Module – 1

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

(b) List and explain the salient features of Unix operating system

Features Of UNIX

UNIX is an operating system, so it has all the features an operating system is expected to have:

- A Multiuser System
- A Multitasking System
- The building-block approach
- The UNIX toolkit
- Pattern Matching
- Programming Facility
- Documentation
- Portable
- Networking
- Organization of File System
- Device Independence
- Utilities
- Services

1. **A Multiuser System:**
   - UNIX is a multiprogramming system; it permits multiple programs to run and compete for the attention of the CPU.
   - This can happen in two ways:
     ✦ Multiple users can run separate jobs ✦ A single user can also run multiple jobs

2. **A Multitasking System:**
   - A single user can also run multiple tasks concurrently.
   - UNIX is a multitasking system.
   - It is usual for a user to edit a file, print another one on the printer, send email to a friend and browse www - all without leaving any of applications.
   - The kernel is designed to handle a user's multiple needs.
   - In a multitasking environment, a user sees one job running in the foreground; the rest run in the background.
   - User can switch jobs between background and foreground, suspend, or even terminate them.

1. **The Building-block Approach:**
   - The designer never attempted to pack too many features into a few tools.
   - Instead, they felt “small is beautiful”, and developed a few hundred commands each of which performed one simple job.
   - UNIX offers the | (filters) to combine various simple tools to carry out complex jobs.
   - **Example:**

     $ cat note #cat displays the file contents
     WELCOME TO HIT
     $ cat note | wc #wc counts number of lines, words & characters in the file **1 3 15**

3. **The UNIX Toolkit:**
   - Kernel itself doesn't do much useful task for users.
   - UNIX offers facility to add and remove many applications as and when required.
   - Tools include general purpose tools, text manipulation tools, compilers, interpreters, networked applications and system administration tools.

4. **Pattern Matching:**
   - UNIX features very sophisticated pattern matching features.
   - Example: The * (zero or more occurrences of characters) is a special character used by system to indicate that it can match a number of filenames.
   - **Ex: ls chap* , ls chap+ , ls chap?**

5. **Programming Facility:**
   - The UNIX shell is also a programming language it was designed for programmer, not for end user.
   - It has all the necessary ingredients, like control structures, loops and variables, that establish powerful programming language.
   - These features are used to design shell scripts – programs that can also invoke UNIX commands.
   - Many of the system's functions can be controlled and automated by using these shell scripts.

6. **Documentation:**
   - The principal on-line help facility available is the man command, which remains the most important references for commands and their configuration files.
   - Apart from the man documentation, there's a vast ocean of UNIX resources available on the Internet.
7. **Portable:** The programs that are written in C are easily moved from one hardware to other hardware. It needs only standard compile.
8. **Networking:** Networking allows the users at one location to log into systems at other sites. Once access is gained to a remote system user operate just as though they were on their own system. TCP/IP protocol is used for communication

9. **Organized File System:** UNIX has a very organized file & directory system that allows users to organize and maintain files.
10. **Device Independence**: UNIX treats input / output devices like ordinary files.
11. **Utilities:** UNIX provides rich library packages. A user can develop programs / products in an efficient manner.
12. **Services:** It also includes the support utilities for system administration & control System Administrator is responsible in maintaining system resources like Disk, security access.

2.(a) Explain the following commands with syntax ,option and example Echo ,ls,who,passwd,date 7

echo: Displaying the Message:

- echo command is used in shell scripts to display a message on the terminal, or to issue a prompt for taking user input.
- **Example:**

```
$ echo "Enter your name:\c"
Enter your name:$_
$echo $SHELL
/usr/bin/bash

bash-4.1$ echo "ENTER
YOUR NAME:"  ENTER
YOUR NAME:
bash-4.1$

bash-4.1$ echo -e "ENTER YOUR NAME:\c"
ENTER YOUR NAME:bash-4.1$
bash-4.1$ echo -e "ENTER YOUR NAME:\t"
ENTER YOUR NAME:
bash-4.1$     bash-4.1$ echo -e
"ENTER YOUR NAME:\n"
ENTER YOUR NAME: bash-4.1$
```

- Echo can be used with different escape sequences.

- who: WHO ARE THE USERS?:

```
bash-4.1$ who
student  tty1        2020-09-07 09:52 (:0)
student  pts/0       2020-09-07 09:53 (:0.0)
bash-4.1$ who -H (H- Header)
NAME      LINE          TIME             IDLE          PID
COMMENT    student  tty1        2020-09-07 09:52  old
2133 (:0)     student   pts/0        2020-09-07 09:53    .
2584 (:0.0)
bash-4.1$
```

bash-4.1**$ who am i student pts/0**
2020-09-07 09:53 (:0.0) bash-4.1**$**

bash-4.1**$**

l

s

1.c ccittu.c~ Documents Music Pictures sample.odt te.cpp text array.c~ Desktop Downloads pcd Public s.doc Templates text1.c

bash-4.1**$**

- **date: DISPLAYING THE SYSTEM DATE**
- One can **display the current date** with the **date command**, which shows the date and time to the nearest second:
- Example: **$ date**

  Mon Sep 4 16:40:02 IST 2017
- The command can also be used with suitable format specifiers as arguments.
- Each symbol is preceded by the " **+** " symbol, followed by the " **%** " operator, and a single character describing the format.
- **Syntax:** $ date +%format_specifier
- For instance, you can print only the month using the format +%m:
- **Example**:

  $ date +%m

  09
- Or can print the month name:
- **Example:**

  $ date +%h

  Sep
- Or You can combine them in one command:
- **Example:**

  $ date + "%h %m"

  Sep 09

- There are many other format specifiers, and the useful ones are listed below:
  - **d – The day of month (1 - 31)**
  - **y – The last two digits of the year.**
  - **H, M and S – The hour, minute and second, respectively.**
  - **D – The date in the format *mm/dd/yy***
  - **T – The time in the format *hh:mm:ss***

Ls

passwd

(b) With suitable example bring out the differences between absolute and relative pathnames 6

Absolute Path:

- **Absolute path name:** The pathname begins with **/ & denotes the file location with respect to the root (/) is called absolute path name.**
- **Ex:** /home/student
- Directories are arranged in a hierarchy with root (/) at the top. The position of any file within the hierarchy is described by its pathname.
- Elements of a pathname are separated by a /.
- **A pathname is absolute, if it is described in relation to root, thus absolute pathnames always begin with a /.**
- Following are some examples of absolute filenames:
  - /etc/passwd
  - /users/student/pcd/binomial
  - /dev/rdsk/Os3

• **Relative Path:**

- A pathname can also be relative to your current working directory. Relative pathnames never begin with /. Relative to user home directory, some pathnames might look like this – cs/3cs ec/3ec.
- Using . and .. in relative path name –
  - User can move from working directory **/home/student/cs/3cs** to home directory **/home/student** using cd command like:

    $pwd
    /home/student/cs/3cs
    $cd /home/student
    $pwd
    /home/kumar
  - Navigation becomes easy by using **common ancestor.**
  - **(.) (a single dot) - This represents the current directory**
  - **(..) (two dots) - This represents the parent directory**
  - Assume user is currently placed in **/home/student/cs/3cs**

    $pwd
    /home/student/cs/3cs
    $cd ..
    $pwd
    /home/student/cs

  This method is compact and easy when ascending the directory hierarchy. The command: **cd .. Translates to this —change your current directory to parent of current directory**‖.
- The relative paths can also be used as: **$pwd**

  /home/student/cs/3cs
  $cd ../..
  $pwd
  /home

(c) Explain the basic file categories in Unix operating system?

- The UNIX has divided files into three categories:
    1. **Ordinary file** – also called as regular file. It contains only data as a stream of characters.
    2. **Directory file** – it contains files and other sub-directories.
    3. **Device file** – all devices and peripherals are represented by files.

  • **Ordinary File** – ordinary file itself can be divided into two types

    a) Text File – it contains only printable characters, and you can often view the contents and make sense out of them.
    Example: C, C++, Java files are text files

    b) Binary file – it contains both printable and unprintable characters that cover entire ASCII range.
    Examples: Most Unix commands, executable files, pictures, sound and video files are binary.

- **Directory File** –
    - ✦ a directory contains no data but keeps some details of the files and subdirectories that it contains.
    - ✦ A directory file contains an entry for every file and subdirectories that it houses.
    - ✦ If you have 20 files in a directory, there will be 20 entries in the directory.
    - ✦ Each entry has two components-
        a) The filename
        b) A unique identification number for the file or directory (called as inode number).

  • **Device File** –

    - ✦ All devices & peripherals are represented by files to read or to write a device.
    - ✦ Installing software from CD-ROM, printing files and backing up data files to tape.
    - ✦ All of these activities are performed by reading or writing the file representing the device.
    - ✦ Advantage of device file is that some of the commands used to access an ordinary file also work with device file.
    - ✦ Device filenames are generally found in a single directory structure, /dev.
    - ✦ Character Special (Device) File: Character device file is a physical device, that reads or writes one character at a time.
    - ✦ Ex: Terminal

- **Block Special (Device) File** – Block device file is a physical device, that reads or writes one block at a time. **Ex:** Disk

- **Symbolic Link (soft) file** – It is a logical file that defines the location of another file.

- **FIFO file** – A first-in first-out is also known as a PIPE used in inter-processor communication.

- **SOCKET file** – A socket file is special file that is used for network communication. A Unix socket (or Inter-process communication socket) is a special file which allows for advanced inter-process communication. A Unix Socket is used in a client-server application framework.

Module – 2

3.(a) Which command is for is used for listing file attributes ?explain the significance of each field in the attributes? 7

File Attributes

- ls command can be used to obtain a list of all filenames in the current directory.
- -l option can be used to obtain a detailed list of attributes of all files in the current directory. **Example:**

$ ls -l

Type & Perm	Link	Owner	Group	Size	Date & Time	File Name
-rwxr-xr--	1	kumara	metal	195	may 10 13:45	chap01
drwxr-xr-x	2	kumar	metal	512	may 09 12:55	helpdir

1.Type & Perm

- This represents the file-type and the permission given on the file.

File-Type

The first character indicates type of the file as shown below:

i) hyphen (-) for regular file ii) d for Directory file iii) l for

Symbolic link file iv) b for block special file v) c for

character special file

File Permission

The remaining 9 characters indicates permission of the file.

There are 3 permissions: read (r), write (w), execute (x).

i) Read: Grants the capability to read, i.e., view the contents of the file.

ii) Write: Grants the capability to modify, or remove the content of the file.

iii) Execute: User with execute permissions can run a file as a program.

There are 3 types of users: owner, groups and others.

- The permission is broken into group of 3 characters:

i) The first 3 characters (2-4) represent the permissions for the file's owner.

ii) The middle 3 characters (5-7) represent the permissions for the group to which the file belongs.

iii) The last 3 characters (8-10) represents the permissions for everyone else.

2) Link

• This indicates the number of file names maintained by the system.

• This does not mean that there are so many copies of the file. (Link similar to pointer).

3) Owner

• This represents the owner of the file. This is the user who created this file.

4) Group

• This represents the group of the owner.

• Each group member can access the file depending on the permission assigned.

• The privileges of the group are set by the owner of the file and not by the group members.

• When the system admin creates a user account, he has to assign these parameters to the user:

i) The user-id (UID) and ii) The group-id (GID)

5) Size

• This represents the file size in bytes.

• It is the number of character in the file rather than the actual size occupied on disk.

6) Date & Time

• This represents the last modification date and the time of the file.

• If you change only the permissions /ownership of the file, the modification time remains unchanged.

• If at least one character is added/removed from the file then this field will be updated.

7) File name :This represents the file or the directory name

(b) What are file permissions? Explain the use of chmod to change file permissions using both absolute and relative methods? 7

There are 3 permissions: read (r), write (w), execute (x).

iv) **Read:** Grants the capability to read, i.e., view the contents of the file.

v) **Write:** Grants the capability to modify, or remove the content of the file.

vi) **Execute:** User with execute permissions can run a file as a program.

vii) This command can be used in two ways: 1) Relative mode and 2) Absolute mode.

Relative Permissions

This command can be used to add/delete permission for specific type of user (owner, group or others).

• This command can be used to

→ change only those permissions specified in

the command line and → leave the other permissions

unchanged.

Syntax: chmod category operation

permission filename

This command takes 4 arguments:

viii) category can be u → user (owner)

g → group o → others a → all (ugo)

ix) operation can be

+ → assign

- → remove (refer pdf)

Absolute Permissions

- This command can be used to add/delete permission for all type of users (owner, group or others).
- This command can be used to change all permissions specified in the command line. **Syntax:** chimed octal_value filename This command takes 2 arguments:
- octal_value contains 3 octal digits to represent 3 type of users • Filename whose permission has to changed.

- **octal_value contains 3 octal digits to represent 3 type of users (owner, group or others).** • First digit is for user
- Second digit is for group and
- Third digit is for others

Each digit represents a permission as shown below:

- 4 (100) – read only
- 2 (010) – write only
- 1 (001) - execute only
- 6 (110) – read & write only
- For ex: octal_value of 644(110 100 100) means
- → user can read & write only
- → group can read only
- → others can read only

(c) Explain grep command?List its options with its significance 6

grep

- g/re/p means "globally search for a regular expression and print all lines containing it".
- This command can be used to search a file(s) for lines that have a certain pattern.
- This command

→ scans the file for a pattern and

→ displays

i)

lines containing the pattern

or ii) line numbers **Syntax:**

$grep

pattern file(s)

Example:

$grep "MH" student.lst // display lines containing "MH" from the file student.lst

- Patterns with and without quotes is possible.
- Quote is mandatory when pattern involves more than one word.

Example:

$grep "My Document" student.lst // display lines containing "My Document" from student.lst

- This command can be used with multiple filenames.

Example:

$grep "MH" student.lst vtu.lst rank.lst

-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
-x	matches pattern with entire line
-f	file takes patterns from file, one per line
-E	treats pattern as an extended RE
-F	matches multiple fixed strings

4.(a) Explain the concept of escaping and quoting with suitable example? 5

Removing the Special Meanings of Wild Cards (Escaping and Quoting)

Escaping is providing a \ (backslash) before the wildcard to remove its special meaning.

Example:

$ rm chap* // to remove file named "chap*" "\" is used to suppress special meaning of *

$ cat chap0\[1-3\] // to list the contents of the file named "chap0[1-3]"

$ rm My\ Document.doc // to remove file named "My Documend.doc"

$ echo \\ // outputs \

- Quoting is enclosing the wild-card within quotes to remove its special meaning.
- When a command argument is enclosed in quotes, the meanings of all enclosed special characters are turned off.

Example:

$ rm 'chap*' // to remove file named "chap*"

$ rm "My Document.doc" // to remove file named "My Documend.doc"

$ echo "\" // outputs "\"

(b) Explain three standard files supported by unix? Explain about special files used for output redirection? 10

Three Standard Files

- The shell associates three standard files with the terminal:

  → two for display and

  → one for the keyboard. • When a user

  logs in, the shell makes available three

  standard files.

- Each standard file is associated with a default device:

1) Standard input: The file representing input which is connected to the keyboard.

2) Standard output: The file representing output which is connected to the display. 3) Standard error: The file representing error messages that come from the command or shell

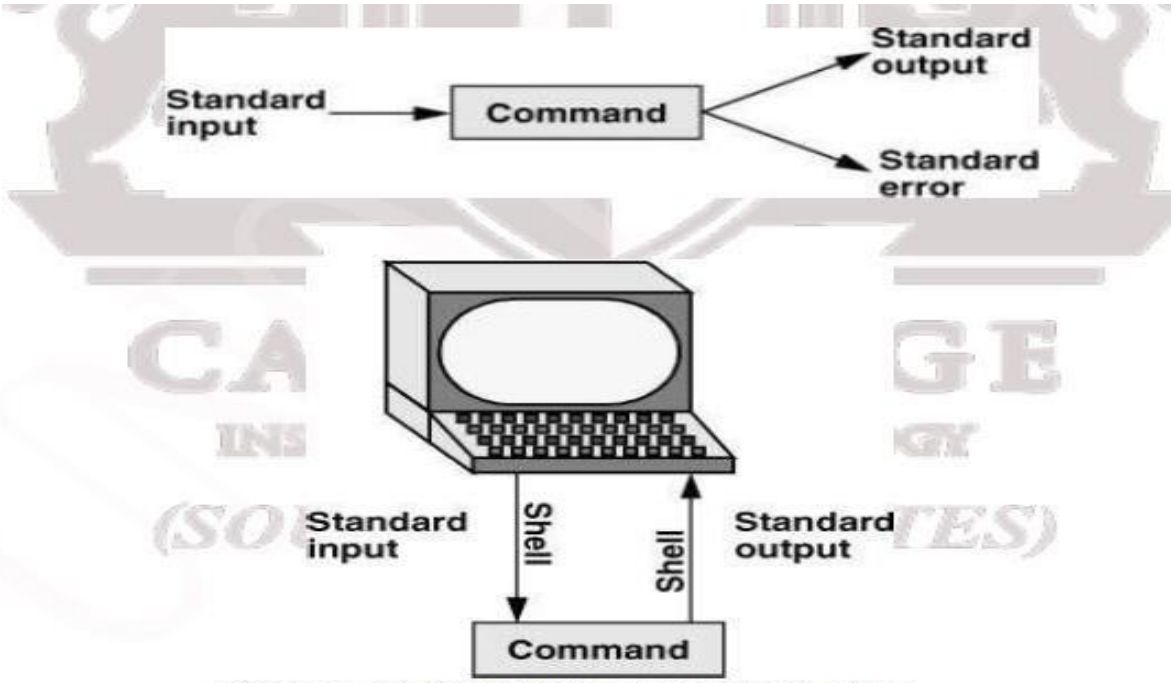


Figure : Default three Standard Files

(refer pdf)

(c) What are wild card characters? Explain shell wild card characters with example? 5

Wild Cards

- The metacharacters that are used to construct the generalized pattern for matching filenames belong to a category called wild-cards.

Wild Card/ Character class	Match
*	Any number of characters including none
?	A single character
[ijk]	A single character either an i, j or k
[x-z]	A single character that is within the ASCII range of the characters x and z
[!ijk]	A single character that is not an i, j, or k
[!x – z]	A single character that is not within the ASCII range of the characters x and z

Metacharacter *

The metacharacter "*" matches any number of characters including none.

Examples:

$ ls chap* // To list all files that begin with "chap" chap chap01 chap02

chap03 chap04 chapx chapy chapz

- When the shell encounters this command line, it identifies the * immediately as a wild-card. It then looks in the current directory and recreates the command line as below

$ ls chap chap01 chap02 chap03 chap04 chapx chapy chapz

- The shell now hands over this command to the kernel which uses its process creation facilities to run the command.

- The metacharacter ? matches a single character.

Metacharacter ?

- The metacharacter ? matches a single character.

Example:

$ ls chap? // To list all five-characters filenames

beginning with "chap" chapx chapy

chapz

$ ls chap?? // To list six-characters filenames beginning with "chap" chap01

chap02 chap03

- Both * and ? operate with some restrictions.
- The * doesn't match all filenames beginning with a dot (.) or forward slash(/).

Example:

$ ls .C* // to list all C extension filenames

$ cd /usr?local // this doesn't match /usr/local

5.(a) Describe how a c program is started and various ways it terminates. 10

PROCESS TERMINATION:

There are eight ways for a process to terminate.

1. Normal termination occurs in five ways:

✓ Return from main
✓ Calling exit
✓ Calling _exit or _Exit
✓ Return of the last thread from its start routine
✓ Calling pthread_exit from the last thread **2. Abnormal termination occurs in three ways:**
✓ Calling abort
✓ Receipt of a signal

✓ Response of the last thread to a cancellation request

A C program starts execution with a function called **main.**

The prototype for the main function is ***int main***
(int argc, char *argv[]);

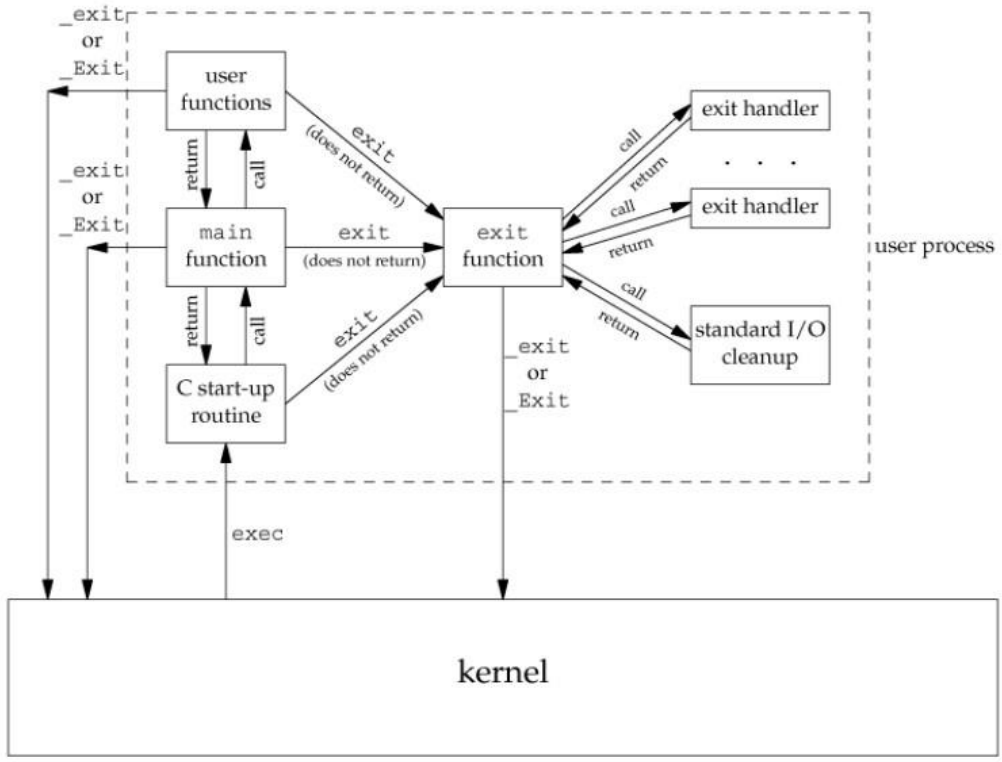
Where: argc is the number of command-line arguments.

argv is an array of pointers to the arguments.

✓ When a C program is executed by the kernel—by one of the exec functions a special **start-up routine** is called before the main function is called.

✓ The executable program file specifies this routine as the starting address for the program; this is set up by the link editor when it is invoked by the C compiler.

✓ This start-up routine takes values from the kernel—the command-line arguments and the environment—and sets things up so that the main function is called.



The exit handlers are:

a) exit:

✓ exit function terminates the process normally, performs certain cleanup processing and then returns to the kernel.

✓ the exit function has always performed a clean shutdown of the standard I/O library: the fclose function is called for all open streams. This causes all buffered output data to be flushed (written to the file).

b) exit and _Exit:

✓ _exit and _Exit functions terminate the process normally and return to the kernel immediately

All three exit functions expect a single integer argument, called the **exit status.**

The exit status of the process is undefined if

- Any of these functions is called without an exit status,
- main does a return without a return value, or
- The main function is not declared to return an integer.

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. **c) atexit Function:**

✓ 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 atexit function.

✓ **Prototype:** #include <stdlib.h>
int atexit(void (*func)(void)); Returns: 0 if
OK, nonzero on error

✓ This declaration says that the address of a function is passed as an 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.

(b) With neat sketch, explain memory layout of C program. 10

C program has been composed of the following pieces:

a) **Text segment,** consists of 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.

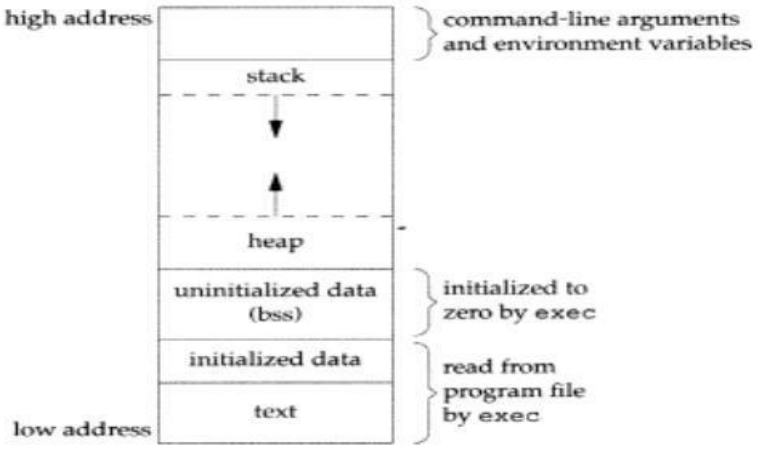
b) **Initialized data segment** usually called simply the data segment, containing variables that are specifically initialized in the program. For example, the C declaration ***int maxcount = 99;***
appearing outside any function causes this variable to be stored in the initialized data segment with its initial value.

c) **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.

d) **Stack** where (local variables)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.

e) **Heap** where dynamic memory allocation usually takes place. Historically, the heap has been located between the uninitialized data and the stack.



The memory layout of a c program

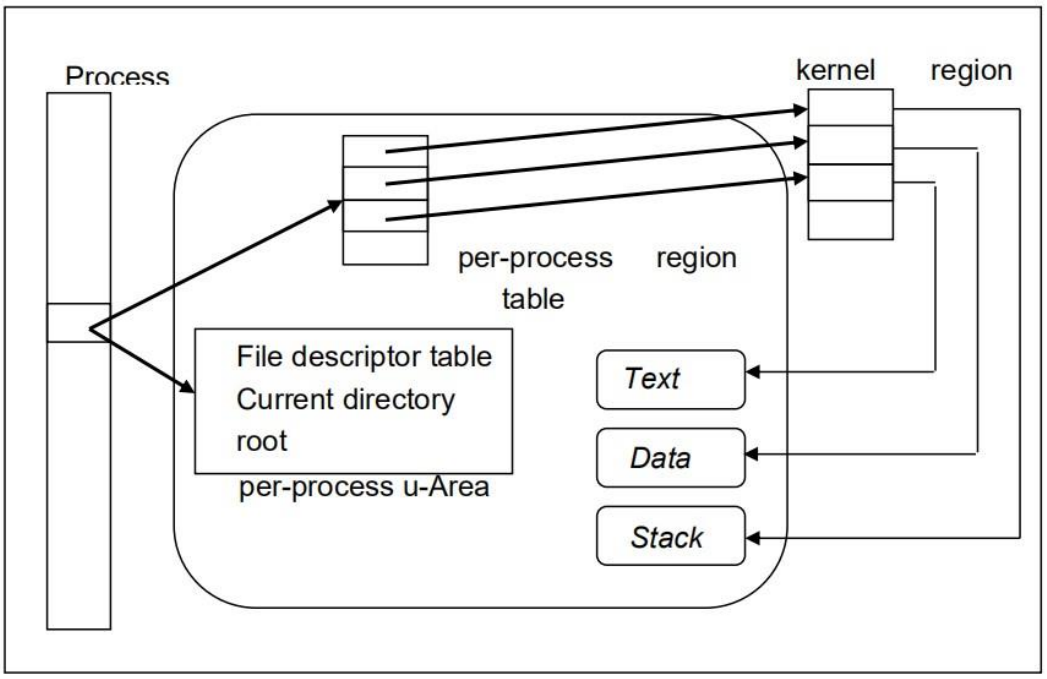
6.(a) With related data structures explain UNIX kernel support for a process.

✓ A UNIX kernel has a **process table** that keeps track of all active processes.
Some of the processes belong to the kernel, they are called **system processes**. The majority of processes are associated with the users who are logged in.

✓ Each entry in the process table contains pointers to the text, data, stack segments and the U-area of a process.

✓ The **U-area is an extension of a process table entry and contains other process specific data,** such as the file descriptor table, current root, and working directory inode numbers, and a set of system –imposed process resource limits, etc.

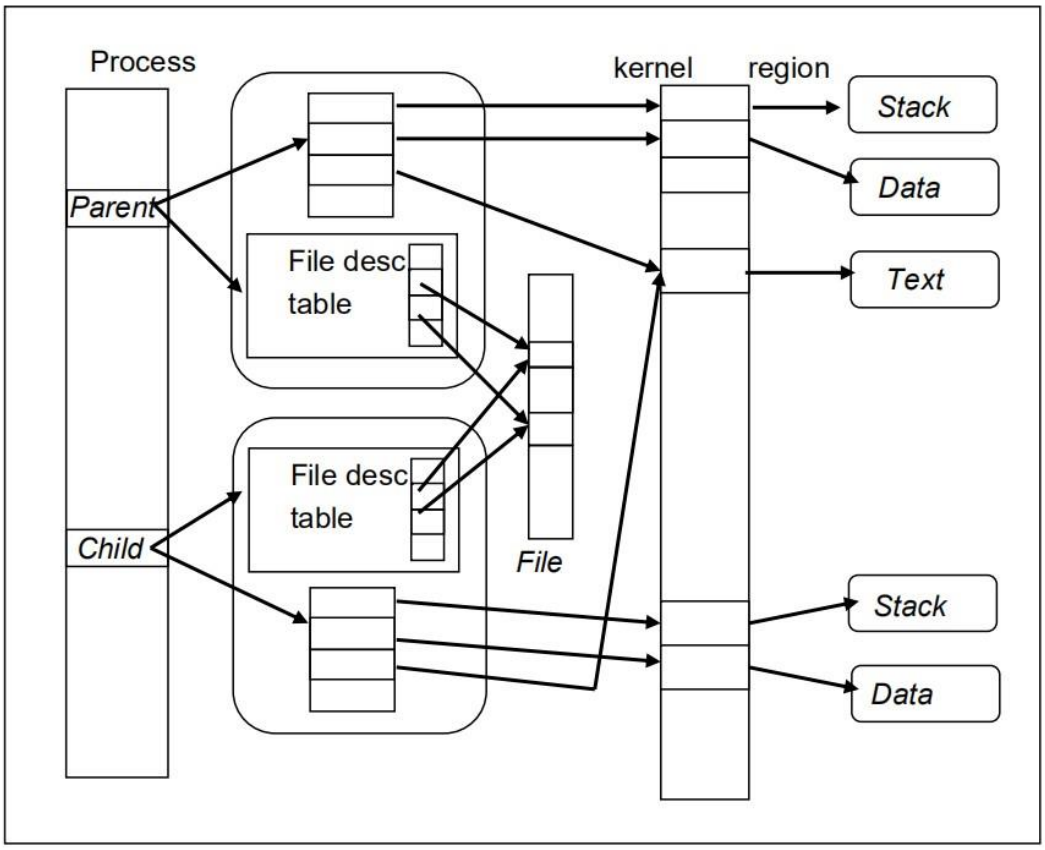
✓ All processes in UNIX system, except the first process (process 0) which is created by the system boot code, are created via the *fork* system call.



Unix Kernel support for process

✓ After the fork system call both the parent and child processes resume execution at the return of the fork function.

✓ As shown in the figure below, when a process is created by fork, it contains duplicate **copies of the text, data, and stack segments of its parent.** Also, it has an FDT that contains references to the same opened files as its parent, such that they both share the same file pointer to each opened file.



✓ the process is assigned the following attributes which are either inherited by from its parent or set by the kernel.

a) ***A real user identification number (rUID):*** the user ID of a user who created the parent process.

b) ***A real group identification number (rGID)*****:** the group ID of a user who created the parent process.

c) ***An effective user identification number (eUID)*****:** this is normally the same as the real UID, except when the file that was executed to create the process has its set UID flag turned on, in that case the eUID will take on the UID of the file. d) ***An effective group identification number (eGID)*****:** this is normally the same as the real GID, except when the file that was executed to create the process has its set UID flag turned on, in that case the eGID will take on the GID of the file.

e) ***Saved set-UID and saved set-GID*****:** these are the assigned eUID and eGID, respectively of the process.

f) ***Process group identification number (PGID) and session identification number (SID)*****:** these identify the process group and session of which the process is member.

g) ***Supplementary group identification numbers*****:** this is a set of additional group IDs for a user who created the process.

h) ***Current Directory*****:** this is the reference (inode number) to a working directory file.

i) ***Root Directory*****:** this is the reference (inode number) to a root directory file.

j) ***Signal handling*****:** the signal handling settings.

k) ***Signal mask*****:** a signal mask that specifies which signals are to be blocked.

l) ***Umask*****:** a file mode mask that is used in creation of files to specify which accession rights should be taken out.

m) ***Nice value*****:** the process scheduling priority value.

n) ***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.

d) ***Process identification number (PID)***: an integer identification number that is unique per process in an entire operating system.

e) ***Parent process identification number (PPID)*****:** the parent process ID.

f) ***Pending signals*****:** the set of signals that are pending delivery to the parent process. This is reset to none in the child process.

g) ***Alarm clock time*****:** the process alarm clock time is reset to zero in the child Process.

h) ***File locks*****:** the set of file locks owned by the parent process is not inherited by the child process.

✓ After *fork* a parent process may choose to suspend its execution until its child process terminates by calling the *wait* or *waitpid* system call, or it may continue execution independently of its child process.

✓ A process terminates its execution by calling the *_exit* system call. The argument to the *_exit* call is the exit status code of the process.

(b) What do you mean by fork and vfork functions. Explain both functions with example programs. 10

fork FUNCTION

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

```
#include <unistd.h>
pid_t fork(void);
```

Returns: 0 in child, process ID of child in parent, -1 on error.

- The new process created by fork is called the child process.
- This function is called once but returns twice.
- The only difference in the returns is that the return value in the child is 0, whereas the return value in the parent is the process ID of the new child.
- 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<unistd.h>
void main( )
{
fork( );
printf("\n hello USP");
}
```

Output :

```
$ cc fork1.c
$ ./a.out
hello USP
hello USP
```

vfork FUNCTION

- The function vfork has 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 exec or exit. When the child calls either of these functions, the parent resumes.

Example of vfork function

```
int glob = 6; /* external variable in initialized data */
int main(void)
{
int  var=88;        /* automatic variable on the stack */
pid_t     pid;
printf("before vfork\n");
if ((pid = vfork()) < 0)
    perror("vfork error");
else if (pid == 0)   /* child */
{
   glob++;              /* modify parent's variables */
   var++;
   _exit(0);         /* child terminates */
 }
```

```
/* Parent continues here.*/
printf("pid = %d\n", getpid());
printf("glob = %d, var = %d\n", glob, var);

exit(0);
}
```

Output:

```
$ ./a.out
before vfork
pid = 29039
 glob = 7, var = 89
```

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. 10

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 pipe function.

#include <unistd.h>

int pipe(int filedes[2]);

Returns: 0 if OK, 1 on error.

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

Two ways to picture a half-duplex pipe are shown in Figure. 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.

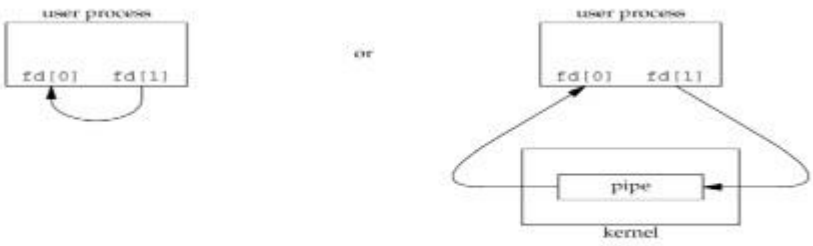


Figure : Two ways to view a half-duplex pipe

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. Below figure shows this scenario.

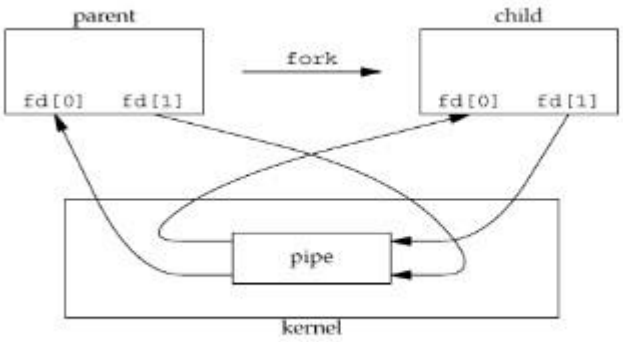


Figure: Half-duplex pipe after a fork

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 shows the resulting arrangement of descriptors.

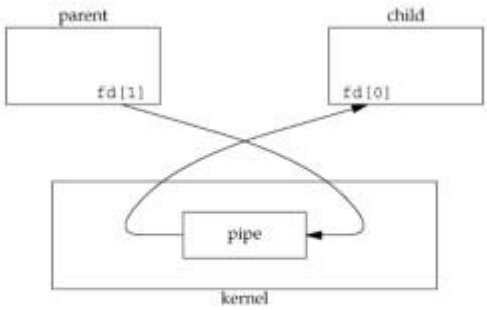


Figure: Pipe from parent to child

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 SIGPIPE is generated. If we either ignore the signal or catch it and return from the signal handler, write returns 1 with errno set 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);
}
```

(b) What is fifo? With a neat diagram explain the client server communication using fifo? 10

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

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's process 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 clients pecific 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.

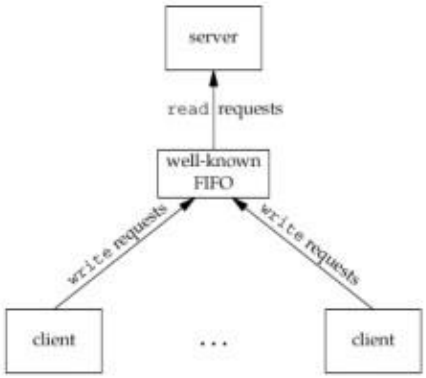


Figure : Clients sending requests to a server using a FIFO

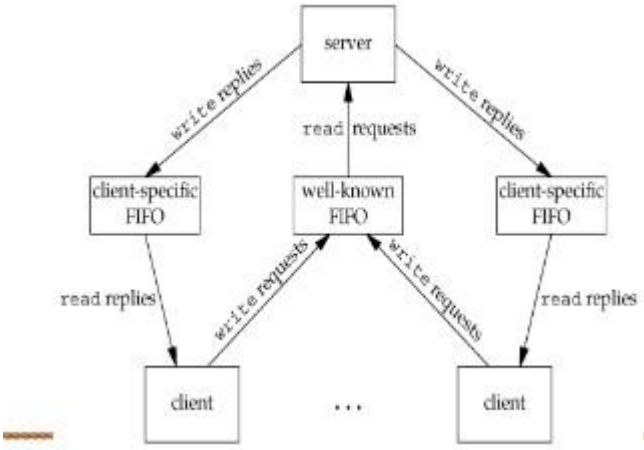


Figure : Client-server communication using FIFOs

8.(a) Explain briefly with example a) Message queue b) Semaphores. 10

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 nonnegative length, and the actual data bytes (corresponding to the length), all of which are specified to msgsnd when 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. This structure defines the current status of the

queue. The first function normally called is msgget to either open an existing queue or create a new queue.

#include <sys/msg.h>

int msgget(key_t key, int flag);

Returns: message queue ID if OK, 1 on error

When a new queue is created, the following members of the msqid_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.
- msg_qnum, msg_lspid, msg_lrpid, msg_stime, and msg_rtime are all set to 0.
- msg_ctime is set to the current time.
- msg_qbytes is 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 msgctl function performs various operations on a queue.

#include <sys/msg.h>

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

Returns: 0 if OK, 1 on error.

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

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_ds structure for each semaphore set:

```
struct semid_ds {
    struct ipc_perm     sem_perm; /* see Section 15.6.2 */
    unsigned time_t time_t
    .
    .
};
```

9.(b) Write a note on (i) Process accounting (ii) Process Times. 10

- Kernel writes an accounting record each time a process terminates.
- These accounting records are 32 bytes of binary data which contains the name of the command, amount of CPU time used, the user ID, group user ID, starting time and so on.
- *accton command* enables and disables process accounting.
- 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***
- The data required for the accounting record are all kept by the kernel in the process table and initialized whenever a new process is created (e.g., 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.
- 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 execed.

Process times

- Process time is also called as CPU time, which measures the CPU resources used by a particular process.
- Process time is measured in clock ticks (50, 60 or 100 ticks per second).
- To find clock ticks supported by our system the sysconf function can be used.
- Whenever we can measure the execution time of a process, UNIX maintains three values for a process
  a) **wall clock time:** amount of time a particular process takes to run.
  b) **user CPU time:** time taken by the CPU to execute the user instructions of a program.
  c) **system CPU time:** time taken by the CPU to execute the system calls.

9.(a) What are signals? Mention different source of signals? Write program to setup signal handlers for SIGINIT and SIGALRM 10

Signals are software interrupts. Signals provide a way of handling asynchronous events: a user at a terminal typing the interrupt key to stop a program or the next program in a pipeline terminating prematurely.

//source of signal and program

(b) Explain daemon characteristics and basic coding rules. 10

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. 10

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.

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

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.

```
#include <signal.h>
#include <iostream.h>
#include<unistd.h>
#include<signal.h>
#include<setjmp.h>
sigjmp_buf  env;

void callme ( int sig_num )
{ cout <<“catch signal:”<<sig_num<<
endl; siglongjump(env, 2);
}
int         main(void)
{
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 = callme;
action.sa_flags = 0;

if (sigaction (SIGINT, &action, &old_action) == -1)   perror("sigaction");
if (sigsetjmp(env,1)!=0) printf("return from signal
interruption"); else printf("return from first time
sigsetjmp is called"); pause(); /* wait for signal
interruption*/ return 0;
}
```

Execution:

```
$cc   -o   jmp      sigsetlongjmp.c

$ jmp  & [1]  499 return from first

time sigsetjmp is called

$kill  -INT   499
```