OPERATING SYSTEMS LAB MANUAL

SUB CODE:CS2257 **COMMON TO CSE & IT**

(Implement the following on LINUX or other Unix like platform. Use C for high level language Implementation)

1. **Write programs using the following system calls of UNIX operating system:**
   fork, exec, getpid, exit, wait, close, stat, opendir, readdir

2. **Write programs using the I/O System calls of UNIX operating system. (open, read, write, etc)**

3. **Write C programs to simulate UNIX commands like ls, grep, etc.**

4. **Given the list of processes, their CPU burst times and arrival times. Display/print the Gantt chart for FCFS and SJF. For each of the scheduling policies, compute and print the average waiting time and average turnaround time (2 sessions).**

5. **Given the list of processes, their CPU burst times and arrival times. Display/print the Gantt chart for Priority and Round robin. For each of the scheduling policies, compute and print the average waiting time and average turnaround time (2 sessions).**

6. **Develop Application using Inter-Process-Communication (Using shared memory, pipes or message queues).**

7. **Implement the Producer-Consumer problem using semaphores(Using UNIX system calls)**

8. **Implement some Memory management schemes like Paging and Segmentation.**

9. **Implement some Memory management schemes like FIRST FIT, BEST FIT & WORST FIT.**

10. **Implement any file allocation techniques(Contiguous, Linked or Indexed)**

Example for exercises 8 & 9

Free space is maintained as a linked list of nodes with each node having the starting byte address and the ending byte address of a free block. Each memory request consists of the process-id and the amount of storage space required in bytes. Allocated memory space is again maintained as a linked list of nodes with each node having the process-id, starting byte address and the ending byte address of the allocated space. When a process finishes (taken as input) the appropriate node from the allocated list should be deleted and this free space should be added to the free space list (care should be taken to merge contiguous free blocks into one single block. This results in deleting more than one node from the from the free space list and changing the start and end address in the appropriate node). For allocation use first fit, worst fit and best fit.

Total: 45 hrs

STUDY OF LINUX

INTRODUCTION TO LINUX

Linux is a generic term referring to Unix-like computer operating systems based on the Linux kernel. Their development is one of the most prominent examples of free and open source software collaboration; typically all the underlying source code can be used, freely modified, and redistributed by anyone.

The name "Linux" comes from the Linux kernel, originally written in 1991 by Linus Torvalds. The rest of the system usually comprises components such as the Apache HTTP Server, the X Window System, the K Desktop Environment, and utilities and libraries from the GNU operating system (announced in 1983 by Richard Stallman).

Many quantitative studies of free / open source software focus on topics including market share and reliability, with numerous studies specifically examining Linux. The Linux market is growing rapidly, and the revenue of servers, desktops, and packaged software running Linux was expected to exceed $35.7 billion by 2008.

LINUX FILE SYSTEM

A *file system* is the methods and data structures that an operating system uses to keep track of files on a disk or partition; that is, the way the files are organized on the disk. The word is also used to refer to a partition or disk that is used to store the files or the type of the file system.

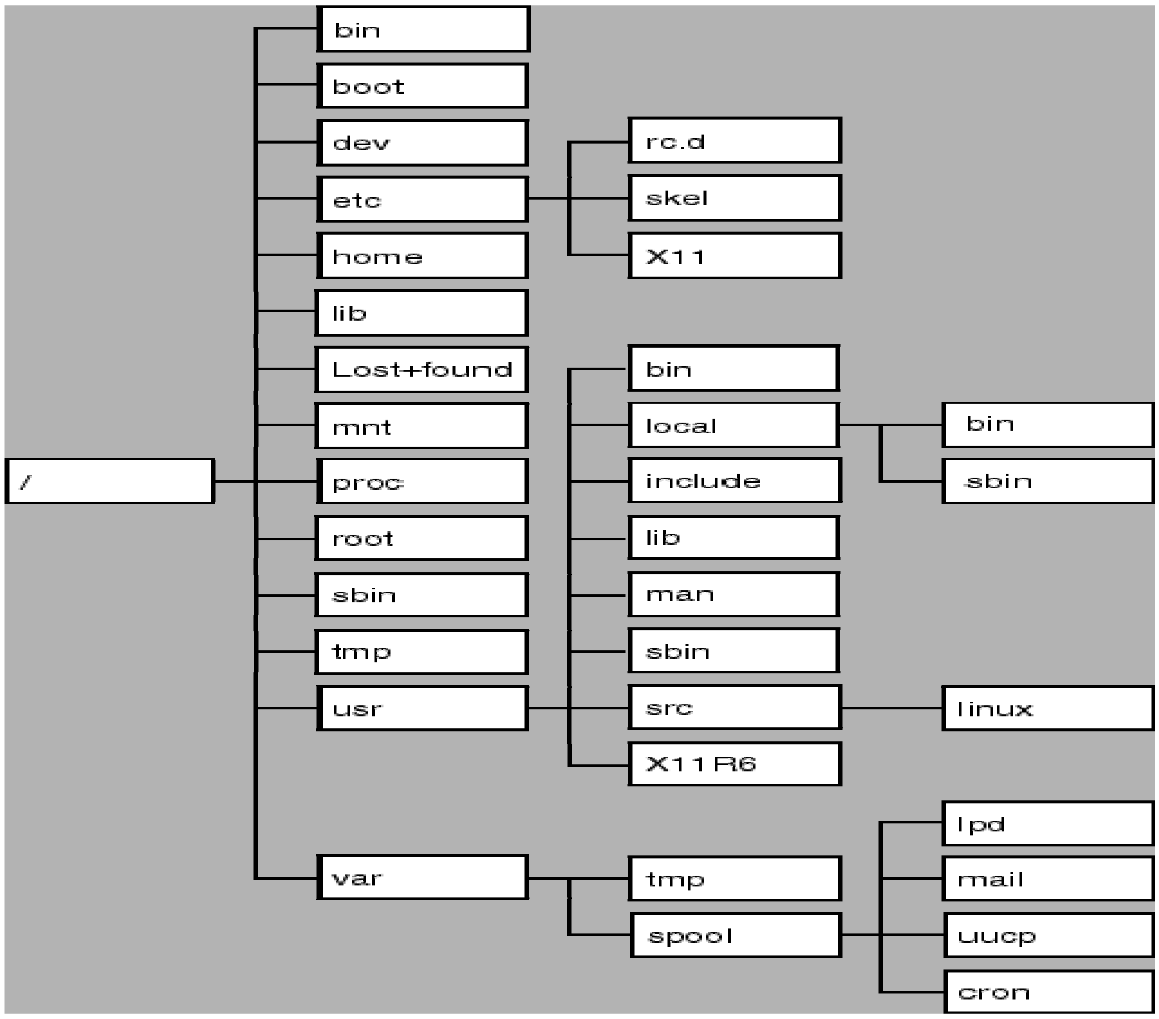
The difference between a disk or partition and the file system it contains is important. A few programs (including, reasonably enough, programs that create file systems) operate directly on the raw sectors of a disk or partition; if there is an existing file system there it will be destroyed or seriously corrupted. Most programs operate on a file system, and therefore won't work on a partition that doesn't contain one (or that contains one of the wrong type). Before a partition or disk can be used as a file system, it needs to be initialized, and the bookkeeping data structures need to be written to the disk. This process is called *making a file system.*

Most UNIX file system types have a similar general structure, although the exact details vary quite a bit. The central concepts are *superblock*, *inode*, *data block*, *directory block*, and *indirection block*. The superblock contains information about the file system as a whole, such as its size (the exact information here depends on the file system). An inode contains all information about a file, except its name. The name is stored in the directory, together with the number of the inode. A directory entry consists of a filename and the number of the inode which represents the file. The inode contains the numbers of several data blocks, which are used to store the data in the file. There is space only for a few data block numbers in the inode, however, and if more are needed, more space for pointers to the data blocks is allocated dynamically. These dynamically allocated blocks are indirect blocks; the name indicates that in order to find the data block, one has to find its number in the indirect block first.

Like UNIX, Linux chooses to have a single hierarchical directory structure. Everything starts from the root directory, represented by /, and then expands into sub-directories instead of having so-called 'drives'. In the Windows environment, one may put one's files almost anywhere: on C drive, D drive, E drive etc. Such a file system is called a hierarchical structure and is managed by the programs themselves (program directories), not by the operating system. On the other hand, Linux sorts directories descending from the root directory / according to their importance to the boot process.

Linux, like Unix also chooses to be case sensitive. What this means is that the case, whether in capitals or not, of the characters becomes very important. This feature accounts for a fairly large proportion of problems for new users especially during file transfer operations whether it may be via removable disk media such as floppy disk or over the wire by way of FTP.

The image below shows the file system of Linux



The following bin/ dev/ home/ lost+found/ proc/ sbin/ usr/ boot/ etc/ lib/ mnt/ root/ tmp/ var/ are explained in detail.

/sbin - This directory contains all the binaries that are essential to the working of the system. These include system administration as well as maintenance and hardware configuration programs.

/bin - In contrast to /sbin, the bin directory contains several useful commands that are used by both the system administrator as well as non-privileged users.

/boot - This directory contains the system.map file as well as the Linux kernel. Lilo places the boot sector backups in this directory.

/dev - This is a very interesting directory that highlights one important characteristic of the Linux filesystem - everything is a file or a directory. Look through this directory and you should see hda1, hda2 etc, which represent the various partitions on the first master drive of the system. /dev/cdrom and /dev/fd0 represent your CDROM drive and your floppy drive.

/etc - This directory contains all the configuration files for your system. Your lilo.conf file lies in this directory as does hosts, resolv.conf and fstab.

/home –These are the user home directories, which can be found under /home/username.

/lib - This contains all the shared libraries that are required by system programs. Windows equivalent to a shared library would be a DLL file.

/lost+found - Linux should always go through a proper shutdown. Sometimes your system might crash or a power failure might take the machine down. Either way, at the next boot, a lengthy filesystem check using fsck will be done. Fsck will go through the system and try to recover any corrupt files that it finds. The result of this recovery operation will be placed in this directory.

/mnt - This directory usually contains mount points or sub-directories where you mount your floppy and your CD.

/opt - This directory contains all the software and add-on packages that are not part of the default installation.

/proc - This is a special directory on your system.

/root - We talked about user home directories earlier and well this one is the home directory of the user root.

/tmp - This directory contains mostly files that are required temporarily.

/usr - This is one of the most important directories in the system as it contains all the user binaries. /usr/src/linux contains the source code for the Linux kernel.

/var - This directory contains spooling data like mail and also the output from the printer daemon. The above content briefs about Linux and the file system of Linux.

Thus the Linux file system is explained in detail.

UNIX SYSTEM CALLS

The fork() & getpid() System Call

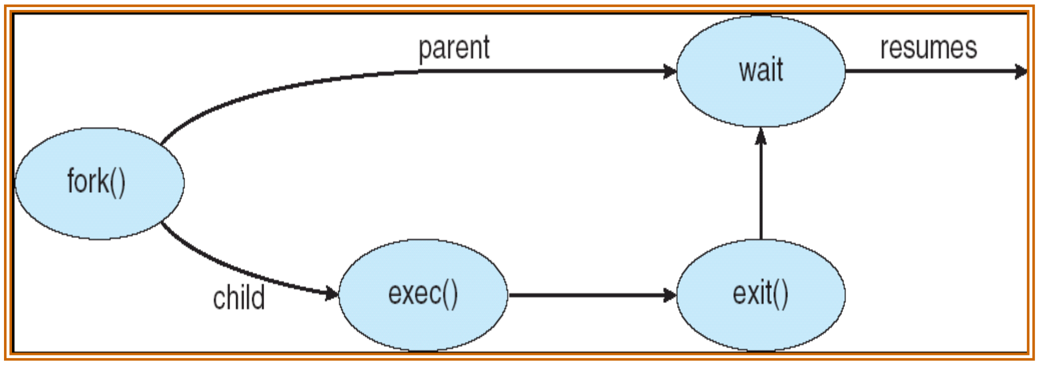
System call **fork()** is used to create processes. It takes no arguments and returns a process ID. The purpose of **fork()** is to create a ***new*** process, which becomes the *child* process of the caller. After a new child process is created, ***both*** processes will execute the next instruction following the ***fork()*** system call. Therefore, we have to distinguish the parent from the child. This can be done by testing the returned value of **fork()**:

- If **fork()** returns a negative value, the creation of a child process was unsuccessful.
- **fork()** returns a zero to the newly created child process.
- **fork()** returns a positive value, the ***process ID*** of the child process, to the parent. The returned process ID is of type **pid_t** defined in **sys/types.h**.
- Normally, the process ID is an integer. Moreover, a process can use function **getpid()** to retrieve the process ID assigned to this process.

Therefore, after the system call to **fork()**, a simple test can tell which process is the child. **Please note that Unix will make an exact copy of the parent's address space and give it to the child. Therefore, the parent and child processes have separate address spaces**.

Execution

Parent and children execute concurrently
Parent waits until children terminate



The following is a simple example of fork()

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

int main(void)
{
  printf("Hello \n");
  fork();
  printf("bye\n");
  return 0;
}
```

Hello –is printed once by parent process

bye - is printed twice, once by the parent and once by the child

If the fork system call is successful a child process is produced that continues execution at the point where it was called by the parent process.

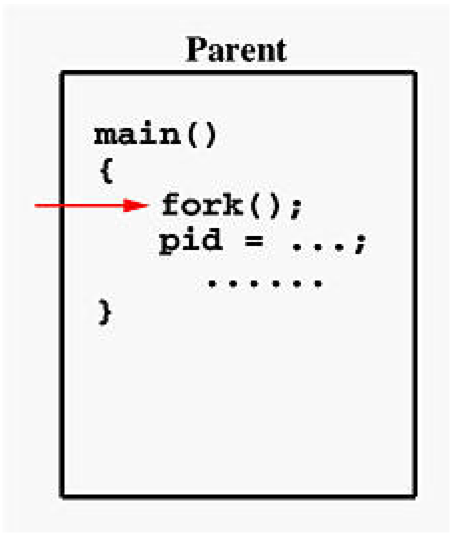
After the fork system call, both the parent and child processes are running and continue their execution at the next statement in the parent process.

Let us take another example to make the above points clear.

```
#include  <stdio.h>
#include  <string.h>
#include  <sys/types.h>
```

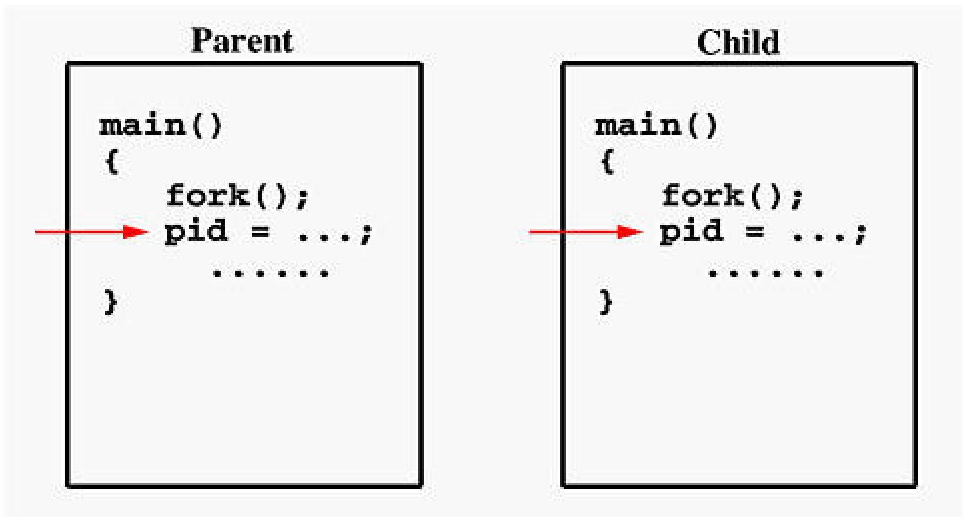
```
Void main()
{
Pid_t pid;

fork();
pid=getpid();
if(pid == -1)
printf("\n Error in creating process ");
else if(pid == 0)
printf("\nExecuting in child process, pid=%d and its parent pid = %d ",
getpid(),getppid());
else
printf("\nExecuting in parent process,pid=%d \n",getppid());
}
```

Suppose the above program executes up to the point of the call to **fork()** (marked in red color):



If the call to **fork()** is executed successfully, Unix will

- make two identical copies of address spaces, one for the parent and the other for the child.
- Both processes will start their execution at the next statement following the **fork()** call. In this case, both processes will start their execution at the assignment statement as shown below:



Both processes start their execution right after the system call **fork()**. Since both processes have identical but separate address spaces, those variables initialized **before** the **fork()** call have the same values in both address spaces. Since every process has its own address space, any modifications will be independent of the others. In other words, if the parent changes the value of its variable, the modification will only affect the variable in the parent process's address space. Other address spaces created by **fork()** calls will not be affected even though they have identical variable names.

What is the reason of using **write** rather than **printf**? It is because **printf()** is "buffered," meaning **printf()** will group the output of a process together. While buffering the output for the parent process, the child may also use **printf** to print out some information, which will also be buffered. As a result, since the output will not be send to screen immediately, you may not get the right order of the expected result. Worse, the output from the two processes may be mixed in strange ways. To overcome this problem, you may consider to use the "unbuffered" **write**.

The exec() System Call

The **exec** functions of Unix-like operating systems are a collection of functions that causes the running process to be completely replaced by the program passed as argument to the function. As a new process is not created, the process ID (PID) does not change across an execute, but the data, heap and stack of the calling process are replaced by those of the new process.

Fork-exec is a commonly used technique in Unix whereby an executing process spawns a new program. fork() is the name of the system call that the parent process uses to "divide" itself ("fork") into two identical processes. After calling fork(), the created child process is actually an exact copy of the parent - which would probably be of limited use - so it replaces itself with another process using the system call exec().

The parent process can either continue execution or wait for the child process to complete. The child, after discovering that it is the child, replaces itself completely with another program, so that the code and address space of the original program are lost.

If the parent chooses to wait for the child to die, then the parent will receive the exit code of the program that the child executed. Otherwise, the parent can ignore the child process and continue executing as it normally would; to prevent the child becoming a zombie it should wait on children at intervals or on SIGCHLD.

When the child process calls exec(), all data in the original program is lost, and replaced with a running copy of the new program. This is known as overlaying. Although all data is replaced, the file descriptors that were open in the parent are closed only if the program has explicitly marked them *close-on-exec*. This allows for the common practice of the parent creating a pipe prior to calling fork() and using it to communicate with the executed program.

Example:

```
/* using execvp to execute the contents of argv */
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  execvp(argv[1], &argv[1]);
  perror("exec failure");
  exit(1);
}
```

The wait() System Call

- A parent process usually needs to synchronize its actions by waiting until the child process has either stopped or terminated its actions.
- The wait() system call allows the parent process to suspend its activities until one of these actions has occurred.
- The wait() system call accepts a single argument, which is a pointer to an integer and returns a value defined as type pid_t.
- If the calling process does not have any child associated with it, wait will return immediately with a value of -1.
- If any child processes are still active, the calling process will suspend its activity until a child process terminates.

An example of wait():

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

Void main()
{
          int status;
          pid_t  pid;

          pid = fork();

          if(pid == -1)
            printf("\nERROR child not created ");
          else if (pid == 0) /* child process */
          {
                    printf("\n I'm the child!");
                    exit(0);
          }
          else /* parent process */
```

```
        {
                wait(&status);
                printf("\n I'm the parent!")
                printf("\n Child returned: %d\n", status)
        }
}
```

A few notes on this program:

wait(&status) causes the parent to sleep until the child process is finished execution .The exit status of the child is returned to the parent.

The stat() System Call

There are a number of system calls that a process can use to obtain file information. The most useful one is "stat" system call.
The stat() system call is used to obtain file information.

Its prototype is like this:

int stat(const char *file_name, struct stat *buf)

The ***stat*** structure is a pre-defined structure, which contains the following fields:

```
struct stat
{
dev_t st_dev; /* device */
ino_t st_ino; /* inode */
mode_t st_mode; /* protection */
nlink_t st_nlink; /* number of hard links */
uid_t st_uid; /* user ID of owner */
gid_t st_gid; /* group ID of owner */
dev_t st_rdev; /* device type (if inode device) */
off_t st_size; /* total size, in bytes */
blksize_t st_blksize; /* blocksize for filesystem I/O */
blkcnt_t st_blocks; /* number of blocks allocated */
time_t st_atime; /* time of last access */
time_t st_mtime; /* time of last modification */
time_t st_ctime; /* time of last change */
};
```

Here is a small program which use the stat call:

```
#include <iostream>
#include <cstdio>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
using namespace std;
const int N_BITS = 3;
int main(int argc, char *argv[ ])
{
unsigned int mask = 0700;
struct stat buff;
static char *perm[] = {"---", "--x", "-w-", "-wx", "r--", "r-x", "rw-", "rwx"};
if (argc > 1) {
    if ((stat(argv[1], &buff) != -1))
        {
          cout << "Permissions for " << argv[1] << " ";
          for (int i=3; i; --i)
             {
             cout << perm[(buff.st_mode & mask) >> (i-1)*N_BITS];
             mask >>= N_BITS;
             }
        cout << endl;
        }
  else
      {
       perror(argv[1]);
      return 1;
     }
  }
else
{
cerr << "Usage: " << argv[0] << " file_name\n";
return 2;
}
```

return 0;
}

The close() System Call

The close() system call is used to close a file descriptor.

SYNOPSIS

```
#include <unistd.h>

int close(int fd);
```

DESCRIPTION

close() closes a file descriptor, so that it no longer refers to any file and may be reused. If fd is the last copy of a particular file descriptor the resources associated with it are freed;

RETURN VALUE

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

The readdir() System Call

NAME

Readdir() –To read a directory

SYNOPSIS

```
#include <sys/types.h>
#include <dirent.h>
struct dirent *readdir(DIR *dir);
```

DESCRIPTION

The readdir() function returns a pointer to a dirent structure representing the next directory entry in the directory stream pointed to by dir. It returns NULL on reaching the end-of-file or if an error occurred.

On Linux, the dirent structure is defined as follows:

```
  struct dirent {
     ino_t          d_ino;       /* inode number */
     off_t          d_off;       /* offset to the next dirent */
     unsigned short d_reclen;    /* length of this record */
     unsigned char  d_type;      /* type of file */
     char           d_name[256]; /* filename */
  };
```

The data returned by readdir() may be overwritten by subsequent calls to readdir() for the same directory stream.

RETURN VALUE

The readdir() function returns a pointer to a dirent structure, or NULL if an error occurs or end-of-file is reached. On error, errno is set appropriately.

The opendir() System Call

NAME

Opendir() – To open a directory

SYNOPSIS

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

DIR *opendir(const char *name);
```

DESCRIPTION

The opendir() function opens a directory stream corresponding to the directory name, and returns a pointer to the directory stream. The stream is positioned at the first entry in the directory.

RETURN VALUE

The opendir() function returns a pointer to the directory stream. On error, NULL is returned, and errno is set appropriately.

ERRORS

EACCES Permission denied.

EMFILE Too many file descriptors in use by process.

ENFILE Too many files are currently open in the system.

ENOENT Directory does not exist, or name is an empty string.

ENOMEM Insufficient memory to complete the operation.

ENOTDIR name is not a directory.

The exit() System Call

This system call is used to terminate(normal/abnormal) the current running program.

I/O SYSTEM CALLS

File-I/O through system calls is simpler and operates at a lower level than making calls to the C file-I/O library. There are seven fundamental file-I/O system calls:

creat()	Create a file for reading or writing.
open()	Open a file for reading or writing.
close()	Close a file after reading or writing.
unlink()	Delete a file.
write()	Write bytes to file.
read()	Read bytes from file.

These calls were devised for the UNIX operating system and are not part of the ANSI C spec. Use of these system calls requires a header file named "fcntl.h":

The creat() Sytem Call

The "creat()" system call, of course, creates a file. It has the syntax:

```
int  fp; /* fp is the file descriptor variable */

  fp = creat( <filename>, <protection bits> );
```

Ex: fp=creat("students.dat",RD_WR);

This system call returns an integer, called a "file descriptor", which is a number that identifies the file generated by "creat()". This number is used by other system calls in the program to access the file. Should the "creat()" call encounter an error, it will return a file descriptor value of -1.

The "filename" parameter gives the desired filename for the new file. The "permission bits" give the "access rights" to the file. A file has three "permissions" associated with it:

1.	Write permission -	Allows data to be written to the file.
2.	Read permission -	Allows data to be read from the file.

3. Execute permission - Designates that the file is a program that can be run.

These permissions can be set for three different levels:

User level:	Permissions apply to individual user.
Group level:	Permissions apply to members of user's defined "group".
System level:	Permissions apply to everyone on the system.

The open() Sytem Call

The "open()" system call opens an existing file for reading or writing. It has the syntax:

```
<file descriptor variable> = open( <filename>, <access mode> );
```

The "open()" call is similar to the "creat()" call in that it returns a file descriptor for the given file, and returns a file descriptor of -1 if it encounters an error. However, the second parameter is an "access mode", not a permission code. There are three modes (defined in the "fcntl.h" header file):

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

For example, to open "data" for writing, assuming that the file had been created by another program, the following statements would be used:

```
int fd;

fd = open( "students.dat", O_WRONLY );
```

A few additional comments before proceeding:

A "creat()" call implies an "open()". There is no need to "creat()" a file and then "open()" it.

The close() Sytem Call

The "close()" system call is very simple. All it does is "close()" an open file when there is no further need to access it. The "close()" system call has the syntax:

```
   close( <file descriptor> );
```

The "close()" call returns a value of 0 if it succeeds, and returns -1 if it encounters an error.

The unlink() Sytem Call

The "unlink()" system call deletes a file. It has the syntax:

```
   unlink( <file_name_string> );
```

It returns 0 on success and -1 on failure.

The write() Sytem Call

The "write()" system call writes data to an open file. It has the syntax:

```
   write( <file descriptor>, <buffer>, <buffer length> );
```

The file descriptor is returned by a "creat()" or "open()" system call. The "buffer" is a pointer to a variable or an array that contains the data; and the "buffer length" gives the number of bytes to be written into the file.

While different data types may have different byte lengths on different systems, the "sizeof()" statement can be used to provide the proper buffer length in bytes. A "write()" call could be specified as follows:

```
   float array[10];

    write( fd, array, sizeof( array ) );
```

The "write()" function returns the number of bytes it actually writes. It will return -1 on an error.

The read() Sytem Call

The "read()" system call reads data from a open file. Its syntax is exactly the same as that of the "write()" call:

read(<file descriptor>, <buffer>, <buffer length>);

The "read()" function returns the number of bytes it actually returns. At the end of file it returns 0, or returns -1 on error.

Example:

```
#include<stdio.h>
#include<fcntl.h>
#include<stdlib.h>
main(int argc,char *argv[])
{
int fd,i;
char ch[1];
if (argc<2)
{ printf("Usage: mycat filename\n");
exit(0);
}
fd=open(argv[1],O_RDONLY);
if(fd==-1)
printf("%s is not exist",argv[1]);
else
{
printf("Contents of the file %s is : \n",argv[1]);
while(read(fd,ch,1)>0)
printf("%c",ch[0]);
close(fd);

}
```

UNIX COMMANDS

Command: mkdir

Syntax mkdir [options] directory name

Description:
The "mkdir" command is used to create new directories (sub-directories).

Examples:

mkdir tmp
This command creates a new directory named "tmp" in your current directory. (This example assumes that you have the proper permissions to create a new sub-directory in your current working directory).

mkdir memos letters e-mail
This command creates three new sub-directories (memos, letters, and e-mail) in the current directory.

mkdir /usr/fred/tmp
This command creates a new directory named "tmp" in the directory "/usr/fred". "tmp" is now a sub-directory of "/usr/fred". (This example assumes that you have the proper permissions to create a new directory in /usr/fred.)

mkdir -p /home/joe/customer/acme
This command creates a new directory named /home/joe/customer/acme, and creates any intermediate directories that are needed. If only /home/joe existed to begin with, then the directory "customer" is created, and the directory "acme" is created inside of customer.

Command rmdir

Syntax:

rmdir [options] directories

Description:

The "rm" command is used to remove files and directories. (Warning - be very careful when removing 17files and directories!)

Examples:

rm Chapter1.bad

This command deletes the file named "Chapter1.bad" (assuming you have permission to delete this file).

rm Chapter1 Chapter2 Chapter3

This command deletes the files named "Chapter1","Chapter2", and "Chapter3".

rm -i Chapter1 Chapter2 Chapter3

This command prompts you before deleting any of the three files specified. The -i option stands for *inquire*. You must answer y (for yes) for each file you really want to delete. This can be a safer way to delete files.

rm *.html

This command deletes all files in the current directory whose filename ends with the characters ".html".

rm index*

This command deletes all files in the current directory whose filename begins with the characters "index".

rm -r new-novel

This command deletes the directory named "new-novel". This directory, and all of its' contents, are erased from the disk, including any sub-directories and files.

Command cd, chdir

Syntax:
cd [name of directory you want to move to]

Description:
"cd" stands for change directory. It is the primary command for moving around the filesystem.

Examples:
cd /usr

This command moves you to the "/usr" directory. "/usr" becomes your current working directory.

cd /usr/fred
Moves you to the "/usr/fred" directory.

cd /u*/f*
Moves you to the "/usr/fred" directory - if this is the only directory matching this wildcard pattern.

cd
Issuing the "cd" command without any arguments moves you to your *home* directory.

cd -
Using the Korn shell, this command moves you back to your previous working directory. This is very useful when you're in the middle of a project, and keep moving back-and-forth between two directories.

The ls Command

Syntax:
ls [options] [names]

Description:
"ls" stands for list. It is used to list information about files and directories.

Examples:

ls

This is the basic "ls" command, with no options. It provides a very basic listing of the files in your current working directory. Filenames beginning with a decimal are considered *hidden* files, and they are not shown.

ls -a

The -a option tells the ls command to report information about all files, including hidden files.

ls -l

The -l option tells the "ls" command to provide a *long* listing of information about the files and directories it reports. The long listing will provide important information about file permissions, user and group ownership, file size, and creation date.

ls -al

This command provides a *long* listing of information about *all* files in the current directory. It combines the functionality of the -a and -l options. *This is probably the most used version of the ls command.*

ls -al /usr

This command lists long information about all files in the "/usr" directory.

ls -alR /usr | more

This command lists long information about all files in the "/usr" directory, and all sub-directories of /usr. The -R option tells the ls command to provide a *recursive* listing of all files and sub-directories.

ls -ld /usr

Rather than list the files contained in the /usr directory, this command lists information about the /usr directory itself (without generating a listing of the contents of /usr). This is very useful when you want to check the permissions of the directory, and not the files the directory contains.

SOURCE CODE: LS SIMULATION

```
#include<stdio.h>
#include<dirent.h>
int main()
{
   struct dirent **namelist;
   int n,i;
   char pathname[100];
   getcwd(pathname);
   n=scandir(pathname,&namelist,0,alphasort);
   if(n<0)
   printf("Error");
   else
      for(i=0;i<n;i++)
      printf("%s\n",namelist[i]->d_name);
}
```

The grep Command

Syntax:

grep flag PatternList filename

The grep command is a Unix command that is used to search for patterns within one or more files. It displays the name of the file that contains the matched line.

Following is a list of some flags used with the grep command:Flag
Description

-c	Displays a count of matching lines.
-f	Specifies a file containing search patterns.
-p	Displays the entire paragraph containing matched lines.
-v	Displays all lines not matching the specified pattern.

PatternList	Specifies one or more patterns to be used during the search.
File	Specifies the name of the file that is to be searched for the pattern.

Example:

Simulation of grep command:

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

/* calculate filesize */
int calc_bufsize(char *filename)
{
        struct stat st;

        stat(filename, &st);
        return ((int)st.st_size);
}

int main(int argc, char *argv[])
{
        if(argc < 3)
        {
                printf("Usage:\n\t%s <word> <files> ...\n", argv[0]);
                return -1;
        }

        FILE *fp;
        char *filename;
        int x = 2;

        /* process each file */
        for(x; x != argc; x++)
        {
                filename = argv[x];

                if( (fp = fopen(filename, "r")) == NULL)
                {
                        printf("Failed to open file: %s\n", argv[2]);
                        return -2;
                }
                int BUFSIZE = calc_bufsize(filename);

                /* read ENTIRE file into buf[] */
                char buf[BUFSIZE];
                fread(&buf, sizeof(char), BUFSIZE, fp);

                /* search buf for word (case sensitive) */
                char *ans = strstr(buf, argv[1]);

                /* word found, print filename */
                if(ans != NULL)
                        printf("%s\n", filename);

                /* word not found, do nothing */

                fclose(fp);
        }
        return 0;
}
```

compile this code and run as follows

```
prog_name word_to_be_found list of files to search in
```

notes:

- search is case sensitive
- loads entire file into memory, so be careful not to search large files

CPU SCHEDULING ALGORITHMS

IMPLEMENTATION OF FIRST COME FIRST SERVE SCHEDULING ALGORITHM

1. Start the process
2. Get the number of processes to be inserted
3. Get the value for burst time of each process from the user
4. Having allocated the burst time(bt) for individual processes , Start with the first process from it's initial position let other process to be in queue
5. Calculate the waiting time(wt) and turn around time(tat) as
   Wt(pi) = wt(pi-1) + tat(pi-1) (i.e wt of current process = wt of previous process + tat of previous process)
   tat(pi) = wt(pi) + bt(pi) (i.e tat of current process = wt of current process + bt of current process)
6. Calculate the total and average waiting time and turn around time
7. Display the values
8. Stop the process

IMPLEMENTATION OF SHORTEST JOB FIST SCHEDULING ALGORITHM

1. Start the process
2. Get the number of processes to be inserted
3. Sort the processes according to the burst time and allocate the one with shortest burst to execute first
4. If two process have same burst length then FCFS scheduling algorithm is used
5. Calculate the total and average waiting time and turn around time
6. Display the values
7. Stop the process

IMPLEMENTATION OF PRIORITY SCHEDULING ALGORITHM

1. Start the process
2. Get the number of processes to be inserted

3. Get the corresponding priority of processes
4. Sort the processes according to the priority and allocate the one with highest priority to execute first
5. If two process have same priority then FCFS scheduling algorithm is used
6. Calculate the total and average waiting time and turn around time
7. Display the values
8. Stop the process

IMPLEMENTATION OF ROUND ROBIN SCHEDULING ALGORITHM

1. Start the process
2. Get the number of elements to be inserted
3. Get the value for burst time for individual processes
4. Get the value for time quantum
5. Make the CPU scheduler go around the ready queue allocating CPU to each process for the time interval specified
6. Make the CPU scheduler pick the first process and set time to interrupt after quantum. And after it's expiry dispatch the process
7. If the process has burst time less than the time quantum then the process is released by the CPU
8. If the process has burst time greater than time quantum then it is interrupted by the OS and the process is put to the tail of ready queue and the schedule selects next process from head of the queue
9. Calculate the total and average waiting time and turn around time
10. Display the results
11. Stop the process

****