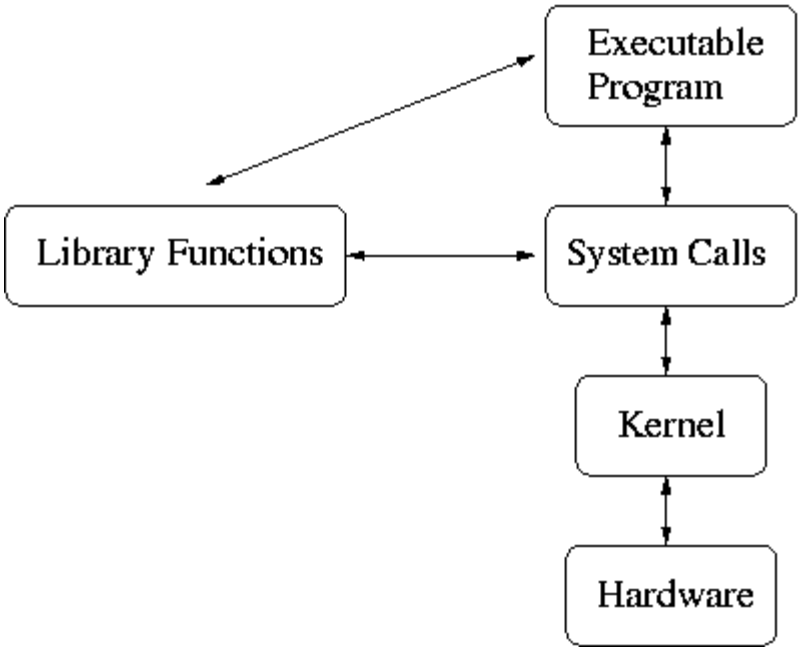
Q.2 a. Define System Calls. Explain with the help of Example System Calls. (8)
Answer:

A **system call** is how a program requests a service from an operating system's kernel. This may include hardware related services (e.g. accessing the hard disk), creating and executing new processes, and communicating with integral kernel services (like scheduling). System calls provide an essential interface between a process and the operating system.

systems provide a library or API that sits between normal programs and the operating system. On Unix-like systems, that API is usually part of an implementation of the C library (libc), such as glibc, that provides wrapper functions for the system calls, often named the same as the system calls that they call. On Windows NT, that API is part of the Native API, in the ntdll.dll library; this is an undocumented API used by implementations of the regular Windows API and directly used by some system programs on Windows. The library's wrapper functions expose an ordinary function calling convention (a subroutine call on the assembly level) for using the system call, as well as making the system call more modular. Here, the primary function of the wrapper is to place all the arguments to be passed to the system call in the appropriate processor registers (and maybe on the call stack as well), and also setting a unique system call number for the kernel to call. In this way the library, which exists between the OS and the application, increases portability.

On Unix, Unix-like and other POSIX-compliant operating systems, popular system calls are open, read, write, close, wait,execve, fork, exit, and kill. Many of today's operating systems have hundreds of system calls. For example, Linux andOpenBSD each have over 300 different calls,[1][2] NetBSD has close to 500,[3] FreeBSD has over 500,[4] while Plan 9 has 51.[5]

Tools such as strace and truss allow a process to execute from start and report all system calls the process invokes, or can attach to an already running process and intercept any system call made by said process if the operation does not violate the permissions of the user. This special ability of the program is usually also implemented with a system call, e.g. strace is implemented with ptrace or system calls on files in procfs.



b. Explain with the help of example the following functions: (8)
(i) open function
(ii) cat function
(iii) close function
(iv) lseek function
(v) read function

Answer:

1. open function

Ans. The **open()** system call is used to convert a pathname into a file descriptor (a small, non-negative integer for use in subsequent I/O as with **read**, **write**, etc.). When the call is successful, the file descriptor returned will be the lowest file descriptor not currently open for the process. This call creates a new open file, not shared with any other process. (But shared open files may arise via the **fork**(2) system call.) The new file descriptor is set to remain open across exec functions (see **fcntl**(2)). The file offset is set to the beginning of the file.

#include <<A HREF="file:/usr/include/sys/types.h">sys/types.h>

int open(const char **pathname*, **int** *flags*);

2. cat function

This is one of the most flexible Unix commands. We can use to create, view and concatenate files. For our first example we create a three-item English-Spanish dictionary in a file called "dict."

```
  % cat >dict
    red rojo
    green verde
    blue azul
<control-D>
  %
```

3. close function

close() closes a file descriptor, so that it no longer refers to any file and may be reused. Any record locks (see **fcntl**(2)) held on the file it was associated with, and owned by the process, are removed (regardless of the file descriptor that was used to obtain the lock).

If *fd* is the last copy of a particular file descriptor the resources associated with it are freed; if the descriptor was the last reference to a file which has been removed using **unlink**(2) the file is deleted.

RETURN VALUE

close() returns zero on success. On error, -1 is returned, and *errno* is set appropriately.

4. lseek function

The **lseek**() function repositions the offset of the open file associated with the file descriptor*fildes* to the argument *offset* according to the directive *whence* as follows:

Tag	Description
SEEK_SET	
	The offset is set to *offset* bytes.
SEEK_CUR	
	The offset is set to its current location plus *offset* bytes.
SEEK_END	
	The offset is set to the size of the file plus *offset* bytes.

The **lseek**() function allows the file offset to be set beyond the end of the file (but this does not change the size of the file). If data is later written at this point, subsequent reads of the data in the gap (a "hole") return null bytes (’\0’) until data is actually written into the gap.

RETURN VALUE
Upon successful completion, **lseek**() returns the resulting offset location as measured in bytes from the beginning of the file. Otherwise, a value of *(off_t)-1* is returned and *errno* is set to indicate the error.
5. read function

Q.3 a. How will you set permissions for a file? (8)

Answer:
Files and directories in Unix may have three types of permissions: read (`r'), write (`w'), and execute (`x'). Each permission may be `on' or `off' for each of three categories of users: the file or directory owner; other people in the same group as the owner; and all others.

Files
To determine the mode (or permission settings) of a particular file, use the command `ls -lg filename'. This command will produce a message similar to the following:

-rwxr-x--x 1 owner group 2300 Jul 14 14:38 filename

The string of 10 characters on the left shows the mode. The initial character ('-' in this case) indicates what type of file it is. A '-' indicates that the file is a plain file. The character 'd' means it is a directory. Characters 2-4 are, respectively, `r', `w', or `x' if the corresponding permission is turned on for the owner or `-' if the permission is turned off. Characters 5-7 similarly show the permissions for the group; characters 8-10 for all others. The second string shows the number of links that exist to the file. The third string identifies the owner of the file and the fourth string tells what group the owner of the file is in.

To change the mode of a file, use the chmod command. The general form is

chmod X@Y file1 file2 ...
where: X is any combination of the letters `u' (for owner), `g' (for group), `o' (for others), `a' (for all; that is, for `ugo'); @ is either `+' to add permissions, `-' to remove permissions, or `=' to assign permissions absolutely; and Y is any combination of `r', `w', `x'. Following are some examples:

```
chmod u=rx file        (Give the owner rx permissions, not w)
chmod go-rwx file      (Deny rwx permission for group, others)
chmod g+w file         (Give write permission to the group)
chmod a+x file1 file2  (Give execute permission to everybody)
chmod g+rx,o+x file    (OK to combine like this with a comma)
```

Directories

The permission scheme described above also applies to directories. For a directory, whoever has `read' permission can list files using the ls command (and thus discover what files are there); whoever has `write' permission can create and delete files in that directory; whoever has execute permission can access a file or subdirectory of known name. To find out the mode of a directory:

b. What are links and symbolic links in UNIX file system? (8)

Answer:

A link is a pointer or reference to another file. A directory in UNIX has a list of file names and their corresponding inodes. A directory entry can have an Inode pointing to another file.

This is a hard link. When a hard link is made, then the i-numbers of two different directory file entries point to the same inode.

A symbolic link or a soft link is a special type of file containing links or references to another file or directory in the form of a path. The path may be relative or absolute. To create a symbolic link, following command is used:

Ln –s target link_name

Here, target is the path and link_name is the name of the link. Symbolic links can be created to create a file system based on different views of the user.

Link is a utility program in UNIX which establishes a hard link from one directory to another directory. A hard link is a reference to a directory or to file on storage media. A symbolic link is a type of file. It contains references to another file directory in the form of absolute or a relative path.

Q.4 a. What are Streams? Explain the Standard Input, Output and Error and Buffering in Standard IO Library. (8)

Answer:

The standard I/O package provides a wide variety of functions to perform input, output, and associated tasks. It includes both standard functions and augmented functions to support 370-oriented features.

In general, a program that uses standard I/O accesses a file in the following steps:

1. Open the file using the standard function **fopen** or the augmented function **afopen** . This establishes a connection between the program and the external file. The name of the file to open is passed as an argument to **fopen** or **afopen** . The **fopen** and **afopen** functions return a pointer to an object of type **FILE** . (This type is defined in the header file **<stdio.h>** , which should be included with a**#include** statement by any program that uses standard I/O.) The data addressed by this **FILE** pointer are used to control all further program access to the file.

2. Transfer data to and from the file using any of the functions listed in this section. The **FILE** pointer returned by **fopen** is passed to the other functions to identify the file to be processed.
3. Close the file. After the file is closed, all changes have been written to the file and the **FILE** pointer is no longer valid. When a program terminates (except as the result of an ABEND), all files that have not been closed by the program are closed automatically by the library.

For convenience, three standard files are opened before program execution, accessible with the **FILE** pointers **stdin** , **stdout** , and **stderr** . These identify the standard input stream, standard output stream, and standard error stream, respectively. For TSO or CMS programs, these **FILE** objects normally identify the terminal, but they can be redirected to other files by use of command-line options. For programs running under the USS shell, these **FILE** objects reference the standard files for the program that invoked them. More information on the standard streams is available later in this section.

Standard I/O functions may be grouped into several categories. The functions in each category and their purposes are listed in Standard I/O Functions.

Standard I/O Functions	
Function	**Purpose**
Control Functions	control basic access to files
fopen+	opens a file
afopen*+	opens a file with system-dependent options
freopen+	reopens a file
afreopen*+	reopens a file with system-dependent options
tmpfile	creates and opens a temporary file
tmpnam	generates a unique filename
fflush	writes any buffered output data
afflush+	forces any buffered output data to be written immediately
fclose+	closes a file
setbuf+	changes stream buffering
setvbuf+	changes stream buffering
Character I/O Functions	read or write single characters

fgetc	reads a character
getc	reads a character (macro version)
ungetc	pushes back a previously read character
getchar	reads a character from **stdin**
fputc	writes a character
putc	writes a character (macro version)
putchar	writes a character to **stdout**
String I/O Functions	read or write character strings
fgets	reads a line into a string
gets	reads a line from **stdin** into a string
fputs	writes a string
puts	writes a line to **stdout**
Array I/O Functions	read or write arrays or objects of any data type
fread	reads on or more data elements
fwrite	writes one or more data elements
Record I/O Functions	read or write entire funtions
afread*	reads a record
afread0*	reads a record (possibly length 0)

In Standard I/O Functions,

- Functions marked with a *are not defined in the ANSI standard. Programs that use them should include **lcio.h** rather than **stdio.h** .
- Functions marked with a **+**may be used with files opened for keyed access.

UNIX Style I/O Overview

The UNIX style I/O package is designed to be compatible with UNIX low-level I/O, as described in previous sections. When you use UNIX style I/O, your program still performs the same three steps (open, access, and close) as those performed for standard I/O, but there are some important distinctions.

- To open a file using UNIX style I/O, you call **open** or **aopen** . (**aopen** is not compatible with UNIX operating systems but permits the program to specify 370-dependent file processing options.) The name of the file to open is passed as an argument to **open** or **aopen** .
- **open** and **aopen** return an integer called the file number (sometimes file descriptor). The file number is passed to the other UNIX style functions to identify the file. It indexes a table containing information used to access all files accessed with UNIX style I/O. Be sure to use the right kind of object to identify a file: a **FILE** pointer with standard I/O, but an integer file number with UNIX style I/O.

By convention, UNIX assigns the file numbers 0, 1, and 2 to the standard input, output, and error streams. Some programs use UNIX style I/O with these file numbers in place of standard I/O to **stdin** ,**stdout** , and **stderr** , but this practice is nonportable. The library attempts to honor this kind of usage in simple cases, but for the best results the use of standard I/O is recommended.

UNIX style I/O offers fewer functions than standard I/O. No formatted I/O functions or error-handling functions are provided. In general, programs that require elaborately formatted output or control of error processing should, where possible, use standard I/O rather than UNIX style I/O. Some UNIX style I/O functions, such as **fcntl** and **ftruncate** are supported only for files in the USS hierarchical file system.

The functions supported by UNIX style I/O and their purposes are listed in UNIX Style I/O Functions. Note that the **aopen** function is not defined by UNIX operating systems. Also note that some POSIX-defined functions, such as **ftruncate** , are not implemented by all versions of UNIX.

each form, valid abbreviations are given. (None of these forms can be used with the **xed** style.)

Alternate Forms	Abbreviations
TERMINAL	TERM,*
READER	RDR
PRINTER	PRT
PUNCH	PUN, PCH
%MACLIB (member **member-name**)	%MACLIB (**member-name**)
TXTLIB (member **member-name**)	%TXTLIB (**member-name**)

Standard I/O and UNIX Style I/O Open Modes

Standard form	UNIX style form
'r'	O_RDONLY | O_TEXT
'rb'	O_RDONLY
'r+'	O_RDWR | O_TEXT
'r+b'	O_RDWR
'w'	O_WRONLY | O_CREAT | O_TRUNC | O_TEXT
'wb'	O_WRONLY | O_CREAT | O_TRUNC
'w+'	O_RDWR | O_CREAT | O_TRUNC | O_TEXT
'w+b'	O_RDWR | O_CREAT | O_TRUNC
'a'	O_WRONLY | O_APPEND | O_CREAT | O_TEXT
'ab'	O_WRONLY | O_APPEND | O_CREAT
'a+'	O_RDWR | O_APPEND | O_CREAT | O_TEXT
'a+b'	O_RDWR | O_APPEND | O_CREAT

Library access method selection

When you use **afopen** or **afreopen** to open a file, you can specify the library access method to be used. If you use some other open routine, or specify the null string as the access method name, the library selects the most appropriate access method for you. If you specify an access method that is incompatible with the attributes of the file being opened, the open fails, and a diagnostic message is produced. Six possible access method specifications are available:

- A null (**""**) access method name allows the library to select an access method.
- The **"term"** access method applies only to terminal files.
- The **"seq"** access method is primarily oriented towards sequential access. (**"seq"** may also be specified for terminal files, in which case, the **"term"** access method is automatically substituted.)
- The **"rel"** access method is primarily oriented toward access by relative character number. The **"rel"** access method can be used only when the open mode specifies binary access. Additionally, the external file must have appropriate attributes, as discussed in 370 Perspectives on SAS/C Library I/O.
- The **"kvs"** access method provides keyed access to VSAM files.

- The **"fd"** access method provides access to USS hierarchical file system files.

When no specific access method is requested by the program, the library selects an access method as follows:

- **"term"** for a TSO or CMS terminal file
- **"kvs"** if the open mode specifies keyed access
- **"fd"** for a hierachical file system file
- **"rel"** if the open mode includes binary access and the file has suitable attributes
- **"seq"** otherwise.

Terminal Options

eof=string
end-of-file string
prompt=string
terminal input prompt

VSAM Performance Options

bufnd=nnn
number of data I/O buffers VSAM is to use
bufni=nnn
number of index I/O buffers VSAM is to use
bufsp=nnn
maximum number of bytes of storage to be used by VSAM for file data and index I/O buffers
bufsize=nnn
size, in bytes, of a DIV window for a linear data set
bufmax=n
number of DIV windows for a linear data set

See Terminal I/O for a discussion of the **eof** and **prompt** amparms. See VSAM-related amparms for a discussion of the VSAM Performance amparms.

The default amparms vary greatly between OS/390 and CMS, so they are described separately for each system.

File characteristics amparms The **recfm** , **reclen** , **blksize** , **keylen** , **keyoff** , and **org** keywords specify the program's expectations for record format, maximum r

You can use the **xed** style even when XEDIT is not active. In this case, or when the file requested is not in the XEDIT ring, the file is read from disk.

Table: Printf/scanf format characters

Format Spec (%)	Type	Result

c	char	single character
i,d	int	decimal number
o	int	octal number
x,X	int	hexadecimal number
		lower/uppercase notation
u	int	unsigned int
s	char *	print string
		terminated by \0
f	double/float	format -m.ddd...
e,E	"	Scientific Format
		-1.23e002
g,G	"	e or f whichever
		is most compact
%	-	print % character

b. Write short notes on the following terms: (8)
(i) Binary I/O
(ii) Formatted I/O

Answer:

Q.5 a. Explain Kill and Raise functions with the help of examples. (8)

Answer:

Advanced Programming in UNIX by W. Richard, Sec. 10.9 (Page No. 311 – 313)

b. Describe sigaction function with the help of example. (8)

Answer:

Advanced Programming in UNIX by W. Richard, Sec. 10.14 (Page No. 324 – 328)

Q.6 a. What does fork() do? What is the difference between fork() and vfork()? (8)

Answer:

Fork : The fork call basically makes a duplicate of the current process, identical in almost every way (not everything is copied over, for example, resource limits in some implementations but the idea is to create as close a copy as possible).

The new process (child) gets a different process ID (PID) and has the the PID of the old process (parent) as its parent PID (PPID). Because the two processes are now running exactly the same code, they can tell which is which by the return code of fork - the child gets 0, the parent gets the PID of the child. This is all, of course, assuming the fork call works - if not, no child is created and the parent gets an error code.

Vfork : The basic difference between vfork and fork is that when a new process is created with vfork(), the parent process is temporarily suspended, and the child process might borrow the parent's address space. This strange state of affairs continues until the child process either exits, or calls execve(), at which point the parent process continues.

This means that the child process of a vfork() must be careful to avoid unexpectedly modifying variables of the parent process. In particular, the child process must not return from the function containing the vfork() call, and it must not call exit() (if it needs to exit, it should use _exit(); actually, this is also true for the child of a normal fork()).

b. Difference between wait and waitpid. (4)

Answer:

The system calls wait() and waitpid() are used to synchronize a parent process with the termination of its child processes. The parent blocks until a child process dies. The difference is that wait() waits for the termination of any child process while waitpid() allows to specify the child process by its PID. The wait system call has some closely related variants, wait3() and wait4() which allow to gather more information about the process and its termination and to supply options for the waiting.

wait() allows a process to wait until one of its child processes change its state, exists for example. If waitpid() is called with a process id it waits for that *specific* child process to change its state, if apid is not specified, then it's equivalent to calling wait() and it waits for *any* child process to change its state.

The wait() function returns child pid on success, so when it's is called in a loop like this:

```
while(wait(NULL)>0)
```

It means wait until all child processes exit (or change state) and no more child processes are unwaited-for (or until an error occurs)

c. Explain the term Process Accounting. (4)

Answer:

Unix Process Accounting is a tool that literally takes account of every single process on your server, recording how much CPU (processor) and memory (RAM) each process consumes. When summarized, this information can be very useful for tracking down overzealous processes and hogs of resources so that you may restore your server to it's rightful speed and performance!

DreamHost keeps a daily accounting of your processes, which can be found in /var/log/sa/. For example, /var/log/sa/sa.itemized.0 is an itemized report of all the processes run from yesterday, sa.itemized.1 is the itemized report for the day before yesterday, sa.itemized.2 is the itemized report for two days before yesterday. Along those same lines is sa.summary.0, which is a summary of each user's consumption of resources.

> **Note:** most web server users can only access files within their /home/username directory. VPS or dedicated servers can have admin users who can use sudo to access logs in /var/logs.

You may have the daily reports of your processes in your home directory, in the logs/resources/ directory, by enabling CPU reporting for the user. See Where can I find resource reports?for more information.

What does all of this mean? The best way to use this information is to examine the CPU-minutes. A CPU-minute can be considered a unit of work for a server. No, CPU-minutes are not "real time", that is, there are not 60minutesX24hours=1440 CPU-minutes in a day. If a machine is idle, doing nothing at all, that doesn't count towards CPU time. Only when the processor is actually doing something does it count as CPU-minutes.

So knowing how many CPU-minutes per day your users and processes are consuming can help you to find the user or process which is causing problems for the entire machine. The best way to understand all of this is to breeze through the examples below, and then try it out on your own machine.

Q.7 a. What are Command line Arguments? (4)

Answer:

The command-line arguments $1, $2, $3,...$9 are positional parameters, with $0 pointing to the actual command, program, shell script, or function and $1, $2, $3, ...$9 as the arguments to the command.

Following script uses various special variables related to command line:

```
#!/bin/sh

echo "File Name: $0"
echo "First Parameter : $1"
echo "First Parameter : $2"
echo "Quoted Values: $@"
echo "Quoted Values: $*"
echo "Total Number of Parameters : $#"
```

Many Unix commands allow **command line arguments**, that is, additional words or phrases that follow the name of the command and give more information about what to do.

For example, the man command will give you information about any Unix command. To get information about the date command, for example, you can type:

```
yourname@yourcomputer<21>% man date
```

A command can have any number of arguments. For example, the cat command con*cat*enates any number of files. Suppose you want to display the contents of three files named names, dates, andaddresses on your display screen. This command would do it:

```
yourname@yourcomputer<22>% cat names dates addresses
```

b. How to set Environment Variables? (4)

Answer:

Environment variables defined in this chapter affect the operation of multiple utilities, functions, and applications. There are other environment variables that are of interest only to specific utilities. Environment variables that apply to a single utility only are defined as part of the utility description. See the ENVIRONMENT VARIABLES section of the utility descriptions in the Shell and Utilities volume of IEEE Std 1003.1-2001 for information on environment variable usage.

The value of an environment variable is a string of characters. For a C-language program, an array of strings called the environment shall be made available when a process begins. The array is pointed to by the external variable *environ*, which is defined as:

```
extern char **environ;
```

These strings have the form *name*=*value*; *name*s shall not contain the character '='. For values to be portable across systems conforming to

IEEE Std 1003.1-2001, the value shall be composed of characters from the portable character set (except NUL and as indicated below). There is no meaning associated with the order of strings in the environment. If more than one string in a process' environment has the same *name*, the consequences are undefined.

c. Difference between getrlimit and setrlimit functions. (8)

Answer:

Every process has a set of resource limits, some of which can be queried and changed by the geTRlimit and setrlimit functions.

[View full width]

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

int getrlimit(int resource, struct rlimit *rlptr);

int setrlimit(int resource, const struct rlimit
➥ *rlptr);
```

Both return: 0 if OK, nonzero on error

These two functions are defined as XSI extensions in the Single UNIX Specification. The resource limits for a process are normally established by process 0 when the system is initialized and then inherited by each successive process. Each implementation has its own way of tuning the various limits.

Each call to these two functions specifies a single *resource* and a pointer to the following structure:

```
struct rlimit {
  rlim_t  rlim_cur;   /* soft limit: current limit */
  rlim_t  rlim_max;   /* hard limit: maximum value for rlim_cur */
};
```

Three rules govern the changing of the resource limits.

1. A process can change its soft limit to a value less than or equal to its hard limit.
2. A process can lower its hard limit to a value greater than or equal to its soft limit. This lowering of the hard limit is irreversible for normal users.
3. Only a superuser process can raise a hard limit.

An infinite limit is specified by the constant RLIM_INFINITY.

Q.8 a. How to get and set the attributes of Terminal? (8)

Answer:

ll of the attributes that can be controlled in the terminal device are contained in the termios structure. This structure is defined as:

```
struct termios {
    tcflag_t    c_iflag;    /* input flags */
    tcflag_t    c_oflag;    /* output flags */
    tcflag_t    c_cflag;    /* control flags */
    tcflag_t    c_lflag;    /* local flags */
    cc_t        c_cc[NCCS]; /* control characters */
};
```

The c_iglag attribute is what controls any input characteristics of the terminal (map CR to NL, ring bell on input queue full, etc.). The c_oflag attribute is what you set to control any output processing of the terminal (expand tabs to spaces, map lowercase to uppercase on output, etc.). Most of the c_oflag settings are not Posix compliant. The c_cflag attribute is for setting the serial line attributes (enable parity, set flow control, etc.). The c_lflag attribute is for the settigns of the interface between the user and the device driver (local echo, enable signals generated byt the terminal, etc.).

This structure is used with two different functions, tcgetattr() and tcsetattr(). The prototypes are as follows:

```
#include <termios.h>

int tcgetattr(int filedes, struct termios *termptr);

int tcsetattr(int filedes, int opt, const struct termios *termptr);

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

As the names suggest, tcgetattr() gets the current state of the terminal that the open file descriptor *filedes* points to, and tcsetattr() sets attributes for the terminal that *filedes* is associated with. These functions will return an error if the *filedes* argument is not associated with a terminal device.

The argument *opt* in tcsetattr() is for specifying when the changes are to take place. This is defined by the following macros:

TCSANOW	Make the changes now.
TCSADRAIN	Make the changes after all output has been transmitted from the buffer. This should be used when setting output attributes.
TCSAFLUSH	Make the changes after all output has been transmitted, and flush the input queue of any unprocessed data.

How fast are we talking?

Sometimes you may find that you need to change the speed of the terminal dsession to match that of the device it is connected to. This is done with four functions in combination with the tcgetattr() andtcsetattr() functions.

```
#include <termios.h>

speed_t cfgetispeed(const struct termios *termptr);

speed_t cfgetospeed(const struct termios *termptr);
Both return: baud rate value
speed_t cfsetispeed(struct termios *termptr, speed_t speed);

speed_t cfsetospeed(struct termios *termptr, speed_t speed);
Both return: 0 if OK, -1 on error
```

The first thing that must be done here in order to change the baud rate of the terminal is use tcgetattr() so that you can pass the termios struct to the cfset functions. You then pass the struct to the cfset functions to set the correct baud rate in the termios struct. This does not actually set the terminal speed, however. You still need to make a call to tcsetattr() with termios struct that has the changed baud rate.

The order of calls to change the baud rate:

1. tcgetattr() -- Get the current settings
2. cfsetispeed() -- Set the input speed in the termios struct
3. cfsetospeed() -- Set the output speed in the termios struct
4. cfsetattr() -- Make the changes to the terminal to reflect the changed struct

Terminal line control

The line control for the terminal is important if you want to prevent overflowing the buffer for the device when there is no hardware flow control implemented. Also, you can flush the input and/or output of a device discarding any data that has not already been sent or read from the buffer.

```
#include <termios.h>

int tcdrain(int filedes);

int tcflow(int filedes, int action);

int tcflush(int filedes, int queue);

int tcsendbreak(int filedes, int duration);
All four return: 0 if OK, -1 on error
```

The tcdrain() function suspends the process until all of the data in the ouput buffer has been transmitted. The tcflow() function gives control over input and output flow control. The *action* argument totcflow() can be any of the following macros:

TCOOFF	Suspend Output
TCOON	Restart output
TCIOFF	Suspend input
TCION	Restart input

The tcflush() function lets us discard input or output buffer data. Data in the input buffer is data that has been received but not read yet. Data in the output buffer is data that has been written but not transmitted yet. The *queue* argument can have the follow macro values:

TCIFLUSH	Flush the input buffer
TCOFLUSH	Flush the output buffer
TCIOFLUSH	Flush both the input and output buffers

The tcsendbreak() function transmits a continous stream of zero bits. If the *duration* attribute is set to 0, then the duration of the transmition is between 0.25 and 0.5 seconds. If the *duration* is nonzero, it is implementation specific. Under Linux, if the *duration* is nonzero, the length of transmission is ***duration**N** seconds where N is between 0.25 and 0.5.

b. Explain in detail Client Server Model. (8)

Answer:

Client Process:

This is the process which typically makes a request for information. After getting the response this process may terminate or may do some other processing.

For example: Internet Browser works as a client application which sends a request to Web Server to get one HTML web page.

Server Process:

This is the process which takes a request from the clients. After getting a request from the client, this process will do required processing and will gather requested information and will send it to the requestor client. Once done, it becomes ready to serve another client. Server process are always alert and ready to serve incoming requests.

For example: Web Server keeps waiting for requests from Internet Browsers and as soon as it gets any request from a browser, it picks up a requested HTML page and sends it back to that Browser.

Notice that the client needs to know of the existence and the address of the server, but the server does not need to know the address or even the existence

of the client prior to the connection being established. Once a connection is established, both sides can send and receive information.

2-tier and 3-tier architectures:
There are two types of client server architectures:

- **2-tier architectures:** In this architecture, client directly interact with the server. This type of architecture may have some security holes and performance problems. Internet Explorer and Web Server works on two tier architecture. Here security problems are resolved using Secure Socket Layer(SSL).
- **3-tier architectures:** In this architecture, one more software sits in between client and server. This middle software is called middleware. Middleware are used to perform all the security checks and load balancing in case of heavy load. A middleware takes all requests from the client and after doing required authentication it passes that request to the server. Then server does required processing and sends response back to the middleware and finally middleware passes this response back to the client. If you want to implement a 3-tier architecture then you can keep any middle ware like Web Logic or WebSphere software in between your Web Server and Web Browsers.

Types of Server:
There are two types of servers you can have:

- **Iterative Server:** This is the simplest form of server where a server process serves one client and after completing first request then it takes request from another client. Meanwhile another client keeps waiting.
- **Concurrent Servers:** This type of server runs multiple concurrent processes to serve many request at a time. Because one process may take longer and another client can not wait for so long. The simplest way to write a concurrent server under Unix is to *fork* a child process to handle each client separately.

How to make client:
The system calls for establishing a connection are somewhat different for the client and the server, but both involve the basic construct of a socket. The two processes each establish their own sockets.

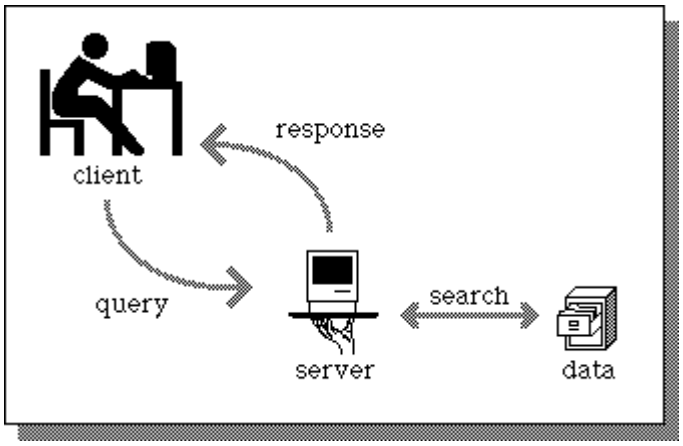
The steps involved in establishing a socket on the client side are as follows:

1. Create a socket with the *socket()* system call.
2. Connect the socket to the address of the server using the *connect()* system call.
3. Send and receive data. There are a number of ways to do this, but the simplest is to use the *read()* and *write()* system calls.

How to make a server:
The steps involved in establishing a socket on the server side are as follows:

1. Create a socket with the *socket()* system call.
2. Bind the socket to an address using the *bind()* system call. For a server socket on the Internet, an address consists of a port number on the host machine.
3. Listen for connections with the *listen()* system call.
4. Accept a connection with the *accept()* system call. This call typically blocks until a client connects with the server.
5. Send and receive data using the *read()* and *write()* system calls.

Client and Server Interaction:
Following is the diagram showing complete Client and Server interaction:

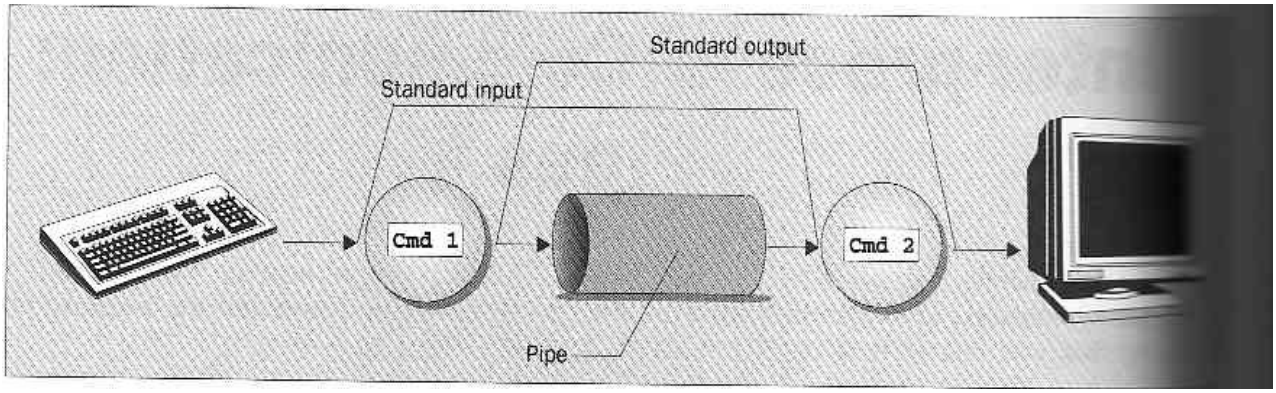


Q.9 a. Define Pipes. What are popen and pclose functions in Interprocess communication? (8)

Answer:
The shell arranges the standard input and output of the two commands, so that:

- The standard input to **cmd1** comes from the terminal keyboard.
- The standard output from **cmd1** is fed to **cmd2** as its standard input.
- The standard output from **cmd2** is connected to the terminal screen.

The shell has reconnected the standard input and output streams so that data flows from the keyboard input through the two commands and is then output to the screen.



Process Pipes

Perhaps the simplest way of passing data between two programs is with the **popen** and **pclose** functions. These have the prototypes:

```
#include <stdio.h>

FILE *popen(const char *command, const char *open_mode);
int pclose(FILE *stream_to_close);
```

popen

The **popen** function allows a program to invoke another program as a new process and either pass data to or receive data from it.

pclose

When the process started with **popen** has finished, we can close the file stream associated with it using **pclose**.

Try It Out - Using popen and pclose

Having initialized the program, we open the pipe to **uname**, making it readable and setting **read_fp** to point to the output.

At the end, the pipe pointed to by **read_fp** is closed

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main()
{
    FILE *read_fp;
    char buffer[BUFSIZ + 1];
    int chars_read;
    memset(buffer, '\0', sizeof(buffer));
    read_fp = popen("uname -a", "r");
    if (read_fp != NULL) {
        chars_read = fread(buffer, sizeof(char), BUFSIZ, read_fp);
        if (chars_read > 0) {
            printf("Output was:-\n%s\n", buffer);
        }
        pclose(read_fp);
        exit(EXIT_SUCCESS);
    }
    exit(EXIT_FAILURE);
}
```

When we run this program on one of the author's machine, we get:

```
$ popen1
Output was:-
Linux stones 1.2.8 #1 Mon Sep 18 18:20:08 BST 1995 i586
```

How It Works

The program uses the **popen** call to invoke the **uname** command. It read some information and prints it to the screen.

Sending Output to popen

Here's a program, **popen2.c**, that pipes dta to another. Here, we use the **od** (octal dump).

```
$ popen3
Reading:-
   PID TTY STAT  TIME COMMAND
     1  ?  S     0:00 init
     6  ?  S     0:00 bdflush (daemon)
     7  ?  S     0:00 update (bdflush)
    24  ?  S     0:00 /usr/sbin/crond -l10
    39  ?  S     0:00 /usr/sbin/syslogd
...
   240 v02 S     0:02 emacs draft1.txt
Reading:-
   368 v04 S     0:00 popen3
   369 v04 R     0:00 ps -ax
...
```

How It Works

The progran uses **popen** with an **r** parameter, so it continues reading from the file stream until there is no more data available.

How popen is Implemented

The **popen** call runs the program you requested by first invoking the shell, **sh**, passing it the **command** string as an argument.

This has two effects, one good, the other not so good.

1. invoking the shell allows complex shell commands to be started with **popen**.

2. Each call to **popen** invokes the requested program and the shell program. So, each call to **popen** then results in two extra processes being started.

We can count all the lines in example program by **cat**ing

the files and then piping its output to **wc -1** , which counts the number of lines.
On the command line, we would use:

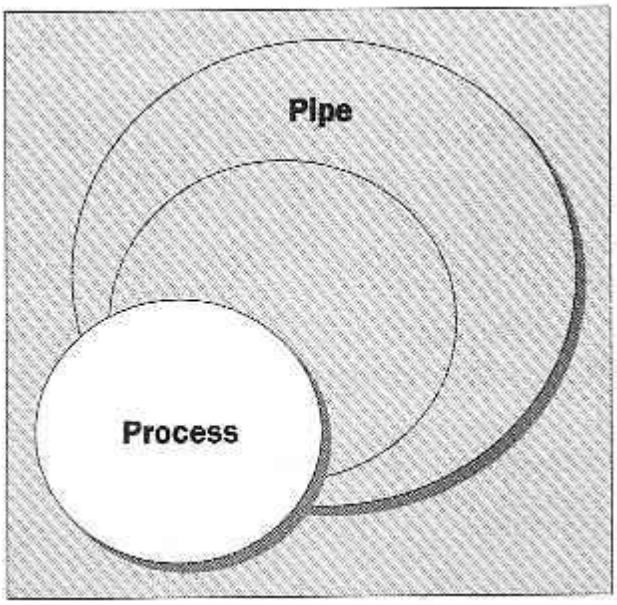
```
$ pipe5
1239 - wrote 3 bytes
0000000   1   2   3
0000003
```

How It Works

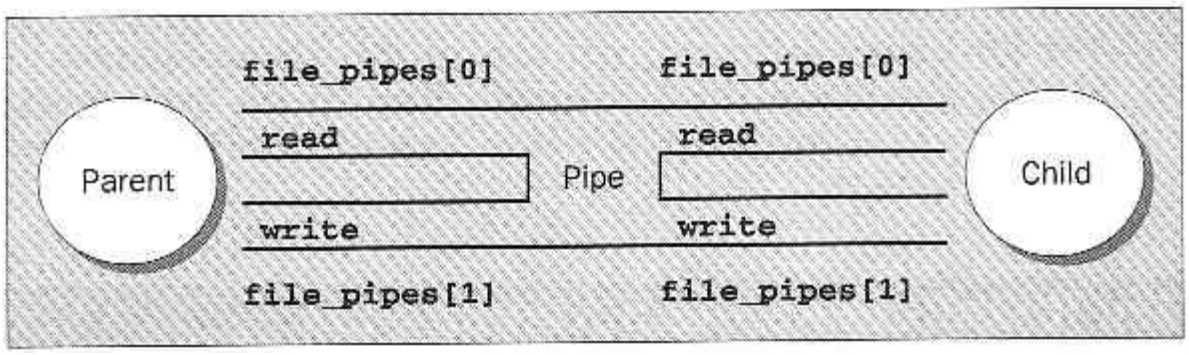
The program creates a pipe and then forks, creating a child process.

The parent and child have access to the pipe.

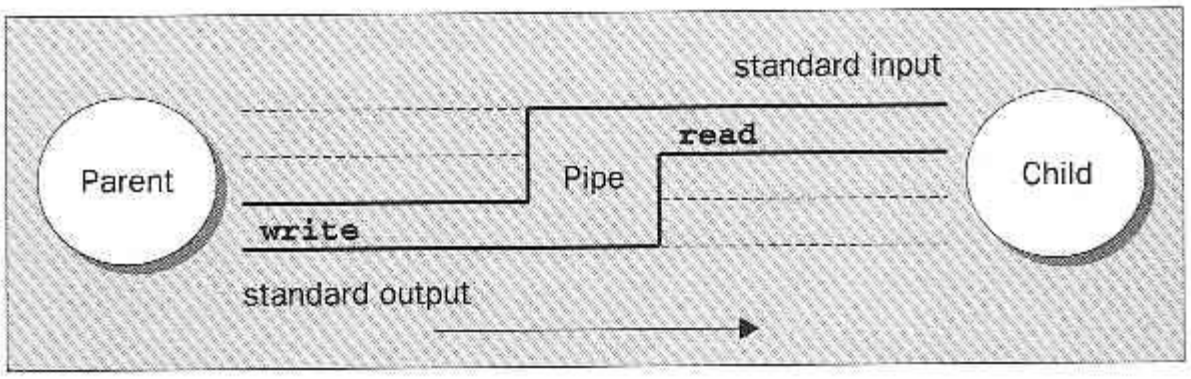
We can show the sequence pictorially. After the call to **pipe**:



After the call to **fork**:



When the program is rady to transfer data:



Named Pipes: FIFOs

We can exchange data with **FIFOs**, often referred to as **named pipes**.

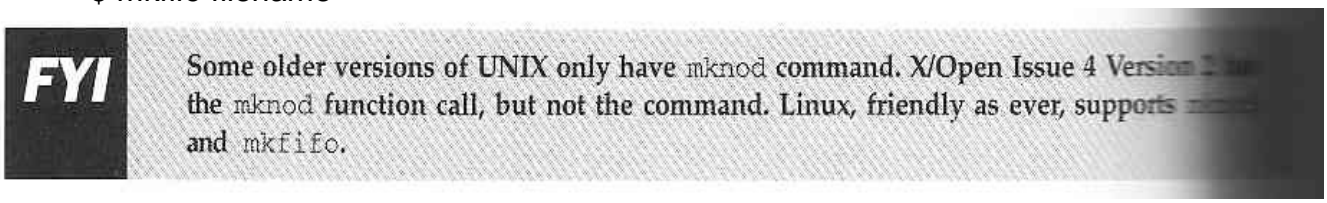
A named pipe is a special type of file that exists as a name in the file system, but behaves like the unnamed pipes that we've met already.

We can create a named pipe using the old UNIX **mknod** command:

```
$ mknod filename p
```

However, it is not in X/Open/ command list, so we use the **mkfifo** command:

```
$ mkfifo filename
```



From inside a program, we can use two different calls. These are:

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

int mkfifo(const char *filename, mode_t mode);
int mknod(const char *filename, mode_t mode | S_IFIFO, (dev_t) 0);
```

Try It Out - Creating a Named Pipe

For **fifo1.c**, just type in the following code:

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>

int main()
{
    int res = mkfifo("/tmp/my_fifo", 0777);
    if (res == 0) printf("FIFO created\n");
    exit(EXIT_SUCCESS);
}
```

We can look for the pipe with:

```
$ ls -lF /tmp/my_fifo
prwxr-xr-x   1 rick     users            0 Dec 10 14:55 /tmp/my_fifo|
```

How It Works

The program uses the **mkfifo** function to create a special file.

Accessing a FIFO

One very useful feature of named pipes is that, because they appear in the file system, we can use them in commands where we would normally use a file name.

Try It Out - Accessing a FIFO File

1. First, let's try reading the (empty) FIFO:

```
$ cat < /tmp/my_fifo
```

2. Now try writing to the FIFO:

```
$ echo "sdsdfasdf" > /tmp/my_fifo
```

3. If we do both at once, we can pass information through the pipe:

```
$ cat < /tmp/my_fifo &
[1] 1316
$ echo "sdsdfasdf" > /tmp/my_fifo
sdsdfasdf

[1]+  Done                    cat </tmp/my_fifo
$
```

NOTICE: the first two stages simply hang until we interrupt them with Ctrl-C.

How It Works

Since there was no data in the FIFO, the **cat** and **echo** programs blocks, waiting for some data to arrive and some other process to read the data, respectively.

The thrid stage works as expected.

Unlike a pipe created with the **pipe** *call, a FIFO exists as a named file, not as an open file descriptor, and must be opened before it can be read from or written to. You open and close a FIFO using the same* **open** *and* **close** *functions that we saw used earlier for files, with some additional functionality. The* **open** *call is passed the path name of the FIFO, rather than that of a regular file.*

Opening a FIFO with open

The main restriction on opening FIFOs is that a program may not open a FIFO for reading and writing with the mode **O_RDWR**.

A process will read its own output back from a pipe if it were opened read/write.

There are four legal combinations of **O_RDONLY**, **O_WRONLY** and the **O_NONBLOCK** flag. We'll consider each in turn.

b. What are Message Queues and Semphores? Explain the terms with the help of Example. (8)

Answer:

In computing, **inter-process communication** (**IPC**) is a set of methods for the exchange of data among multiple threads in one or more processes. Processes may be running on one or more computers connected by a network. IPC methods are divided into methods for message passing, synchronization, shared memory, and remote procedure calls (RPC). The method of IPC used may vary based on the bandwidth and latency of communication between the threads, and the type of data being communicated.

There are several reasons for providing an environment that allows process cooperation:

- Information sharing
- Computational speedup
- Modularity
- Convenience
- Privilege separation

IPC may also be referred to as *inter-thread communication* and *inter-application communication.*

A semaphore is a special varible that takes only whole positive numbers and upon which only two operations are allowed: wait and signal. They are used to ensure that a single executing process has exclusive access to a resource.

Here are the signal notations:

- `P(semaphore variable)` for wait,
- `V(semaphore variable)` for signal.

Semaphore Definition

A **binary semaphore** is a variable that can take only the values 0 and 1.

Semaphores are used to control access to shared resources. Idea for semaphores due to Dijsktra. He introduced the concept semaphores. These are global variables used for access control. He further introduced the P and V operators. The P and V operators are defined (not implemented) as follows:

```
void P(sem)
{
        while ( sem < 1 ) ;
        sem--;
}
void V(sem)
{
        sem++;
}
```

Do example on how to protect a critical area of code.

Discuss why the above implementation will not work. Semaphores usually live in the operating system or have special machine language instructions to guarantee that they really work.

IPC semaphore are a more general and more confusing implementation of semaphores. Care has to be taken with semaphore in that there is some risk associated with them. There are alternatives to semaphores. These are referred to as mutexes. They are simpler and much more efficient when working with threads. They will be discussed in the advanced course.

The following are the system calls for semaphores.

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

int semget( key_t key, int nsem, int semflg );
```

Returns: the semaphore ID for the set of semaphores. -1 if error.

nsem - the number of semaphores requested. Most other operating systems only allow you to request one semaphore at a time.

Semflg - specifies options: IPC_CREAT, IPC_PRIVATE, IPC_EXCL, SEM_R, SEM_A. The "A" stands for alter.

Message queues provide a way of sending a block of data from one process to another.

Message queues provide an asynchronous communications protocol, meaning that the sender and receiver of the message do not need to interact with the message queue at the same time. Messages placed onto the queue are stored until the recipient retrieves them. Message queues have implicit or explicit limits on the size of data that may be transmitted in a single message and the number of messages that may remain outstanding on the queue.

System calls to support messages queues: msgget, msgsend, msgrcv, and msgctl. Some of the names should sound familiar from the discussion we had of shared memory.

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

int msgget ( key_t key,  int msgflg )
```

TEXT BOOK

I. Advanced Programming in the UNIX Environment, W. Richards Stevens, Pearson Education, 2004