Fifth Semester B.E. Degree Examination, Jan./Feb. 2021
Unix Programming

CBCS SCHEME

USN 1 C R 1 8 C S 1 8 3 **18CS56**

Fifth Semester B.E. Degree Examination, Jan./Feb. 2021

UNIX Programming

Time: 3 hrs. Max. Marks: 100

Note: Answer any FIVE full questions, choosing ONE full question from each module.

Module-1

1 a. Explain with a neat diagram a architecture of UNIX operating system. **(10 Marks)**

b. List and explain the silent features of UNIX operating system. 20 **(10 Marks)**

OR

2 a. What is a parent child relationship? With the help of neat diagram, explain UNIX file system. **(06 Marks)**

b. Explain any five file related commands with an example. **(10 Marks)**

c. With suitable example, bring out the differences between absolute and relative pathnames. **(04 Marks)**

Module-2

3 a. Which command is used for listing of file attributes? Explain the significance of each field. **(08 Marks)**

b. File current permissions are rw_r_xr__ specify chmod expression required to change for the following using both relative and absolute methods:

(i) rwxrwxrwx (ii) r__r_____ (iii) _________
(iv) ___r__r__ (v) _____x_w_ **(10 Marks)**

c. What is a shell? Briefly give the shell interpretive cycle. **(02 Marks)**

OR

4 a. With the help of an example, explain grep command with all the options. **(10 Marks)**

b. Explain three standard files supported by UNIX. **(06 Marks)**

c. What is the output for the following:

(i) ls [ijk]*doc (ii) [A – Z] ????* (iii) *.[!s][!h] (iv) *[!0 – 9] **(04 Marks)**

Module-3

5 a. Describe general UNIX file API's with syntax and explain each field in detail. **(10 Marks)**

b. Explain with a neat diagram memory layout of a C program and briefly discuss the different functions used for memory allocation. **(10 Marks)**

OR

6 a. Explain the UNIX Kernal support for process considering parent – child process show the related data structures. **(10 Marks)**

b. Bring out the differences between fork and vfork functions. **(05 Marks)**

c. Explain getrlimit and setrlimit function with prototype. **(05 Marks)**

Important Note : 1. On completing your answers, compulsorily draw diagonal cross lines on the remaining blank pages.
2. Any revealing of identification, appeal to evaluator and /or equations written eg, 42+8 = 50, will be treated as malpractice.

Module-4

7 a. Explain setuid and setgid functions with example and explain various ways to change user ids. **(06 Marks)**

b. What are pipes? What are its limitations? Write a program to send data from parent to child over a pipe. **(08 Marks)**

c. What are Interpreter Files? Give the difference between interpreter files and interpreter. **(06 Marks)**

OR

8 a. What is a FIFO? With a neat diagram, explain client server communication using FIFO. **(08 Marks)**

b. What are stream pipe? What are the different ways to view stream pipes? **(04 Marks)**

c. Explain briefly with example: (i) message queue (ii) semaphores **(08 Marks)**

Module-5

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

b. What are Daemon process? Enlist their characteristics. Also write a program to transform a normal user process into a Daemon process. **(10 Marks)**

OR

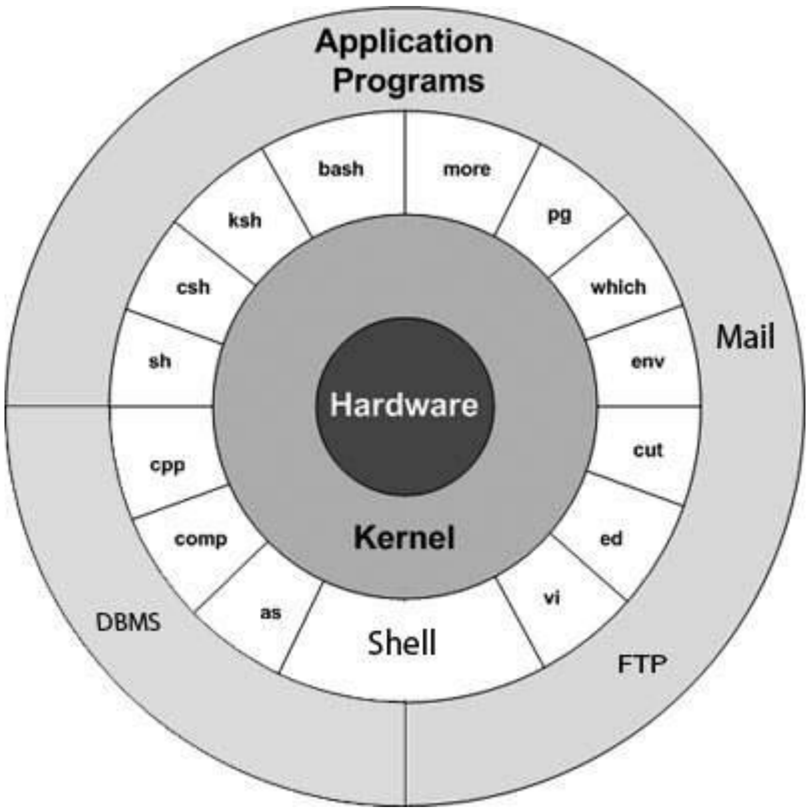
10 a. Explain the kill( ) API and alaram( ) API. **(10 Marks)**

b. Explain the Sigsetjmp and Siglongjmp functions with an example. **(10 Marks)**

* * * * *

Module 1

1a. Explain with a neat diagram an architecture of UNIX Operating System.



Kernel: is the core of the operating system. A collection of routines mostly written inC. It is loaded into memory when the system is booted and communicates directly with the hardware. User programs that need to access the hardware use the services of the kernel which performs the job on the user's behalf. These programs access the kernel through a set of functions called system calls. The kernel manages system memory, processes, decides priorities.

Shell: interface between Kernel and User. It functions as command interpreter i,e it receives and interprets the command from the user and interacts with the hardware. There is only one kernel running on the system, there could be several shells in action - one for each user who is logged in. When a user enters a command, the shell thoroughly examines the keyboard input and simplifies to a command line, and communicates with the kernel to see that the command is executed.

Eg. $echo Sun Solaris Sun Solaris //ignores all spaces in the above command line.

Files and Process: File is an array of bytes and it contains virtually anything. Unix considers even the directories and devices as members of the file system. The dominant file type is text and the behavior of the system is mainly controlled by text files. The second entity is the process, which is the name given to a file when it is executed as a program. Process is simply a "time image" of an executable file.

System Calls: Though there are thousands of commands in the Unix system, they all use a handful of functions called system calls. User programs that need to access the hardware use the services of the kernel, which performs the job on user's behalf. These programs access the kernel through a set of functions called system calls. Ex: open()-- system call to access both file and device. Write()—system call to write a file.

1.b List and Explain the silent features of UNIX OS

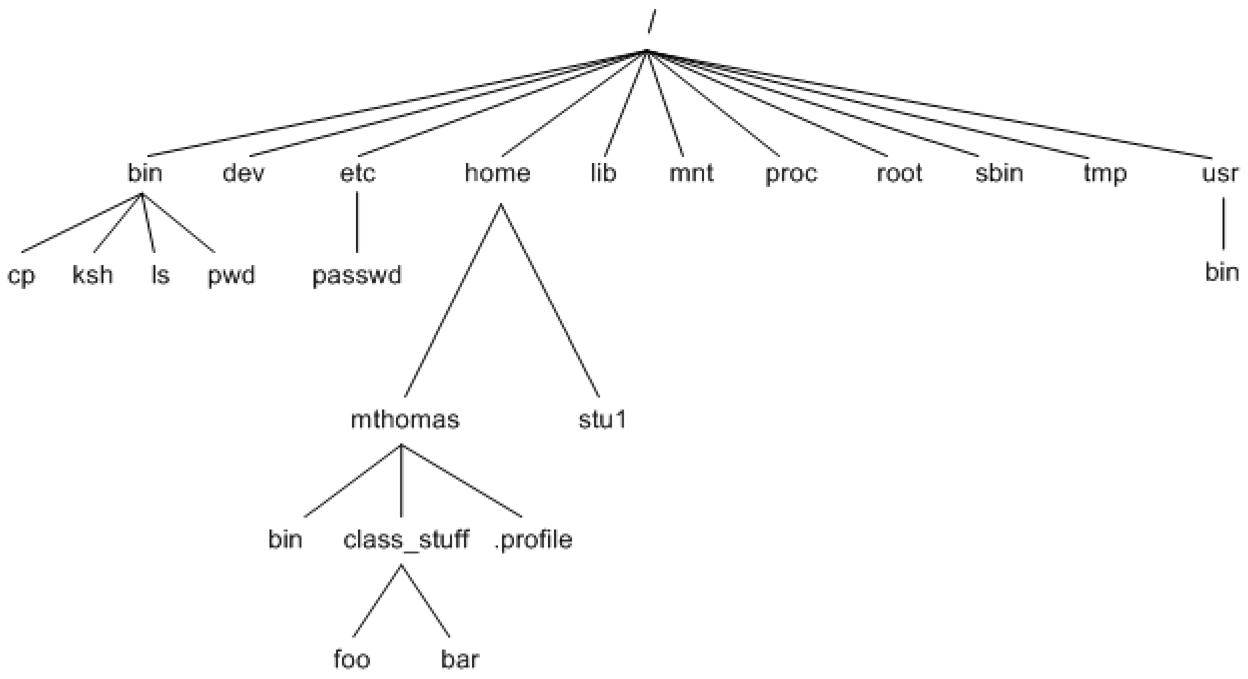
Several features of UNIX have made it popular. Some of them are:

1. **Portable:** UNIX can be installed on many hardware platforms. Its widespread use can be traced to the decision to develop it using the C language. Because C programs are easily

moved from one hardware environment to another, it is relatively simple to port it to different environments.

2. **A Multiuser & Multitasking system:** It is a multiprogramming system. It permits multiple programs to run & compete for the attention of the CPU. This can happen in two ways: Multiuser system: Multiple users can run separate jobs. In unix, the resources are actually shared between all users. The computer breaks up a unit and terminates them. Time expires, the previous job is kept in waiting, & the next user's job is.
3. **Multitasking system:** A single user can run multiple jobs. It's usual for a user to edit a file, print another file, send email etc. without leaving any applications. The kernel is designed to handle a user's multiple needs. In this environment, a user sees one job running in the foreground, & the rest running at the background. We can switch jobs between background & foreground, suspend.
4. The **Building-block approach:** Unix comes with hundreds of simple commands which perform one simple job. It's through pipes & filters unix implement small-is-beautiful philosophy. Using this feature, the output of one tool can be used as input of another tool. Ex: $ls | wc. Here the output of ls command is given as input to another command wc.
5. **Unix toolkit:** Unix supplied with a set of applications such as compilers, interpreters, text manipulation utilities, general purpose tools, & system administration tools. This set is constantly changing with every Unix release. The shell & utilities form part of POSIX specification. We must be able to download these tools & configure them to run on our machine.
6. **Pattern Matching:** Unix has very strong pattern matching features. The * (known as metacharacter) is a special character used by the system to indicate that it can match a number of filenames. Ex: $ls *.c There are many more such special symbols in the unix system. Some of the most advanced and useful tools use a special expression called regular expression that is framed with characters from this set.
7. **Programming facility:** The Unix shell is a programming language. It has all the necessary features of a language like control structures, loops & variables. This makes it a powerful programming language. These features are used to design shell scripts, the programs that can also invoke the Unix commands. Documentation: The principal online help facility is available is the man command in Unix, which remains the most important reference for commands and their configuration files

2.a What is parent child relationship? With the help of neat diagram, explain UNIX file system



A file can be informally defined as a collection of (typically related) data, which can be logically viewed as a stream of bytes (i.e. characters). A file is the smallest unit of storage in the Unix file system.a **file system** consists of files, relationships to other files, as well as the attributes of each file.the Unix file system is essentially composed of files and **directories**. Directories are special files that may contain other files.

The Unix file system has a hierarchical (or tree-like) structure with its highest level directory called root (denoted by /, pronounced *slash*). Immediately below the root level directory are several subdirectories, most of which contain system files. Below this can exist system files, application files, and/or user data files. Similar to the concept of the process parent-child relationship, all files on a Unix system are related to one another. That is, files also have a parent-child existence. Thus, all files (except one) share a common parental link, the top-most file (i.e. /) being the exception.

Above is a diagram (slice) of a "typical" Unix file system. As you can see, the top-most directory is / (slash), with the directories directly beneath being system directories. Note that as Unix implementations and vendors vary, so will this file system hierarchy. However, the organization of most file systems is similar.

2 b. Explain any five file related commands with example

mkdir:-

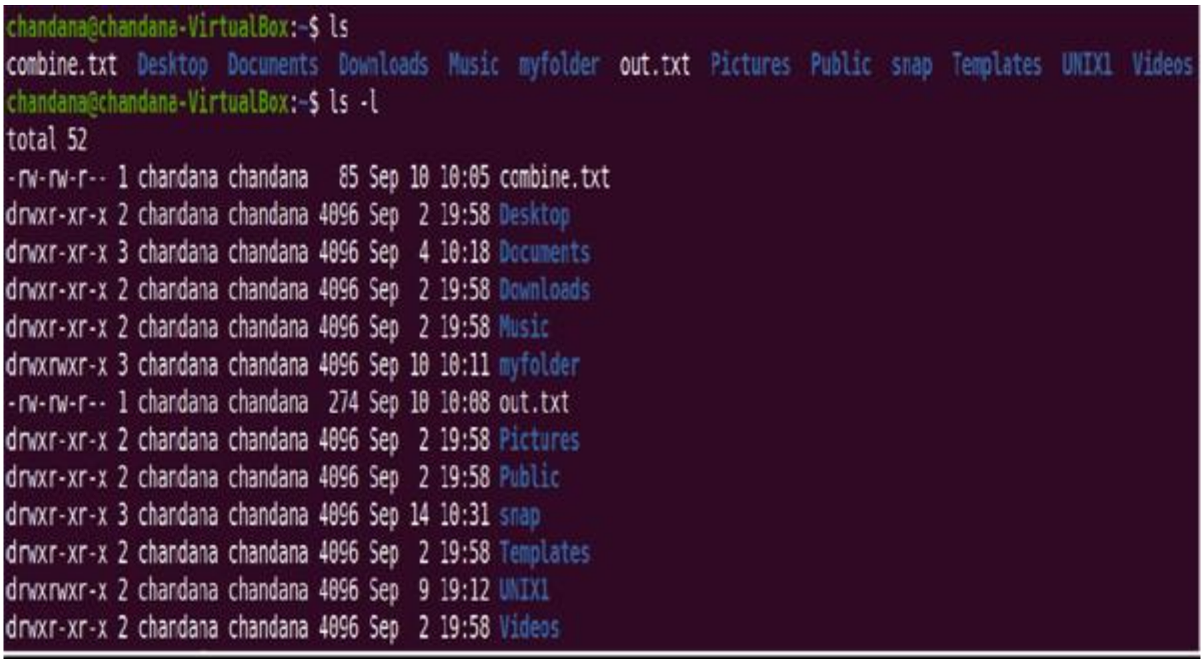
: "making directory". mkdir is used to create directories on a file system. If the specified directory does not already exist, mkdir creates it. More than one directory may be specified when calling mkdir.
mkdir syntax
mkdir [OPTION ...] DIRECTORY ...
Ex 1: To create a directory named cmrit, issue the following command.
$mkdir cmrit cmrit directory is created under present working directory. Assume that pwd is /home/csr, then cmrit directory is created under csr directory. Ex 2: To create three directories at a time, named ise cse mca pass directory names as arguments. $mkdir ise cse mca Ex 3: To create a directory tree: To create a directory named Faculty and create two subdirectories named Teaching and NonTeaching under Faculty, issue the command. Faculty is a parent directory. $mkdir parent directory sub-directories $mkdir Faculty Faculty/Teaching Faculty/NonTeaching Ex 4: Error while creating a directory tree $mkdir Faculty/Teaching Faculty/NonTeaching mkdir: Failed to make a directory “Faculty/Teaching”; no such file or directory mkdir: Failed to make a directory “Faculty/NonTeaching”; no such file or directory Error is due to the fact that the parent directory named Faculty is not created before creating sub directories Teaching and NonTeaching.

rmdir:-
The rmdir command removes the directory entry specified by each directory argument, provided the directory is empty.
Ex 1: $rmdir rnsit removes the directory named rnsit Arguments are processed in the order given. To remove both a parent directory and a subdirectory of that parent, the subdirectory must be specified first, so the parent directory is empty when rmdir tries to remove it. The reverse logic of mkdir is applied.
$rmdir subdirectories parent directory $rmdir Faculty/NonTeaching Faculty/Teaching Faculty You can't delete a directory with rmdir unless it is empty. In this example Faculty directory cannot be removed until the sub directories Faculty/NonTeaching and Faculty/Teaching are removed. You can't remove a sub directory unless you are placed in a directory which is hierarchically above the one you have chosen to remove.

cat:-
Create, view, concatenate files cat command is used to display the contents of a small file on the terminal.

- $ cat cprogram.c #include void main() { printf("hello"); } As like other files cat accepts more than one filename as arguments $ cat a.txt b.txt It contains the contents of a.txt It contains the contents of b.txt In this the contents of the second files are shown immediately after the first file without any header information. So, cat concatenates two files - hence its name. cat Options To view contents of a file preceding with line numbers -n is the numbering option helps programmers in debugging programs. To create a file
- $cat >newfile This is a new file which contains some text, just to add some contents to the file new [ctrl-d] $_ When the command line is terminated with [Enter], the prompt vanishes. Cat now waits to take input from the user. Enter a few lines; press [ctrl-d] to signify the end of input to the system To display the file contents of new use file name with cat command.
- $ cat new This is a new file which contains some text, just to Add some contents to the file new. is the command and can input the content of newfile also. To copy the contents of one file to another file
- $cat [filename whose contents is to be copied] > [destination filename] is the command. Cat command can append the contents of one file to the end of another file by using the command
- $cat file1 >> file2. Cat command can display content in reverse order using tac command.
- $tac filename Cat command can highlight the end of line
- $cat -E "filename" Cat command to merge the contents of multiple file $cat "file1" "file2" "file3" > "newfile"

- **ls:-**
- **Listing Files** The files are organized in separate folders called directories. We can list the names of the files available in this directory with the ls command. Following is the ls command without any option:
- $ ls Chap01 Chap02 Pgm.c F1.doc Unix has a special symbol * to access the files with same pattern $ ls Chap* Chap01 Chap02



```
chandana@chandana-VirtualBox:~$ ls
combine.txt  Desktop  Documents  Downloads  Music  myfolder  out.txt  Pictures  Public  snap  Templates  UNIX1  Videos
chandana@chandana-VirtualBox:~$ ls -l
total 52
-rw-rw-r-- 1 chardana chandana   85 Sep 10 10:05 combine.txt
drwxr-xr-x 2 chardana chandana 4096 Sep  2 19:58 Desktop
drwxr-xr-x 3 chardana chandana 4096 Sep  4 10:18 Documents
drwxr-xr-x 2 chardana chandana 4096 Sep  2 19:58 Downloads
drwxr-xr-x 2 chardana chandana 4096 Sep  2 19:58 Music
drwxrwxr-x 3 chardana chandana 4096 Sep 10 10:11 myfolder
-rw-rw-r-- 1 chardana chandana  274 Sep 10 10:08 out.txt
drwxr-xr-x 2 chardana chandana 4096 Sep  2 19:58 Pictures
drwxr-xr-x 2 chardana chandana 4096 Sep  2 19:58 Public
drwxr-xr-x 3 chardana chandana 4096 Sep 14 10:31 snap
drwxr-xr-x 2 chardana chandana 4096 Sep  2 19:58 Templates
drwxrwxr-x 2 chardana chandana 4096 Sep  9 19:12 UNIX1
drwxr-xr-x 2 chardana chandana 4096 Sep  2 19:58 Videos
```

- $ls – lists directory contents of files and directories.
- $ls -l – Display all information about files/directories contents.
- Here is the information about all the listed columns:
- 1. First Column: represents file type and permission given on the file. Above is the description of all type of files.
- 2. Second Column: represents the number of memory blocks taken by the file or directory.
- 3. Third Column: represents owner of the file. This is the Unix user who created this file. 4. Fourth Column: represents group of the owner. Every Unix user would have an associated group.
- 5. Fifth Column: represents file size in bytes.
- 6. Sixth Column: represents date and time when this file was created.
- 7. Seventh Column: represents filename. In the ls -l listing example, every file line began with a d, -, or l. These characters indicate the type of file that's listed.

- **echo:-**

  The echo command used to display a message. To display the diagnostic messages on the terminal or to issue prompts for taking user input (like echo Sun Solaris).
  To evaluate shell variables (like echo $SHELL)
  General Syntax: $echo [Short-Option]... [String]... Originally, echo was an external command, but nowadays all shells have echo built-in. Most of the echo's behavior differences relates to the way echo's interprets certain strings known as escape sequences. An escape sequence is generally a two character – string beginning with \(backslash).
  Eg:- echo $SHELL
  /bin/bash
  echo " hai gm"
  Hai gm

2 c. With suitable example, bring out the difference between absolute and relative pathnames.

1. **Relative pathname:** Pathnames that don't begin with / specify locations relative to the current working directory. It uses either the current or parent directory as reference and specifies path relative to it.

   A relative pathname uses one of these cryptic symbols. cd progs cat login.sql Here both are presumed to exist in the current directory.

   Now, if progs contain a directory script under it, the user won't need an absolute pathname to change to that directory. Just users can use **cd progs/scripts.**

Here we have a pathname that has a /, but it is not absolute because it doesn't begin with a /.

2. **Absolute:** If the first character of a pathname is / the files location must be determined with respect to root(/) .

Such a pathname is called absolute pathname. cat /home/chandana When users have more than one / in a pathname for such / have to descend one level in the file system.

Thus chandana is one level below home and two levels below root. When a user specifies a file by using front slashes to demarcate the various levels, have a mechanism of identifying a file uniquely. No two files in a UNIX system can have identical absolute pathnames.

Users can have two files with the same name, but in different directories, their pathnames will also be different. Thus, the file /home/chandana/progs/p1.c can coexist with the file /home/chandana/scripts/p1.c. When a user specifies the date command, the system has to locate the file date from a list of directories specified in the PATH variable and then execute it. However if the user knows the location of a command in prior, for example date is usually located in /bin or /usr/bin .

Use absolute pathname i,e precede its name with complete path $/bin/date. For example if you need to execute program less residing in /usr/local/bin you need to enter the absolute pathname **$/usr/local/bin/less**

Module 2

3a. Which command is used for listing file attributes? Explain the siginifance of each field.

The command to list the file attributes is ls -l. The command lists seven attributes of the files in the current directory. The list is preceeded by a string 'total 72' indicating the number of blocks occupied by these files on the disk. The following are the seven attributes:

1) File Type and permissions: These attributes are represented using seven characters. The first character identified whether the file is ordinary or directory type. The next six characters represent the read, write and execute permissions for the file with respect to owner, groups and others.

2) Links: Number of hards associated with the file, which is the number of filenames associated with the file.

3) Ownership: The thirs attribute is the owner of the file, or in otherwords the user who created the file.

4) Group Ownership: When a file is created the administrator assigns the file toa group. The fourth attribute specifies the owner of the group.

5) File size: The number of bytes contained in the file.

6) Last modification time: The last modification time indicates the time at which the contents of the file have changed. It is indicated to the nearest second.

7) Filename

3b. File current permissions are rw_r_xr__. Specify chmod expression required to change for the following using both relative and absolute methods.

Relative Method

i) rwxrwxrwx: chmod u+x, g+w, o+wx <filename>
ii) r--r-----: chmod u-w, r-x, o-r <filename>
iii) ---------: chmod u-rw, g-rx, o-r <filename>
iv) ---r--r--: chmod u-rw,g-x <filename>
v) -----x-w-: chmod u-rw,g-r,o-r,o+w <filename>

Absolute Method

i) rwxrwxrwx: chmod 777 <filename>
ii) r--r-----: chmod 440 <filename>
iii) ---------: chmod 000 <filename>
iv) ---r--r--: chmod 044 <filename>
v) -----x-w-: chmod 012 <filename>

3c. What is Shell? Briefly give the shell interpretive cycle.

A shell is special user program which provide an interface to user to use Unix operating system services. Shell accept commands from user and convert them into a form understood by the kernel. It is a command interpreter that execute commands read from input devices such as keyboards or from files. The shell gets started when the user logs in or start the terminal.

The activities performed by the shell are summarized as shell's interpretive cycle:

1) The shell issues the prompt and waits for user to enter a command.
2) After the command is entered, the shell scans for meta characters and expands abbreviations.
3) It then passes on the command line to kernel for execution.
4) The shell waits for command to complete and normally cannot do any work whie command is running.
5) After command execution is complete, the prompt reappears and the shell waits to its waiting role, to start a new cycle.

4a. With the help of an example, explain grep command with all the options.

The command grep scans the input for a pattern, and displays the lines containing the pattern, the line numbers or filenames where the pattern occurs. The syntax of the command is as follows:

grep options pattern filename(s)

If no filename is specified, grep takes input from standard input device. The output of grep is sent to standard output stream. The following are the grep options:

(i) Ignore case:
grep -i 'agarwal' emp.list
The command will display 'agarwal' as well as 'Agarwal' if they are present in the file emp.list.

(ii) Inverse case:
grep -v ‘agarwal’ emp.list
The command will display all lines not containing the pattern ‘agarwal’ present in the file emp.list.

(iii) Displaying line numbers:
grep -n ‘agarwal’ emp.list
The command will display the line numbers of all lines containing the pattern ‘agarwal’ present in the file emp.list.

(iv) Counting the lines containing patterns
grep -c ‘agarwal’ emp.list
The command will display the count of the lines containing the pattern ‘agarwal’ present in the file emp.list.

(vi) Displaying filenames
grep -l ‘agarwal’ *.list
The command will display the filenames containing the pattern ‘agarwal’ present in the files *.list.

(vii) Taking patterns from a file
grep -f pattern.list emp.list
The file pattern.list can contain the patterns, one in each line, to be searched in the file emp.list.
4b. Explain three standard files supported by Unix.

When user logs in, the shell makes available three files representing three streams – standard input, standard output and standard error. These are streams of characters, which many commands see as input and output.

Standard Input: The file represents three input sources:

1) The keyboard, the default source.
2) A file using redirection using the symbol <
3) Another program using a pipeline

File descriptor of 0 is associated with standard input file.

Standard Output: The file represents three output devices:

1) The terminal, the default destination.
2) A file using the redirection symbol > and >>.
3) As input to another program using a pipeline.

File descriptor of 1 is associated with standard input file.

Standard Error: When a user enters a an incorrect command or tries to open an non-existent file, diagnostic messages appear on the screen. This is the standard error stream, whos default destination is the terminal. These messages can be redirected to afile using the following command:

cat <non-existent file> 2> errorfile.

4c. What is the output of the following;

(i) ls [ijk]*doc
Lists all files starting with i or j or k and ending with doc.

(ii) [A-Z]????*
Matches all strings starting with a captial alphabet followed by 4 or more characters.

(iii) *.[!s][!h]
Matches all strings not ending in '.sh'.

(iv) *[!0-9]
Matches all strings not ending in a digit.

Module 3

5 a. Describe general UNIX file API's with syntax and explain each field in details.

- In UNIX everything can be treated as files. Hence files are the building blocks of the UNIX operating system. When you execute a command in UNIX, the UNIX kernel fetches the corresponding executable file from a file system, loads its instruction text to memory and creates a process to execute the command.
- UNIX / POSIX file Types The different types of files available in UNIX / POSIX are: **Regular files** Example: All .exe files, C, C++, PDF Document files.
    - **Directory files** Example: Folders in Windows.
    - Device files Example: Floppy, CD ROM and Printer.
    - FIFO files Example: Pipes. Link Files (only in UNIX) Example: alias names of a file, Shortcuts in Windows.

Creat:

1. The creat system call is used to create new regular files.
2. **Prototype**:
   #include < sys/types.h>
   #include<unistd.h> int creat (const char *path_name, mode_t mode);

- **The path_name** argument is the path name of a file to be created.
- **The mode argument** is the same as that for open API.
- Since the O_CREAT flag was added to the open API it was used to both create and open regular files.
- So, the creat API has become obsolete.
- It is retained for backward-compatibility with early versions of UNIX.
- The creat function can be implemented using the open function as:
- #define creat (path_name, mode) open(path_name, O_WRONLY|O_CREAT|O_TRUNC, mode)

read:

- This function fetches a fixed size block of data from a file referenced by a given file descriptor.
- **Prototype**: **#include < sys/types.h>**
- **#include<unistd.h> ssize_t read (int fdesc ,void* buf, size_t size);**
- **fdesc**: is an integer file descriptor that refers to an opened file.
  buf: is the address of a buffer holding any data read.
  size: specifies how many bytes of data are to be read from the file.
    a. **Note: read function can read text or binary files. This is why the data type of buf is a universal pointer (void *).
    b. For example, the following code reads, sequentially one or more record of struct sample-typed data from a file called dbase:

```
 struct sample { int x; double y; char* a;} varX;
int fd = open("dbase", O_RDONLY);
while ( read(fd, &varX, sizeof(varX))>0)
```

- The return value of read is the number of bytes of data successfully read and stored in the buf argument.
- It should be equal to the size value.
- If a file contains less than size bytes of data remaining to be read, the return value of read will be less than that of size. If end-of-file is reached, read will return a zero value. size_t is defined as int in header, users should not set size to exceed INT_MAX in any read function call.
- If a read function call is interrupted by a caught signal and the OS does not restart the system call automatically, POSIX.1 allows two possible behaviors: 1. The read function will return -1 value, errno will be set to EINTR, and all the data will be discarded.

1. The read function will return the number of bytes of data read prior to the signal interruption. This allows a process to continue reading the file. The read function may block a calling process execution if it is reading a FIFO or device file and data is not yet available to satisfy the read request. Users may specify the O_NONBLOCK or O_NDELAY flags on a file descriptor to request non blocking read operations on the corresponding file.

5.b Explain with a neat diagram memory layout of a C program and briefly discuss the different functions used for memo

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

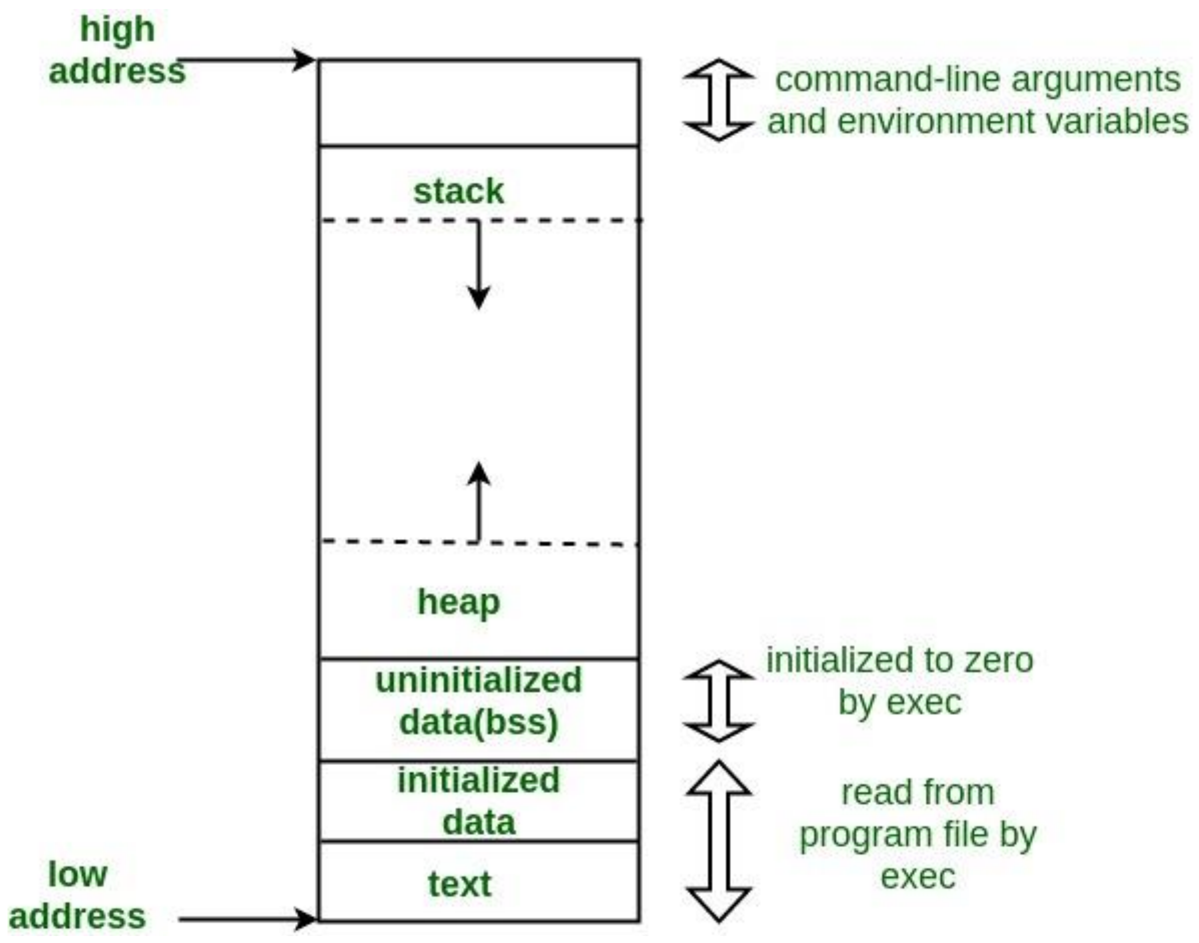
Text segment, the machine instructions that the CPU executes. Usually, the text segment is sharable so that only a single copy needs to be in memory for frequently executed programs, such as text editors, the C compiler, the shells, and so on. Also, the text segment is often read-only, to prevent a program from accidentally modifying its instructions.

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

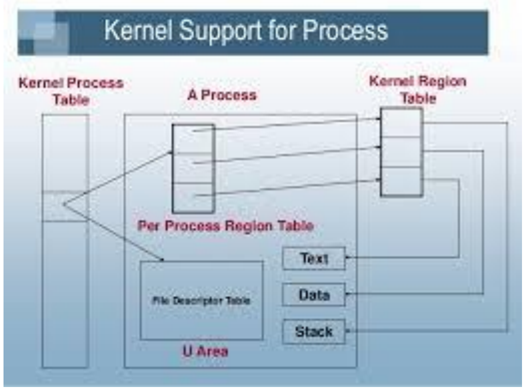
Uninitialized data segment, often called the "bss" segment, named after an ancient assembler operator that stood for "block started by symbol." Data in this segment is initialized by the kernel to arithmetic 0 or null pointers before the program starts executing. The C declaration long sum[1000]; appearing outside any function causes this variable to be stored in the uninitialized data segment.

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

Heap, where dynamic memory allocation usually takes place. Historically, the heap has been located between the uninitialized data and the stack. The figure below shows the typical arrangement of these segments.


high address
command-line arguments and environment variables
stack
heap
uninitialized data(bss)
initialized to zero by exec
initialized data
read from program file by exec
text
low address

6 a. Explain the UNIX kernel support for process considering parent-child process show the related data structure.



1. The data structure and execution of processes are dependent on operating system implementation. As shown in the figure below, a UNIX process consists minimally of a text segment, data segment, and a stack segment.
2. A segment is an area of memory that is managed by the system as a unit. A text segment contains the program text of a process in machine—executable instruction code format. A data segment contains static and global variables and their corresponding data. A stack segment contains a run-time stack. A stack provides storage for function arguments, automatic variables, and return address of all active functions for a process at any time
3. 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**

4. **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.**
5. All processes in the UNIX system, except the first process (process 0) which is created by the system boot code, are created via the fork system call. After the fork system call both the parent and child processes resume execution at the return of the fork function.
6. 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.**
7. Furthermore, 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.

6 b. Bring out the difference between fork and vfork functions.

i)fork:-

An existing process can create a new one by calling the fork function. The prototype for the fork function is:

#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. 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. Current implementations don't perform a complete copy of the parent's data, stack, and heap, since a fork is often followed by an exec. Instead, a technique called copy-on-write (COW) is used. These regions are shared by the parent and the child and have their protection changed by the kernel to read-only. If either process tries to modify these regions, the kernel then makes a copy of that piece of memory only, typically a "page" in a virtual memory system.

Eg:-

```
#include<stdio.h>

int main()

{

        int a=10,pid;

        if((pid=fork())<0)

        {

                printf(“error”);

                return -1;

        }

        else

        {

                a=a+1;

                printf(“child process a =%d\n”,a);

        }

        printf(“parent process a =%d\n”,a);

}
```

ii)vfork

The function vfork has the same calling sequence and same return values as fork. But the semantics of the two functions differ. The vfork function originated with 2.9BSD. It 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. (This can lead to deadlock if the child depends on further actions of the parent before calling either of these two functions.)

```
Eg:-
#include<stdio.h>
int main()
{
        int a=10,pid;
        if((pid=vfork())<0)
        {
                printf("error");
                return -1;
        }
        else
        {
                a=a+1;
                printf("child process a =%d\n",a);
        }
        printf("parent process a =%d\n",a);
}
```

6 c. Explain getlimit and set limit function with prototype.
The getrlimit() and setrlimit() system calls can be used to get and set the resource limits such as files, CPU, memory etc. associated with a process.
Each resource has an associated soft and hard limit.
soft limit: The soft limit is the actual limit enforced by the kernel for the corresponding resource.
hard limit: The hard limit acts as a ceiling for the soft limit.
The soft limit ranges in between 0 and hard limit.

```
struct rlimit {
        rlim_t rlim_cur; /* Soft limit */
        rlim_t rlim_max; /* Hard limit (ceiling for rlim_cur) */
};
int getrlimit(int resource, struct rlimit *rlim);
int setrlimit(int resource, const struct rlimit *rlim);
```

Module 4

7a. Explain setuid and setgid functions with example and explain various ways to change user ids.

User ID is a unique neumeric value assigned by system administrator when a user's login name is assigned. It is used for User Identification and is stored in the password file. User ID of 0 is assigned to the superuser named root.

Users are collected into groups who work together on a project or work in a department. This allows sharing of resources among the members of the same group. Groups are identified by Group Ids. This is assigned by a the system administrator when a login name is assigned. Entry in the password file also specifies a user's group ID.

In Unix, previlages for a user, is based on user and group Ids. When a user's program needed additional previlages, then it is necessary to change the User ID and group ID to an ID which can provide the necessary previlage. In general, applications are designed as a aleast previlage model.

When our programs need additional privileges or need to gain access to resources that they currently aren't allowed to access, they need to change their user or group ID to an ID that has the appropriate privilege or access. Similarly, when our programs need to lower their privileges or prevent access to certain resources, they do so by changing either their user ID or group ID to an ID without the privilege or ability access to the resource.

There are three types of user Ids:

1) Real User Ids: It is account of owner of this process. It defines the access rights to files.
2) Effective User Id: It is normally same as Real UserID, but sometimes it is changed to enable a non-privileged user to access files that can only be accessed by root.
3) Saved Set User Id:

User Ids and Group Ids can be changed using setuid and and setgid functions. The prototype of the functions are as follows:

```
# include <unistd.h>

int setuid(uid_t uid);
int setgid(uid_t gid);
```

Both return 0 on success and -1 on error.

The rules for changing the Ids are as follows. Let's consider only the user ID for now. Everything we describe for the user ID also applies to the group ID.

1. If the process has superuser privileges, the `setuid` function sets the real user ID, effective user ID, and saved set-user-ID to uid.
2. If the process does not have superuser privileges, but uid equals either the real user ID or the saved set-user-ID, `setuid` sets only the effective user ID to uid. The real user ID and the saved set-user-ID are not changed.
3. If neither of these two conditions is true, error is returned.

7b. What are pipes ? What are its limitations? Write a program to send data from a parent to child over a pipe.

(a) Pipes are used for communicating between UNIX processes.

(b) Pipes have two limitations: (i) Pipes are half-duplex (ii) Pipes can be used to communicate only between two processes that have a common ancestor.

(c) Normally, a pipe is created by a process, that process calls fork, and pipe is used between the parent and the child. A pipe is created by calling the pipe() function. The prototype is as follows:

```
int pipe(int filedes[2]);
```

The function returns 0 on success and -1 on error.

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

        if (pipe(fd) < 0) printf("Error in creating pipe\n");
        if ((pid = fork()) < 0) printf("Error in creating process\n");
        else if (pid > 0)
        {
                close(fd[0]);
                write(fd[1],"hello world\n",12);
        }
        else
        {
                close(fd[1]);
                n=read(fd[0], line,MAXLINE);
                write(1, line, n);
        }
        exit(0);
}
```

7c. What are interpreter files ? Give the difference between interpreter files and interpreter.

Interpreter files are text files that begin with a line of the form
#! pathname [optional arguments]. The recognition of these files is done within the kernel as part of processing the exec system call. The actual file that gets executed by the kernel is the file specified by the pathname on the first line of the interpreter file.

Let's look at an example to see what the kernel does with the arguments to the `exec` function when the file being executed is an interpreter file (testinterp)and the optional argument on the first line of the interpreter file.

The following shows the contents of the one-line interpreter file that is executed and the result from running the program.

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

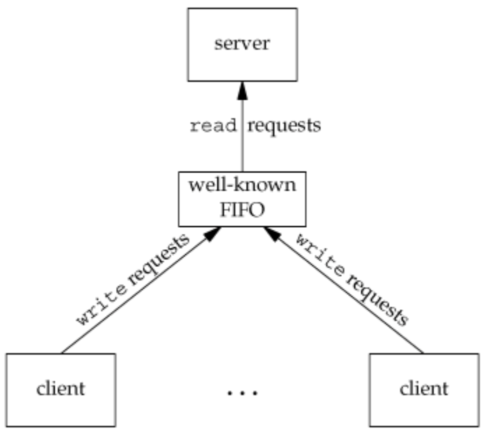
A program that execs an interpreter file

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

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

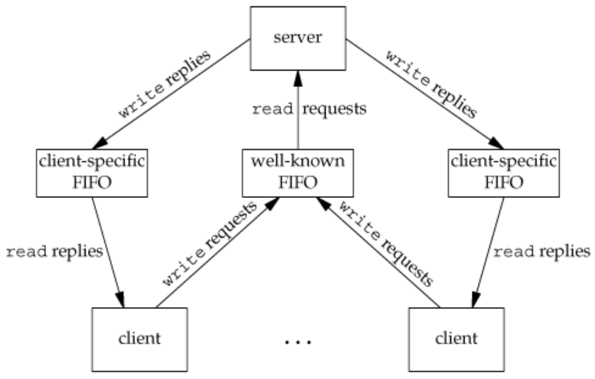
8a. What is a FIFO? With a neat diagram explain client server communiation using FIFO.

FIFOs another means of inter-process communication in Unix. They are also called named pipes. Pipes can be used only between related processes when a common ancestor has created the pipe. With FIFOs, however, unrelated processes can exchange data.

Another use for FIFOs is 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. The pathname of the FIFO must be known to all the clients that need to contact the server. 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 clientserver 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. The arrangement has the limitation that the server is unable to know whether the client has crashed.

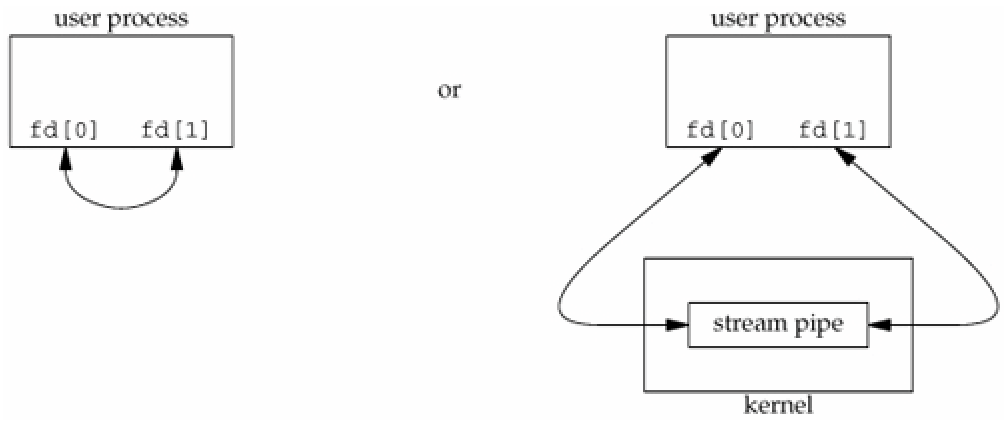


8b. What are stream pipes ? What are the different ways to view stream pipes ?

A STREAMS-based pipe ("STREAMS pipe," for short) is a bidirectional (full-duplex) pipe. To obtain bidirectional data flow between a parent and a child, only a single STREAMS pipe is required.

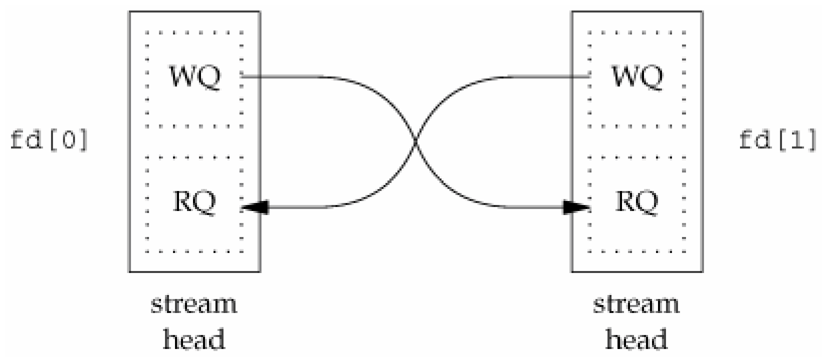
Figure 1 shows the two ways to view a STREAMS pipe. The only difference between this picture and that of pipes is that the arrows have heads on both ends; since the STREAMS pipe is full duplex, data can flow in both directions.

Figure 1. Two ways to view a STREAMS pipe



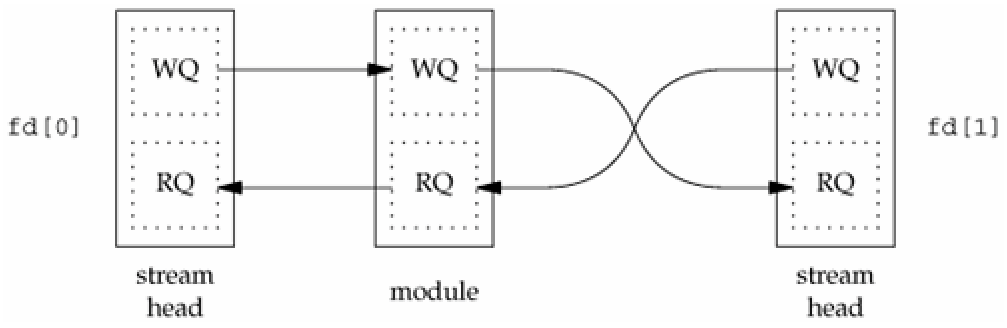
If we look inside a STREAMS pipe (Figure 2), we see that it is simply two stream heads, with each write queue (WQ) pointing at the other's read queue (RQ). Data written to one end of the pipe is placed in messages on the other's read queue.

Figure 2. Inside a STREAMS pipe



Since a STREAMS pipe is a stream, we can push a STREAMS module onto either end of the pipe to process data written to the pipe (Figure 3). But if we push a module on one end, we can't pop it off the other end. If we want to remove it, we need to remove it from the same end on which it was pushed.

Figure 17.3. Inside a STREAMS pipe with a module



Assuming that we don't do anything fancy, such as pushing modules, a STREAMS pipe behaves just like a non-STREAMS pipe, except that it supports most of the STREAMS `ioctl` commands.

Example

Let's redo the coprocess example, with a single STREAMS pipe. Figure 4 shows the new `main` function. The `add2` coprocess is the same.. We call a new function, `s_pipe`, to create a single STREAMS pipe.

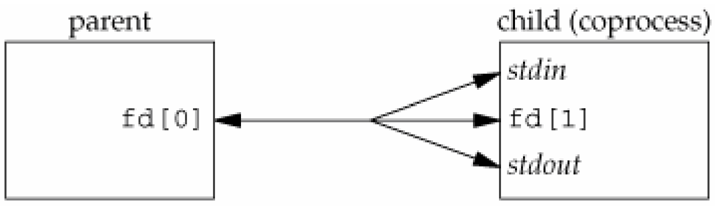
The parent uses only `fd[0]`, and the child uses only `fd[1]`. Since each end of the STREAMS pipe is full duplex, the parent reads and writes `fd[0]`, and the child duplicates `fd[1]` to both standard input and standard output. Figure 5 shows the resulting descriptors. Note that this example also works with full-duplex pipes that are not based on STREAMS, because it doesn't make use of any STREAMS features other than the full-duplex nature of STREAMS-based pipes.

Figure 4. Program to drive the add2 filter, using a STREAMS pipe

```
#include "apue.h"

static void sig_pipe(int);       /* our signal handler */
```

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

    if (signal(SIGPIPE, sig_pipe) == SIG_ERR)
        err_sys("signal error");

    if (s_pipe(fd) < 0)         /* need only a single stream pipe */
        err_sys("pipe error");
    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid > 0) {                           /* parent */
        close(fd[1]);
        while (fgets(line, MAXLINE, stdin) != NULL) {
            n = strlen(line);
            if (write(fd[0], line, n) != n)
                err_sys("write error to pipe");
            if ((n = read(fd[0], line, MAXLINE)) < 0)
                err_sys("read error from pipe");
            if (n == 0) {
                err_msg("child closed pipe");
                break;
            }
            line[n] = 0; /* null terminate */
            if (fputs(line, stdout) == EOF)
                err_sys("fputs error");
        }
        if (ferror(stdin))
            err_sys("fgets error on stdin");
        exit(0);
    } else {                                    /* child */
        close(fd[0]);
        if (fd[1] != STDIN_FILENO &&
          dup2(fd[1], STDIN_FILENO) != STDIN_FILENO)
            err_sys("dup2 error to stdin");
        if (fd[1] != STDOUT_FILENO &&
          dup2(fd[1], STDOUT_FILENO) != STDOUT_FILENO)
               err_sys("dup2 error to stdout");
        if (execl("./add2", "add2", (char *)0) < 0)
            err_sys("execl error");
    }
    exit(0);
}
static void
sig_pipe(int signo)
{
    printf("SIGPIPE caught\n");
    exit(1);
}
```

Figure 17.5. Arrangement of descriptors for coprocess



We define the function `s_pipe` to be similar to the standard `pipe` function. Both functions take the same argument, but the descriptors returned by `s_pipe` are open for reading and writing.

8c. Explain briefly with examples (i) Semaphores (ii) Message Queues.

a) Message Queues

A message queue is a linked list of messages stored within the kernel and identified by a message queue identifier. Every message queue is associated with a structure namely `msqid_ds`. The first function normally called is `msgget` to either open an existing queue or create a new queue. The prototype is as follows:

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

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

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

The `msgctl` function performs various operations on a queue. Its prototype is as follows:

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

Returns: 0 if OK, -1 on error

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

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

IPC_SET : Copy the fields from the structure pointed to by buf to the msqid_ds
structure associated with this queue.

IPC_RMID : Remove the message queue from the system and any data still on the
queue.

Data is placed onto a message queue by calling msgsnd. The prototype is as
follows:

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 byte.

Messages are retrieved from a queue by msgrcv. The prototype is as follows:

```
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 the user specify which message to retrive.

b) 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.

To implement semaphores correctly, the test of a semaphore's value and the decrementing of this value must be an atomic operation. For this reason, semaphores are normally implemented inside the kernel.

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.

In Unix, semaphore is implemented as a set of one or more semaphore values. When we create a semaphore, we specify the number of values in the set.

The first function to call is `semget` to obtain a semaphore ID. The prototype is as follows:

```
int semget(key_t key, int nsems, int flag);
```
Returns: semaphore ID if OK, -1 on error

nsems is the number of semaphores in the set.

Semctl helps to perform operations like get, set and remove semaphores.

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

The semaphores in the set are numbered from 0 to n-1. Semnum identifies the particular semaphore in the set.

The function `semop` atomically performs an array of operations on a semaphore set.

Incrementing or decrementing the counter value of a semaphore can be achieved using semop function whose prototype is given below:

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

Module 5

9 a. What are signals? Mention difference source of signals? Write a Program to setup signal handlers for SIGINIT and SIGALRM.

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.

The most common way of sending signals to processes is using the keyboard. There are certain key presses that are interpreted by the system as requests to send signals to the process with which we are interacting:

Ctrl-C

Pressing this key causes the system to send an INT signal (SIGINT) to the running process. By default, this signal causes the process to immediately terminate.

Ctrl-Z

Pressing this key causes the system to send a TSTP signal (SIGTSTP) to the running process. By default, this signal causes the process to suspend execution.

Ctrl-\

Pressing this key causes the system to send a ABRT signal (SIGABRT) to the running process. By default, this signal causes the process to immediately terminate. Note that this redundancy (i.e. Ctrl-\ doing the same as Ctrl-C) gives us some better flexibility. We'll explain that later on.

The alarm API can be called by a process to request the kernel to send the SIGALRM signal after a certain number of real clock seconds. The function prototype of the API is: Returns: 0 or number of seconds until previously set alarm The alarm API can be used to implement the sleep API:

```
#include<signal.h>
#include<stdio.h>
#include<unistd.h>
void wakeup( )
{ ; }
unsigned int sleep (unsigned int timer )
{
        Struct sigaction action;
        action.sa_handler=wakeup;
        action.sa_flags=0;
        sigemptyset(&action.sa_mask);
        if(sigaction(SIGALARM,&action,0)==-1)
        {
               perror("sigaction");
               return -1;
        }
        (void) alarm (timer);
        (void) pause( );
        return 0;
 }
```

9 b. What is daemon process? Enlist their characteristics. Also write a program to transform a normal user process into a Daemon Process.

A daemon process is a background process that is not under the direct control of the user. This process is usually started when the system is bootstrapped and it terminated with the system shut down.

Usually the parent process of the daemon process is the init process. This is because the init process usually adopts the daemon process after the parent process forks the daemon process and terminates.

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

int main (void)
{
  pid_t pid;
  int i;

  /* create new process */
  pid = fork ( );
  if (pid == -1)
    return -1;
  else if (pid != 0)
    exit (EXIT_SUCCESS);

  /* create new session and process group */
  if (setsid ( ) == -1)
    return -1;

  /* set the working directory to the root directory */
  if (chdir ("/") == -1)
    return -1;
  /* close all open files--NR_OPEN is overkill, but works */
  for (i = 0; i < NR_OPEN; i++)
     close (i);

  /* redirect fd's 0,1,2 to /dev/null */
  open ("/dev/null", O_RDWR);
  /* stdin */
  dup (0);
  /* stdout */
  dup (0);
  /* stderror */

  /* do its daemon thing... */

  return 0;
}
```

10 a. Explain Kill() API and alarm API()

A process can send a signal to a related process via the kill API. This is a simple means of interprocess communication or control. The function prototype of the API is: Returns: 0 on success, -1 on failure. The signal_num argument is the integer value of a signal to be sent to one or more processes designated by pid. The possible values of pid and its use by the kill API are: pid > 0 The signal is sent to the process whose process ID is pid. pid == 0 The signal is sent to all processes whose process group ID equals the process group ID of the sender and for which the sender has permission to send the signal. pid < 0 The signal is sent to all processes whose process group ID equals the absolute value of pid and for which the sender has permission to send the signal. pid == 1 The signal is sent to all processes on the system for which the sender has permission to send the signal.

The following program illustrates the implementation of the UNIX kill command using th
API:

```
#include<iostream.h>
#include<stdio.h>
#include<unistd.h>
#include<string.h>
#include<signal.h>
int main(int argc,char** argv)
{
       int pid, sig = SIGTERM;
       if(argc==3)
       {
              if(sscanf(argv[1],"%d",&sig)!=1)
              {
                     cerr<<"invalid number:" << argv[1] << endl; return -1;
              }

        argv++,argc--;
 }
 while(--argc>0)
 if(sscanf(*++argv, "%d", &pid)==1)
 {
     if(kill(pid,sig)==-1)
        perror("kill");
 }
 else
     cerr<<"invalid pid:" << argv[0] <<endl;
     return 0;
}
```

UNIX kill command invocation syntax is:

Kill [-<signal_num>] <pid>......

ere signal_num can be an integer number or the symbolic name of a signal. <pid> is process ID.

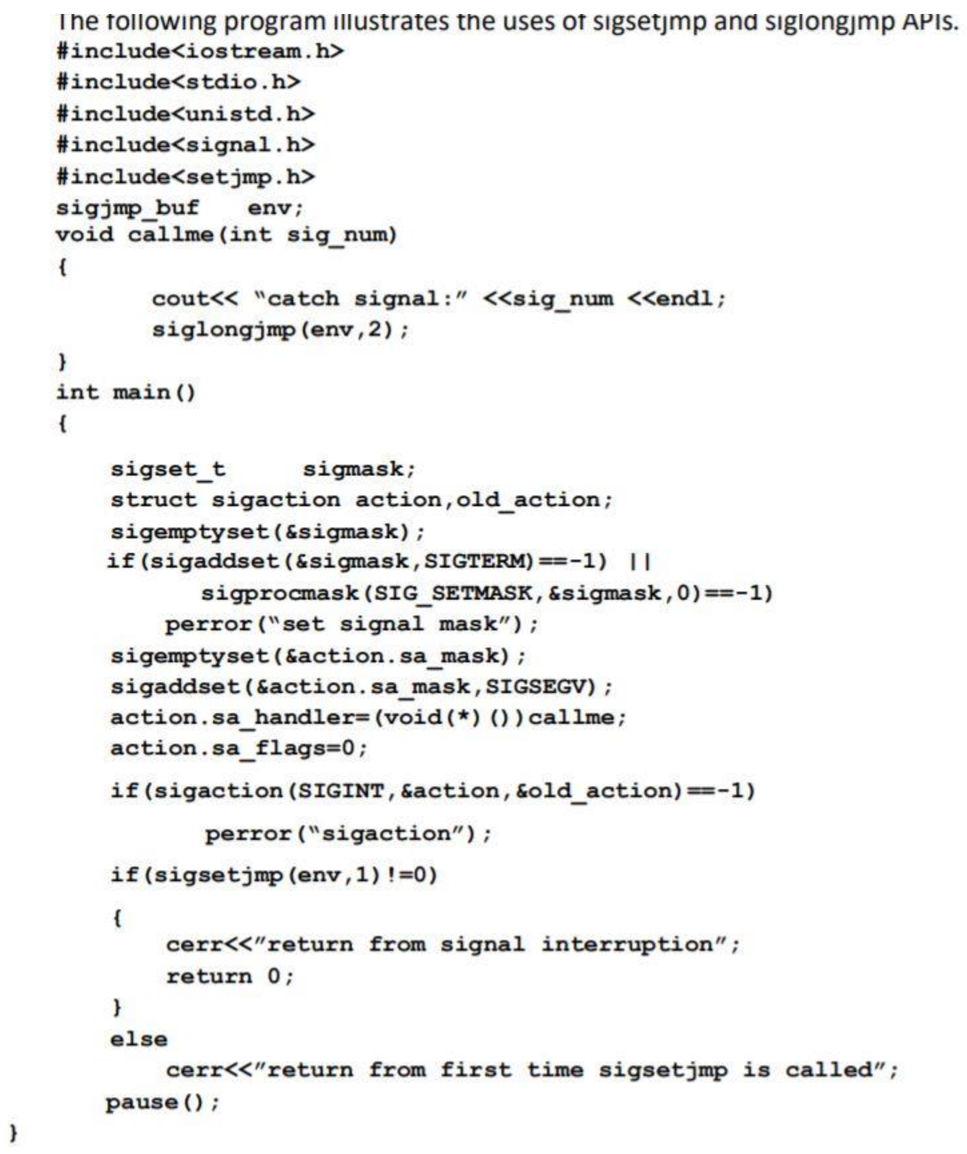
The alarm API can be called by a process to request the kernel to send the SIGALRM signal after a certain number of real clock seconds. The function prototype of the API is: Returns: 0 or number of seconds until previously set alarm The alarm API can be used to implement the sleep API:

```
#include<signal.h>
#include<stdio.h>
#include<unistd.h>
void wakeup( )
{ ; }
unsigned int sleep (unsigned int timer )
{
       Struct sigaction action;
       action.sa_handler=wakeup;
       action.sa_flags=0;
       sigemptyset(&action.sa_mask);
       if(sigaction(SIGALARM,&action,0)==-1)
       {
              perror("sigaction");
              return -1;
       }
       (void) alarm (timer);
       (void) pause( );
       return 0;
 }
```

10 b. Explain the sigsetjmp and Siglongjmp function with an example.

The function prototypes of the APIs are: The sigsetjmp and siglongjmp are created to support signal mask processing. Specifically, it is implementation- dependent on whether a process signal mask is saved and restored when it invokes the setjmp and longjmp APIs respectively. The only difference between these functions and the setjmp and longjmp functions is that sigsetjmp has an additional argument. If savemask is nonzero, then sigsetjmp also saves the current signal mask of the process in env. When siglongjmp is called, if the env argument was saved by a call to sigsetjmp with a nonzero savemask, then siglongjmp restores the saved signal mask. The siglongjmp API is usually called from user-defined signal handling functions. This is because a process signal mask is modified when a signal handler is called, and siglongjmp should be called to ensure the process signal mask is restored properly when "jumping out" from a signal handling function. The following program illustrates the uses of sigsetjmp and siglongjmp APIs

#include int sigsetjmp(sigjmp_buf env, int savemask); int siglongjmp(sigjmp_buf env, int val);



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

```
#include<iostream.h>
#include<stdio.h>
#include<unistd.h>
#include<signal.h>
#include<setjmp.h>
sigjmp_buf    env;
void callme(int sig_num)
{
       cout<< "catch signal:" <<sig_num <<endl;
       siglongjmp(env,2);
}
int main()
{

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