exist,**mkdir** creates it.
- More than one *DIRECTORY* may be specified when calling **mkdir**.

mkdir syntax
mkdir [*OPTION* ...] *DIRECTORY* ...

Ex 1: To create a directory named gmit, issue the following command.

$mkdir gmit

gmit directory is created under present working directory.

Assume that pwd is /home/kumar , then gmit directory is created under kumar directory.

Ex 2: To create three directories at a time, named patch, dbs, doc, pass directory names as argu- ments.

$mkdir patch dbs doc

Ex 3:To create a directory tree:

To create a directory named gmit and create two subdirectories named cse and ise under gmit, issue the command. gmit is a parent directory.

$mkdir parent directory sub-directories

$mkdir gmit gmit/cse gmit/ise

Ex 4: Error while creating a directory tree

$mkdir gmit/cse gmit/ise

mkdir: Failed to make a directory "gmit/cse"; no such file or directory

mkdir: Failed to make a directory "gmit/ise"; no such file or directory

Error is due to the fact that the parent directory named gmit is not created before creating sub directo- ries cse and ise.

Ex 5: **Smkdir test**

mkdir: Failed to make directory "test"; Permission denied.

This can happen due to:

a. The directory named test may already exist

b. There may be an ordinary file by the same name in the current directory.

c. The permissions set for the current directory do not permit the creation of files and directories by the user.

rmdir: REMOVING DIRECTORIES

The **rmdir** utility removes the directory entry specified by each directory argument, provided the directory is empty.

Ex 6.4.1: **Srmdir progs**

removes the directory named progs

Arguments are processed in the order given. To remove both a parent directory and a subdirectory of that parent, the subdirectory must be specified first, so the parent directory is empty when **rmdir** tries to remove it.

The reverse logic of mkdir is applied.

$rmdir subdirectories parent directory

$rmdir gmit/cse gmit/ise gmit

- You cant delete a directory with rmdir unless it is empty.In this example gmit directory cannot be removed until the sub directories cse and ise are removed.
- You cant remove a sub directory unless you are place in a directory which is hierarchically above the one you have chosen to remove.

ls : listing directory contents:

To obtain a list of all filenames in the current directory.

Numerals first

Uppercase next

Lowercase Then

$ls

Output:08_packets.html

calendar

dept.lst

emp.lst

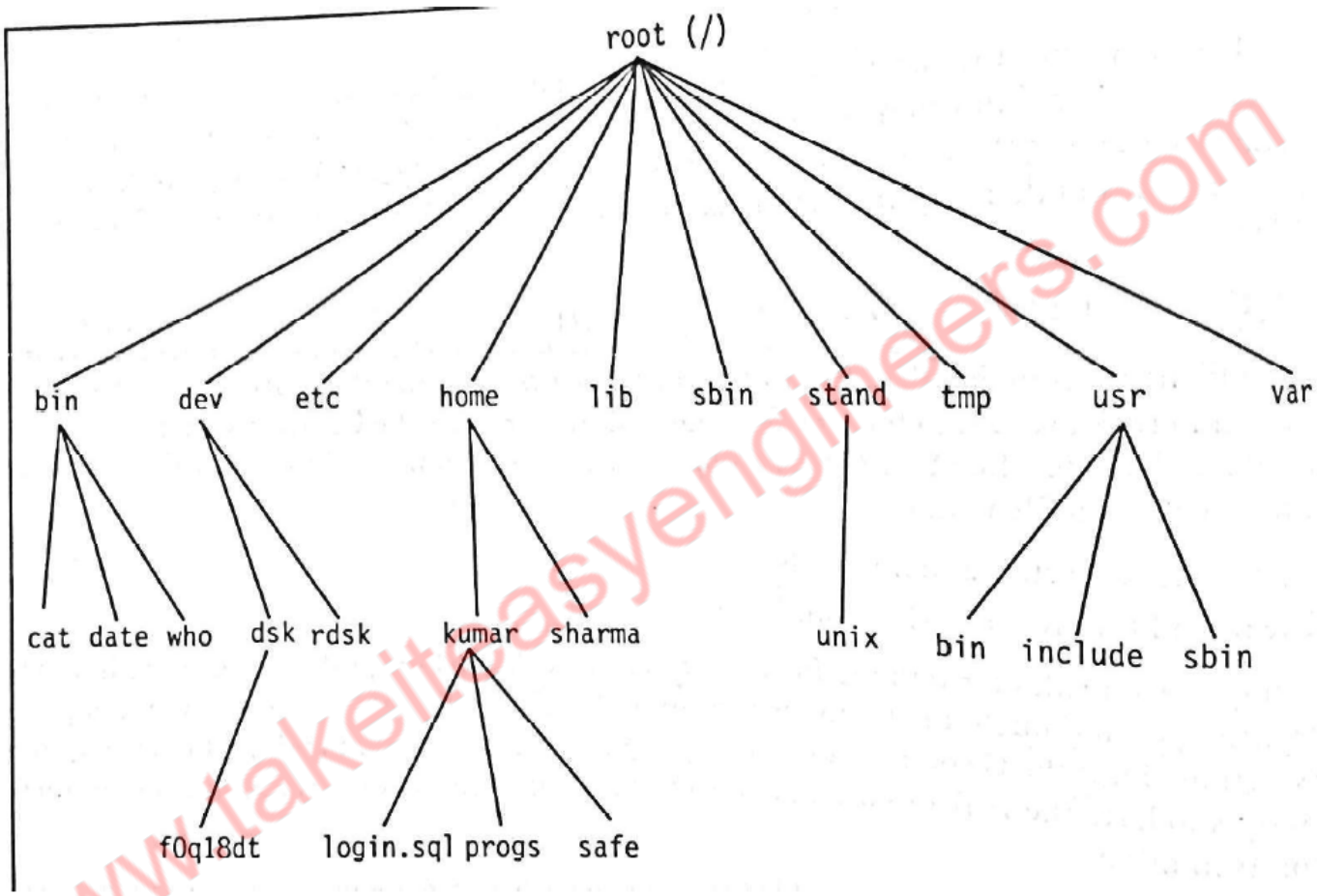
helpdir

uskdsk06

2(C) - Explain parent-child relationship in UNIX file system.

PARENT CHILD RELATIONSHIP/ ORGANIZATION OF FILES

All files in UNIX are related to one another. The file system in unix is a collection of all ordinary, directory and device files and organized in a hierarchical structure as shown in below fig.



The implicit feature of every UNIX file system is that there is a top which serves as reference point for all files.This top is called **root** & is represented by a /(front slash). Root is actually a directory. The root directory has a number of sub directories under it. These sub directories in turn have more sub directories and others files under them.

For instance bin and usr are two directories directly under root, while a second bin and kumar are sub directories under usr.

Every file apart from root must have a parent. Thus the home directory is the parent of kumar , while / is the parent of home and grandparent of kumar. If you create a file login.sql under the kumar directory ,then kumar will be the parent of this file.

The first group contains the files that are made available during system installation

- **/bin and /usr/bin:** these are the directories where all the commonly used UNIX commands are found.
- **/sbin and /usr/sbin:** If there's a command that you can't execute but the system administrator can execute, it would be probably in one of these directories.
- **/etc:** this directory contains the configuration files of the system. You can change a very important aspect of system functioning by editing a text file in this directory. Your login name and password are stored in files /etc/passwd and etc/shadow
- **/dev:** This directory contains all device files. These files don't occupy space on disk.there could be more sub directories like pts, dsk and rdsk in this directory
- **/lib and /usr/lib:** Contains shared library files and sometimes other kernel-related files.
- **/usr and /include:** contains the standard header files used by C programs. The statement #include<stdio.h> used in most C programs referes to the file stdio.h in this directory.
- **/usr/share/man:** this is where the man pages are stored. There are separate subdirectories here(like man1,man2 etc) that contains the pages for each section. For instance, the man page of **ls** can be found in /usr/share/man/man1

User also work with their own files, they write programs, send and receive mail and also create temporary files. These files are available in the second group shown below

- **/tmp:** the directory where users are allowed to create temporary files. These files are wiped away regularly by the system
- **/var:** The variable part of the file system. Contains all your print jobs and your outgoing and incoming mail.
- **/home:**On many systems users are housed here.Kumar would have his home directory in /home/kumar

3(A) - Which command is used for listing file attributes? Briefly describe the significance of each field of the output.

The ls command with options

ls –l: LISTING FILE ATTRIBUTES

ls command is used to obtain a list of all filenames in the current directory. The output in UNIX lingo is often referred to as the listing. Sometimes we combine this option with other options for displaying other attributes, or ordering the list in a different sequence. ls look up the file's inode to fetch its attributes. It lists seven attributes of all files in the current directory and they are:

File type and Permissions

The file type and its permissions: The first column shows the type and permissions associated with each file.The first character in this column is mostly a – which indicates that the file is an ordinary one. In unix, file system has three types of permissions- read, write and execute.

Links: The second column indicates the number of links associated with the file. This is actually the number of filenames maintained by the system of that file.

Ownership: The third column shows the owner of files. The owner has full authority to tamper with files content and permissions. Similarly, you can create, modify or remove files in a directory if you are the owner of the directory.

Group ownership: The fourth column represents the group owner of the file. When opening a user account, the system admin also assigns the user to some group. The concept of a group of users also owning a file has acquired importance today as group members often need to work on the same file.

File size: The fifth column shows the size of the file in bytes. The important thing to remember here is that it only a character count of the file and not a measure of the disk space that it occupies.

Last modification time: The sixth, seventh and eighth columns indicate the last modification time of the file, which is stored to the nearest second. A file is said to be modified only if its content have changed in any way.If the file is less than a year old since its last modification time, the year won't be displayed.

Filename: The last column displays the filename arranged in ASCII collating sequence.

3(C) - Explain wild cards with examples and its various types.

WILD CARDS

The metacharacters taht are used to construct the generalized pattern for matching filenames belong to the category called **wild cards.**

The * and ?

- **The metacharacter** * is one of the characters of the shell wild card set. It matches any number of characters including none.
- For example : to match filenames chap chap01 chap02 chap03 chap04

  $ ls chap*

  Output : // the * matches all strings along with none

  chap
  chap01
  chap02
  chap03
  chap04

- **The metacharacter ?** matches a single character
  For example: to match filenames chapx chapy chapz

  $ ls chap?
  Output: //the ? replaces single character i,e ? is replaced by x,y,z
  chapx
  chapy
  chapz

Matching the dot(.)

- The **.** dot metachacter can be used to match all the hidden files in your directory.
- Example: To list all hidden files in your directory having atleast three characters after the dot.

  $ ls .???*
  Output:
  .bash_profile
  .exrc
  .netscape
  .profile

The character class [ ]

- This class comprises a set of charcters enclosed by the rectangular brackets [and],but it matches only a single character in the class.
- The **pattern [abcd]** is a character class and it matches a single character a, b, c or d
- For example to match chap01 chap02 chap03 chap04
  $ ls chap0[1234]
  output:
  chap01

chap02
chap03
chap04

Negating the character class(!)

- It is used to reverse the matching criteria.
- For example:
  To match all the filenames with single character extensions but not .c or .o files
  $ ls *.[!co]
  *** to match any filename**
  . extension
  ! except
  [!co]---> .c extension or .o extension
- To match filenames that does not begin with a digit
  $ ls [!0-9]*
- To match filename with 3 character that does not begin with an Upper Case letter
  $ ls [!A-Z]??

4(B) - Explain grep command with all options.

grep(globally search regular expression and print)

Unix has a special family of commands for handling search requirements and the principal member of the family is the **grep command**.

grep scans its input for a pattern and displays lines containing the pattern, the line numbers or filename where the pattern occurs.

Syntax:

grep options pattern filename(s)

grep searches for pattern in one or more filename or the standard input if no filename is specified.

The first argument(barring the options) is the pattern and the remaining arguments are filenames.

Consider three files emp.lst, emp1.lst, emp2.lst

Scat emp.lst

```
101|sharma|general manager|sales|10/09/61|6700
102|kumar|director|Sales|09/09/63|7700
103|aggarwal|manager|sales|03/05/70|5000
104|rajesh|manager|marketing|12/04/72|5800
105|adarsh|executive|sales|07/09/57|5300
```

Scat emp1.lst

```
201|anil|director|sales|05/01/59|5000
202|sunil|director|marketing|12/06/51|6000
203|gupta|director|production|09/08/55|7000
```

grep along with options

Table 13.1 Options Used by **grep**

Option	*Significance*
-i	Ignores case for matching
-v	Doesn't display lines matching expression
-n	Displays line numbers along with lines
-c	Displays count of number of occurrences
-l	Displays list of filenames only
-e *exp*	Specifies expression with this option. Can use multiple times. Also used for matching expression beginning with a hyphen.
-x	Matches pattern with entire line (doesn't match embedded patterns)
-f *file*	Takes patterns from *file*, one per line
-E	Treats pattern as an extended regular expression (ERE)
-F	Matches multiple fixed strings (in **fgrep**-style)

Ignoring case: when you look for a name but are not sure of the case, use the –i option to ignore case for pattern matching.

```
$ grep -i "sales" emp.lst
101|sharma|general manager|sales|10/09/61|6700
102|kumar|director|Sales|09/09/63|7700
103|aggarwal|manager|sales|03/05/70|5000
105|adarsh|executive|sales|07/09/57|5300
```

Deleting lines(-v): The –v option selects all lines except those containing the pattern

```
$ grep -v "director" emp2.lst
301|anil Agarwal|manager|sales
303|rajath Agarwal|manager|sales
305|v.s.agrawal|deputy manager|sales
306|agrawal|executive|sales
401|sumith|general manager|marketing
```

The lines containing the pattern **director** are deleted in the output.

Dislaying line numbers(-n).

The –n option displays the line numbers containing the pattern along with the line

```
$ grep -n "sales" emp.lst
1:101|sharma|general manager|sales|10/09/61|6700
3:103|aggarwal|manager|sales|03/05/70|5000
5:105|adarsh|executive|sales|07/09/57|5300
```

Dislaying filenames(-l)

The –l option displays only the names of the files containing the pattern.

Here the pattern **manager** is searched in all files ending with .lst (*.lst)

```
$ grep -l "manager" *.lst
emp2.lst
emp.lst
```

Matching multiple patterns(-e)

By using –e option, you can match multiple patterns

```
$ grep -e "agarwal" -e "Agarwal" -e "agrawal" emp2.lst
301|anil Agarwal|manager|sales
302|sudhir agarwal|director|production
303|rajath Agarwal|manager|sales
304|v.k.agrawal|director|marketing
305|v.s.agrawal|deputy manager|sales
306|agrawal|executive|sales
402|sudhir agarwal|director|production
```

5(A) - Describe general unix file API's with syntax and explain the each field in detail

General file API's

Files in a UNIX and POSIX system may be any one of the following types:

- Regular file
- Directory File
- FIFO file
- Block device file
- character device file
- Symbolic link file.

There are special API's to create these types of files. There is a set of Generic API's that can be used to manipulate and create more than one type of files. These API's are:

❖ **open**

✓ This is used to establish a connection between a process and a file i.e. it is used to open an existing file for data transfer function or else it may be also be used to create a new file.

✓ The returned value of the open system call is the file descriptor (row number of the file table), which contains the inode information.

✓ The prototype of open function is

```
#include<sys/types.h>
#include<sys/fcntl.h>
int open(const char *pathname, int accessmode, mode_t permission);
```

- ✓ If successful, open returns a nonnegative integer representing the open file descriptor.
- ✓ If unsuccessful, open returns –1.
- ✓ The first argument is the name of the file to be created or opened. This may be an absolute pathname or relative pathname.
- ✓ If the given pathname is symbolic link, the open function will resolve the symbolic link reference to a non-symbolic link file to which it refers.
- ✓ The second argument is access modes, which is an integer value that specifies how actually the file should be accessed by the calling process.
- ✓ Generally, the access modes are specified in <fcntl.h>. Various access modes are:

O_RDONLY	- open for reading file only
O_WRONLY	- open for writing file only
O_RDWR	- opens for reading and writing file.

There are other access modes, which are termed as access modifier flags, and one or more of the following can be specified by bitwise-ORing them with one of the above access mode flags to alter the access mechanism of the file.

- ✓ To illustrate the use of the above flags, the following example statement opens a file called /usr/divya/usp for read and write in append mode:
  int fd=open("/usr/divya/usp",O_RDWR | O_APPEND,0);
- ✓ If the file is opened in read only, then no other modifier flags can be used.
- ✓ If a file is opened in write only or read write, then we are allowed to use any modifier flags along with them.

- ✓ The third argument is used only when a new file is being created. The symbolic names for file permission are given in the table in the previous page.

❖ **creat**

- This system call is used to create new regular files.
- The prototype of creat is

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

- Returns: file descriptor opened for write-only if OK, -1 on error.
- The first argument pathname specifies name of the file to be created.
- The second argument mode_t, specifies permission of a file to be accessed by owner group and others.
- The creat function can be implemented using open function as:
  - **#define creat(path_name, mode)**
  - **open (pathname, O_WRONLY | O_CREAT | O_TRUNC, mode);**

❖ **read**

- The read function fetches a fixed size of block of data from a file referenced by a given file descriptor.
- The prototype of read function is:

```
#include<sys/types.h>
#include<unistd.h>
size_t read(int fdesc, void *buf, size_t nbyte);
```

- If successful, read returns the number of bytes actually read.
- If unsuccessful, read returns –1.
- The first argument is an integer, fdesc that refers to an opened file.
- The second argument, buf is the address of a buffer holding any data read.
- The third argument specifies how many bytes of data are to be read from the file.
- The size_t data type is defined in the <sys/types.h> header and should be the same as unsigned int.
- There are several cases in which the number of bytes actually read is less than the amount requested:
  - When reading from a regular file, if the end of file is reached before the requested number of bytes has been read. For example, if 30 bytes remain until the end of file and we try to read 100 bytes, read returns 30. The next time we call read, it will return 0 (end of file).
  - When reading from a terminal device. Normally, up to one line is read at a time.
  - When reading from a network. Buffering within the network may cause less than the requested amount to be returned.
  - When reading from a pipe or FIFO. If the pipe contains fewer bytes than requested, read will return only what is available.

❖ **write**

- The write system call is used to write data into a file.
- The write function puts data to a file in the form of fixed block size referred by a given file descriptor.
- The prototype of write is

```
#include<sys/types.h>
#include<unistd.h>
ssize_t write(int fdesc, const void *buf, size_t size);
```

- If successful, write returns the number of bytes actually written.
- If unsuccessful, write returns –1.
- The first argument, fdesc is an integer that refers to an opened file.
- The second argument, buf is the address of a buffer that contains data to be written.
- The third argument, size specifies how many bytes of data are in the buf argument.
- The return value is usually equal to the number of bytes of data successfully written to a file. (*size* value)

❖ **close**

- The close system call is used to terminate the connection to a file from a process.
- The prototype of the close is

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

- If successful, close returns 0.
- If unsuccessful, close returns –1.
- The argument fdesc refers to an opened file.
- Close function frees the unused file descriptors so that they can be reused to reference other files. This is important because a process may open up to OPEN_MAX files at any time and the close function allows a process to reuse file descriptors to access more than OPEN_MAX files in the course of its execution.
- The close function de-allocates system resources like file table entry and memory buffer allocated to hold the read/write.

❖ **fcntl**

- The fcntl function helps a user to query or set flags and the close-on-exec flag of any file descriptor.
- The prototype of fcntl is

```
#include<fcntl.h>
int fcntl(int fdesc, int cmd, …);
```

- The first argument is the file descriptor.
- The second argument cmd specifies what operation has to be performed.
- The third argument is dependent on the actual cmd value.
- The possible cmd values are defined in <fcntl.h> header.

5(B) - Explain file and record locking in detail.

File and Record Locking

- Multiple processes performs read and write operation on the same file concurrently.
- This provides a means for data sharing among processes, but it also renders difficulty for any process in determining when the other process can override data in a file.

- So, in order to overcome this drawback UNIX and POSIX standard support file locking mechanism.
- File locking is applicable for regular files.
- Only a process can impose a write lock or read lock on either a portion of a file or on the entire file.
- The differences between the read lock and the write lock is that when write lock is set, it prevents the other process from setting any over-lapping read or write lock on the locked file.
- Similarly when a read lock is set, it prevents other processes from setting any overlapping write locks on the locked region.
- The intension of the write lock is to prevent other processes from both reading and writing the locked region while the process that sets the lock is modifying the region, so write lock is termed as "**Exclusive lock**".
- The use of read lock is to prevent other processes from writing to the locked region while the process that sets the lock is reading data from the region.
- Other processes are allowed to lock and read data from the locked regions. Hence a read lock is also called as "**shared lock** ".

- File lock may be **mandatory** if they are enforced by an operating system kernel.
- If a mandatory exclusive lock is set on a file, no process can use the read or write system calls to access the data on the locked region.
- These mechanisms can be used to synchronize reading and writing of shared files by multiple processes.
- If a process locks up a file, other processes that attempt to write to the locked regions are blocked until the former process releases its lock.
- Problem with mandatory lock is – if a runaway process sets a mandatory exclusive lock on a file and never unlocks it, then, no other process can access the locked region of the file until the runway process is killed or the system has to be rebooted.
- If locks are not mandatory, then it has to be **advisory** lock.
- A kernel at the system call level does not enforce advisory locks.
- This means that even though a lock may be set on a file, no other processes can still use the read and write functions to access the file.
- To make use of advisory locks, process that manipulate the same file must co-operate such that they follow the given below procedure for every read or write operation to the file.
    1. Try to set a lock at the region to be accesses. If this fails, a process can either wait for the lock request to become successful.
    2. After a lock is acquired successfully, read or write the locked region.
    3. Release the lock.
- If a process sets a read lock on a file, for example from address 0 to 256, then sets a write lock on the file from address 0 to 512, the process will own only one write lock on the file from 0 to 512, the previous read lock from 0 to 256 is now covered by the write lock and the process does not own two locks on the region from 0 to 256. This process is called "**Lock Promotion**".
- Furthermore, if a process now unblocks the file from 128 to 480, it will own two write locks on the file: one from 0 to 127 and the other from 481 to 512. This process is called "**Lock Splitting**".
- UNIX systems provide fcntl function to support file locking. By using fcntl it is possible to impose read or write locks on either a region or an entire file.
- The prototype of fcntl is

```
#include<fcntl.h>
int fcntl(int fdesc, int cmd_flag,.....);
```

5(C) - List the number of ways a process can terminate?

PROCESS TERMINATION

There are eight ways for a process to terminate. Normal termination occurs in five ways:

- Return from main
- Calling exit
- Calling _exitor _Exit
- Return of the last thread from its start routine
- Calling pthread_exit from the last thread

Abnormal termination occurs in three ways:

- Calling abort
- Receipt of a signal
- Response of the last thread to a cancellation request

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);
```

All three exit functions expect a single integer argument, called the exit status. Returning an integer value from the
main function is equivalent to calling exit with the same value. Thus
exit(0); is the same as
return(0);
from the main function.

In the following situations the exit status of the process is undefined.

- any of these functions is called without an exit status.
- main does a return without a return value
- main "falls off the end", i.e if the exit status of the process is

undefined. Example:

atexitFunction

With ISO C, a process can register up to 32 functions that are automatically called by exit. These are called exit handlers and are registered by calling the atexitfunction.

```
#include <stdlib.h>
int atexit(void (*func)(void));
```

Returns: 0 if OK, nonzero on error

Returns: 0 if OK, nonzero on error

This declaration says that we pass the address of a function as the argument to atexit. When this function is called, it is not passed any arguments and is not expected to return a value. The exit function calls these functions in reverse order of their registration. Each function is called as many times as it was registered.

Example of exit handlers

```
#include "apue.h"

static void my_exit1(void);
static void my_exit2(void);

int main(void)
{
        if (atexit(my_exit2) != 0) err_sys("can't
              register my_exit2");

        if (atexit(my_exit1) != 0) err_sys("can't
              register my_exit1");

        if (atexit(my_exit1) != 0) err_sys("can't
              register my_exit1");

        printf("main is done\n");
        return(0);
}

static void
my_exit1(void)
{
    printf("first exit handler\n");
}

static void
my_exit2(void)

{
    printf("second exit handler\n");
}
```

Output:

```
$ ./a.out
      main is done
      first exit handler first
      exit handler second exit
      handler
```

The below figure summarizes how a C program is started and the various ways it can terminate.

6(B) - Explain the following commands i)fork ii)vfork iii)exit

fork FUNCTION

An existing process can create a new one by calling the forkfunction.

```
#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 forkis 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.
- The reason the child's process ID is returned to the parent is that a process can have more than one child, and there is no function that allows a process to obtain the process IDs of its children.
- The reason fork returns 0 to the child is that a process can have only a single parent, and the child can always call getppid to obtain the process ID of its parent. (Process ID 0 is reserved for use by the kernel, so it's not possible for 0 to be the process ID of a child.)

vfork FUNCTION

- ✓ The function vforkhas the same calling sequence and same return values as fork.
- ✓ The vfork function is intended to create a new process when the purpose of the new process is to exec a new program.
- ✓ 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; the child simply calls exec (or exit) right after the vfork.
- ✓ Instead, while the child is running and until it calls either exec or exit, the child runs in the address space of the parent. This optimization provides an efficiency gain on some paged virtual-memory implementations of the UNIX System.
- ✓ Another difference between the two functions is that vfork guarantees that the child runs first, until the child calls execor exit. When the child calls either of these functions, the parent resumes.

exit FUNCTIONS

A process can terminate normally in five ways:

- Executing a return from the main function.
- Calling the exit function.
- Calling the _exit or _Exit function.

  In most UNIX system implementations, exit(3) is a function in the standard C library, whereas _exit(2) is a system call.
- Executing a return from the start routine of the last thread in the process. When the last thread returns from its start routine, the process exits with a termination status of 0.
- Calling the pthread_exit function from the last thread in the

process. The three forms of abnormal termination are as follows:

- Calling abort. This is a special case of the next item, as it generates the SIGABRT signal.
- When the process receives certain signals. Examples of signals generated by the kernel include the process referencing a memory location not within its address space or trying to divide by 0.
- The last thread responds to a cancellation request. By default, cancellation occurs in a deferred manner: one thread requests that another be canceled, and sometime later, the target thread terminates.

6(C) - Define race condition and polling? How to overcome these conditions.

RACE CONDITIONS

A race condition occurs when multiple processes are trying to do something with shared data and the final outcome depends on the order in which the processes run.

program modification to avoid race condition

```
#include "apue.h"
    static void charatatime(char *);
    int main(void)
    {
        pid_t    pid;

+       TELL_WAIT();
+
        if ((pid = fork()) < 0) {
            err_sys("fork error");
        } else if (pid == 0) {
+           WAIT_PARENT();  /* parent goes first */
            charatatime("output from child\n");
        } else {
            charatatime("output from parent\n");
+           TELL_CHILD(pid);
        }
        exit(0);
    }
```

```
static void
charatatime(char *str)
{
        char    *ptr;
        int     c;

        setbuf(stdout, NULL);                   /* set unbuffered */
        for (ptr = str; (c = *ptr++) != 0; )
                putc(c, stdout);
}
```

8(B) - What are Interpreter Files? Give the difference between Interpreter Files and Interpreter.

INTERPRETER FILES

These files are text files that begin with a line of the form

#! pathname [optional-argument]

The space between the exclamation point and the pathname is optional. The most common of these interpreter files begin with the line

#!/bin/sh

The pathname is normally an absolute pathname, since no special operations are performed on it (i.e., PATH is not used). The recognition of these files is done within the kernel as part of processing the exec system call. The actual file that gets executed by the kernel is not the interpreter file, but the file specified by the pathname on the first line of the interpreter file. Be sure to differentiate between the interpreter filea text file that begins with #!and the interpreter, which is specified by the pathname on the first line of the interpreter file.
Be aware that systems place a size limit on the first line of an interpreter file. This limit includes the #!, the pathname, the optional argument, the terminating newline, and any spaces.

A program that execs an interpreter file

```
#include "apue.h"
#include <sys/wait.h>

Int main(void)
{
    pid_t    pid;
    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) {                  /* child */
        if (execl("/home/sar/bin/testinterp",
                  "testinterp", "myarg1", "MY ARG2", (char *)0) <
            0) err_sys("execl error");
```

```
        if (waitpid(pid, NULL, 0) < 0) /* parent */
            err_sys("waitpid error");
        exit(0);
}
```

Output:

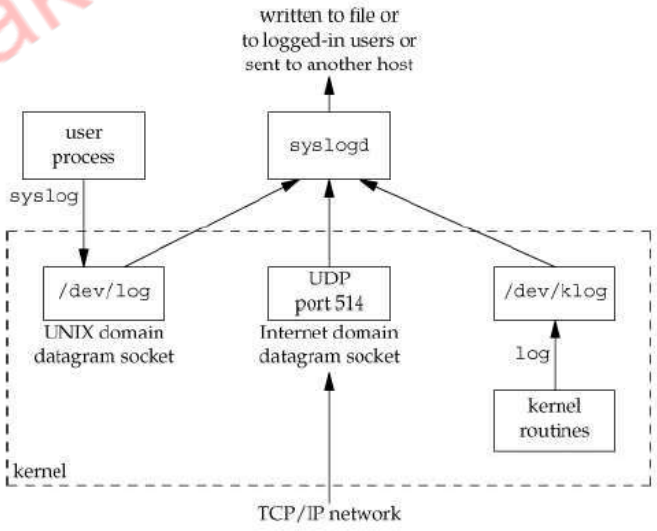
```
$ cat /home/sar/bin/testinterp
#!/home/sar/bin/echoarg foo
$ ./a.out
argv[0]: /home/sar/bin/echoarg
argv[1]: foo
argv[2]: /home/sar/bin/testinterp
argv[3]: myarg1
argv[4]: MY ARG2
```

10(A) - What is error logging? With a neat block schematic discuss the error login facility in BSD.

ERROR LOGGING

One problem a daemon has is how to handle error messages. It can't simply write to standard error, since it shouldn't have a controlling terminal. We don't want all the daemons writing to the console device, since on many workstations, the console device runs a windowing system. A central daemon error-logging facility is required.

Figure 13.2. The BSD `syslog` facility



There are three ways to generate log messages:

- Kernel routines can call the logfunction. These messages can be read by any user process that opens and
  reads the /dev/klogdevice.
- Most user processes (daemons) call the syslog(3) function to generate log messages. This causes the

message to be sent to the UNIX domain datagram socket /dev/log.

- A user process on this host, or on some other host that is connected to this host by a TCP/IP network, can send log messages to UDP port 514. Note that the syslog function never generates these UDP datagrams: they require explicit network programming by the process generating the log message.

Normally, the syslogd daemon reads all three forms of log messages. On start-up, this daemon reads a configuration file, usually /etc/syslog.conf, which determines where different classes of messages are to be sent. For example, urgent messages can be sent to the system administrator (if logged in) and printed on the console, whereas warnings may be logged to a file. Our interface to this facility is through the syslog function.

```
#include <syslog.h>
void openlog(const char *ident, int option, int facility);
```

```
void syslog(int priority, const char *format, ...);
void closelog(void);
int setlogmask(int maskpri);
```