1. **Design, Develop and Implement Basics of UNIX commands.**

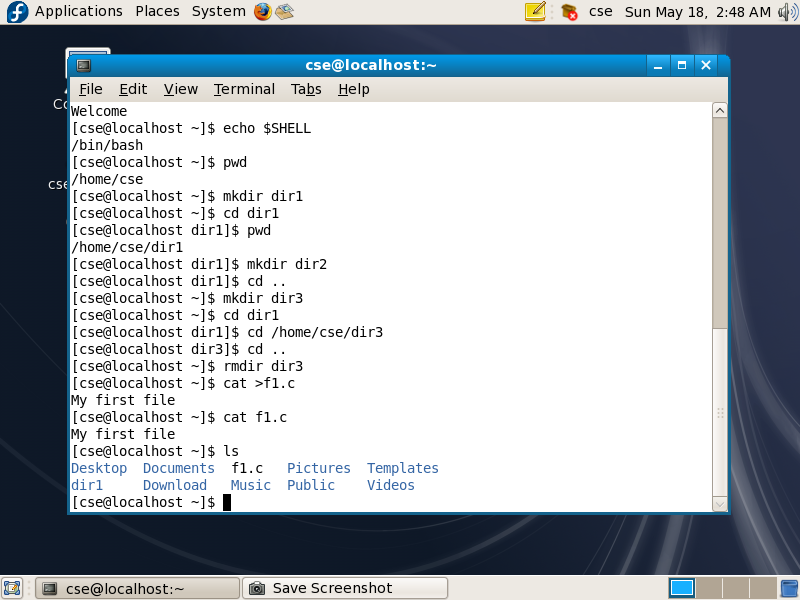
**Commands:**

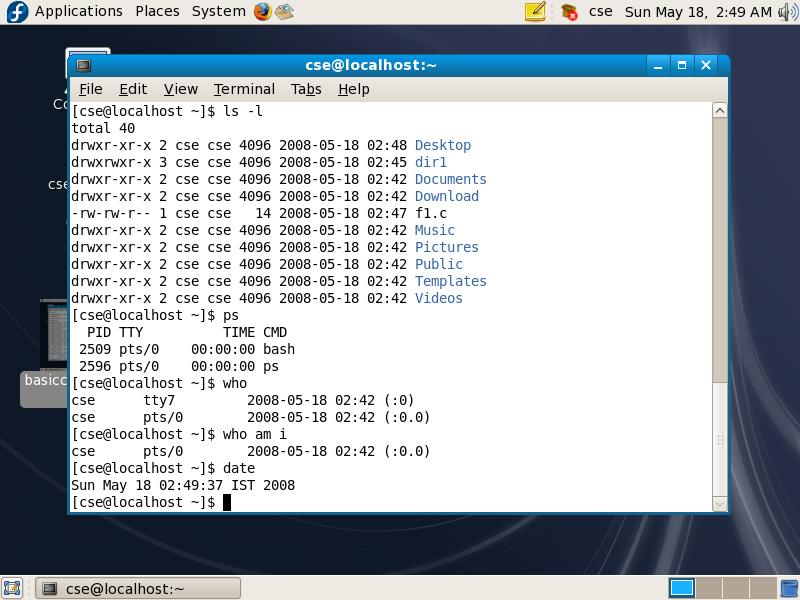
1. **echo command:** **echo** command in linux is used to display line of text/string that are passed as an argument . This is a built in command that is mostly used in shell scripts and batch files to output status text to the screen or a file.

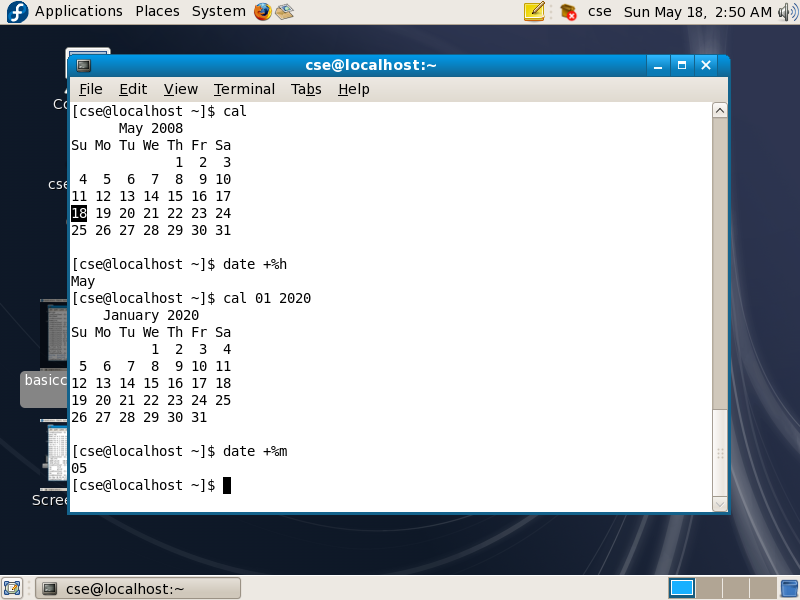
**echo [string]**

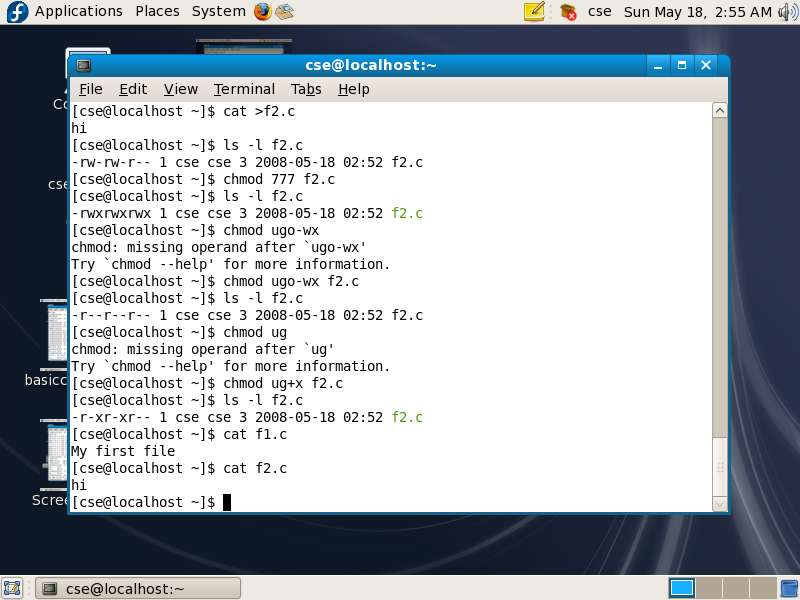
1. **pwd** stands for Print Working Directory: It prints the path of the working directory, starting from the root. pwd is shell built-in command(pwd) or an actual binary(/bin/pwd).
2. **mkdir** command in Linux allows the user to create directories (also referred to as folders in some operating systems ). This command can create multiple directories at once as well as set the permissions for the directories.
3. **rmdir** command is used remove empty directories from the filesystem in Linux. The rmdir command removes each and every directory specified in the command line only if these directories are empty.
4. **cd** command in linux known as change directory command. It is used to change current working directory.
5. **cat**(concatenate) command is very frequently used in Linux. It reads data from the file and gives their content as output. It helps us to create, view, concatenate files.
6. **ls**( listing files) is used to show list of files on your UNIX or Linux system.
7. **ls-l** is used To show long listing information about the file/directory.
8. **ps** is used to view the processes that is currently running.
9. **who** command is used to find out the following information :  
   1. Time of last system boot  
   2. Current run level of the system  
   3. List of logged in users and more.
10. **who am i** command displays the username of the current user when this command is invoked.
11. **date**: displays current date.
12. **cal**: displays the calendar of the current month.
13. **cp**: copy the contents of one file to another
14. **rm**: remove a particular file.

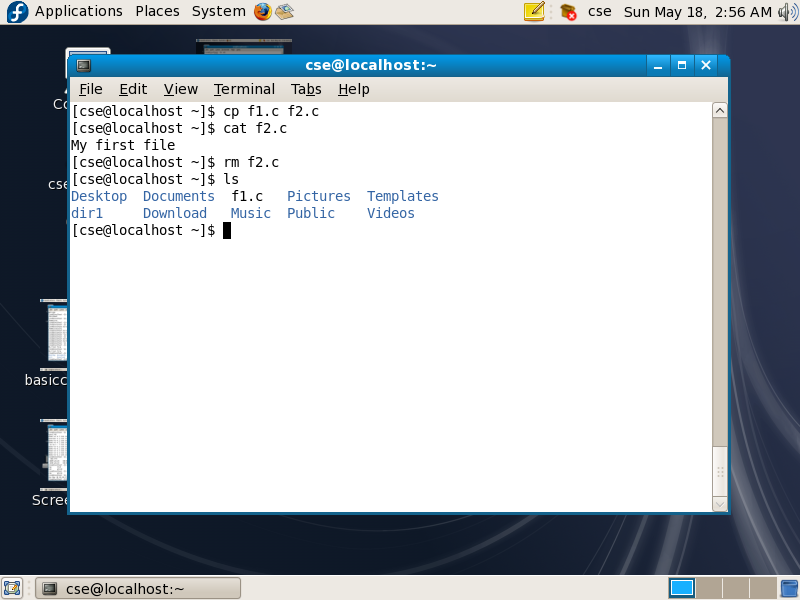
**Output:**

****

****

****

****

****

**2. Design, Develop and Implement a Program to implement a shell.**

**Program:**

#include<stdio.h>

#include<string.h>

#include<stdlib.h>

#include<unistd.h>

#include<sys/types.h>

#include<sys/wait.h>

#include<readline/readline.h>

#include<readline/history.h>

#define MAXCOM 1000 // max number of letters to be supported

#define MAXLIST 100 // max number of commands to be supported

// Clearing the shell using escape sequences

#define clear() printf("\033[H\033[J")

// Greeting shell during startup

void init\_shell()

{

clear();

printf("\n\n\n\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"

"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

printf("\n\n\n\t\*\*\*\*MY SHELL\*\*\*\*");

printf("\n\n\t-USE AT YOUR OWN RISK-");

printf("\n\n\n\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"

"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

char\* username = getenv("USER");

printf("\n\n\nUSER is: @%s", username);

printf("\n");

sleep(1);

clear();

}

// Function to take input

int takeInput(char\* str)

{

char\* buf;

buf = readline("\n>>> ");

if (strlen(buf) != 0) {

add\_history(buf);

strcpy(str, buf);

return 0;

} else {

return 1;

}

}

// Function to print Current Directory.

void printDir()

{

char cwd[1024];

getcwd(cwd, sizeof(cwd));

printf("\nDir: %s", cwd);

}

// Function where the system command is executed

void execArgs(char\*\* parsed)

{

// Forking a child

pid\_t pid = fork();

if (pid == -1) {

printf("\nFailed forking child..");

return;

} else if (pid == 0) {

if (execvp(parsed[0], parsed) < 0) {

printf("\nCould not execute command..");

}

exit(0);

} else {

// waiting for child to terminate

wait(NULL);

return;

}

}

// Function where the piped system commands is executed

void execArgsPiped(char\*\* parsed, char\*\* parsedpipe)

{

// 0 is read end, 1 is write end

int pipefd[2];

pid\_t p1, p2;

if (pipe(pipefd) < 0) {

printf("\nPipe could not be initialized");

return;

}

p1 = fork();

if (p1 < 0) {

printf("\nCould not fork");

return;

}

if (p1 == 0) {

// Child 1 executing..

// It only needs to write at the write end

close(pipefd[0]);

dup2(pipefd[1], STDOUT\_FILENO);

close(pipefd[1]);

if (execvp(parsed[0], parsed) < 0) {

printf("\nCould not execute command 1..");

exit(0);

}

} else {

// Parent executing

p2 = fork();

if (p2 < 0) {

printf("\nCould not fork");

return;

}

// Child 2 executing..

// It only needs to read at the read end

if (p2 == 0) {

close(pipefd[1]);

dup2(pipefd[0], STDIN\_FILENO);

close(pipefd[0]);

if (execvp(parsedpipe[0], parsedpipe) < 0) {

printf("\nCould not execute command 2..");

exit(0);

}

} else {

// parent executing, waiting for two children

wait(NULL);

wait(NULL);

}

}

}

// Help command builtin

void openHelp()

{

puts("\n\*\*\*WELCOME TO MY SHELL HELP\*\*\*"

"\nCopyright @ Suprotik Dey"

"\n-Use the shell at your own risk..."

"\nList of Commands supported:"

"\n>cd"

"\n>ls"

"\n>exit"

"\n>all other general commands available in UNIX shell"

"\n>pipe handling"

"\n>improper space handling");

return;

}

// Function to execute builtin commands

int ownCmdHandler(char\*\* parsed)

{

int NoOfOwnCmds = 4, i, switchOwnArg = 0;

char\* ListOfOwnCmds[NoOfOwnCmds];

char\* username;

ListOfOwnCmds[0] = "exit";

ListOfOwnCmds[1] = "cd";

ListOfOwnCmds[2] = "help";

ListOfOwnCmds[3] = "hello";

for (i = 0; i < NoOfOwnCmds; i++) {

if (strcmp(parsed[0], ListOfOwnCmds[i]) == 0) {

switchOwnArg = i + 1;

break;

}

}

switch (switchOwnArg) {

case 1:

printf("\nGoodbye\n");

exit(0);

case 2:

chdir(parsed[1]);

return 1;

case 3:

openHelp();

return 1;

case 4:

username = getenv("USER");

printf("\nHello %s.\nMind that this is "

"not a place to play around."

"\nUse help to know more..\n",

username);

return 1;

default:

break;

}

return 0;

}

// function for finding pipe

int parsePipe(char\* str, char\*\* strpiped)

{

int i;

for (i = 0; i < 2; i++) {

strpiped[i] = strsep(&str, "|");

if (strpiped[i] == NULL)

break;

}

if (strpiped[1] == NULL)

return 0; // returns zero if no pipe is found.

else {

return 1;

}

}

// function for parsing command words

void parseSpace(char\* str, char\*\* parsed)

{

int i;

for (i = 0; i < MAXLIST; i++) {

parsed[i] = strsep(&str, " ");

if (parsed[i] == NULL)

break;

if (strlen(parsed[i]) == 0)

i--;

}

}

int processString(char\* str, char\*\* parsed, char\*\* parsedpipe)

{

char\* strpiped[2];

int piped = 0;

piped = parsePipe(str, strpiped);

if (piped) {

parseSpace(strpiped[0], parsed);

parseSpace(strpiped[1], parsedpipe);

} else {

parseSpace(str, parsed);

}

if (ownCmdHandler(parsed))

return 0;

else

return 1 + piped;

}

int main()

{

char inputString[MAXCOM], \*parsedArgs[MAXLIST];

char\* parsedArgsPiped[MAXLIST];

int execFlag = 0;

init\_shell();

while (1) {

// print shell line

printDir();

// take input

if (takeInput(inputString))

continue;

// process

execFlag = processString(inputString,

parsedArgs, parsedArgsPiped);

// execflag returns zero if there is no command

// or it is a builtin command,

// 1 if it is a simple command

// 2 if it is including a pipe.

// execute

if (execFlag == 1)

execArgs(parsedArgs);

if (execFlag == 2)

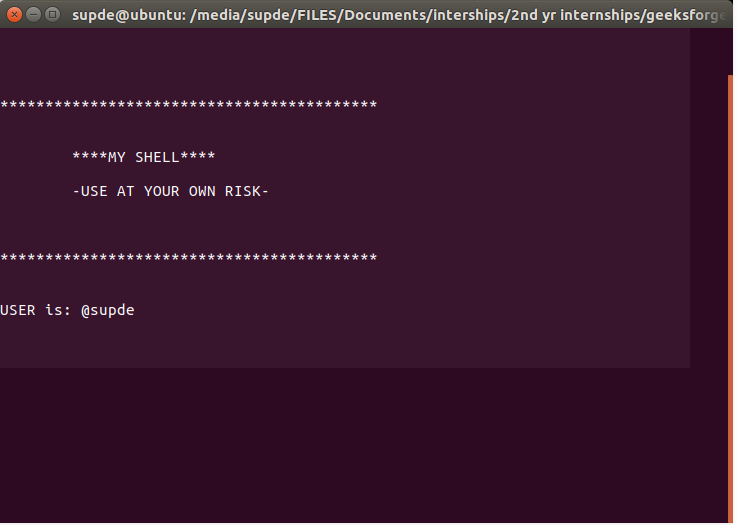
execArgsPiped(parsedArgs, parsedArgsPiped);

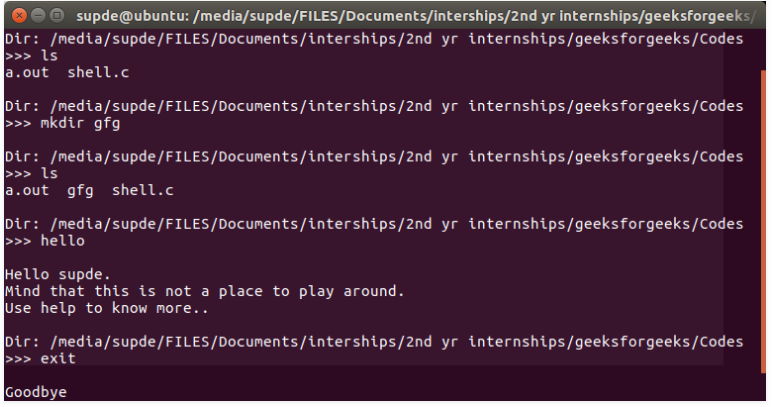
}

return 0;

}

**Output**





**3. Design, Develop and Implement a program to change current working directory and display the inode details for each file in the new directory**

**Program:**

include<stdlib.h>

#include<stdio.h>

#include<string.h>

main(int argc, char \*argv[])

{

char d[50];

if(argc==2)

{

bzero(d,sizeof(d));

strcat(d,"ls ");

strcat(d,"-i ");

strcat(d,argv[1]);

system(d);

}

else

printf("\nInvalid No. of inputs");

}

**output:**

student@ubuntu:~$ mkdir dd

student@ubuntu:~$ cd dd

student@ubuntu:~/dd$ cat >f1

hello

^z

student@ubuntu:~/dd$ cd

student@ubuntu:~$gcc –o flist.out flist.c

student@ubuntu:~$./flist.out dd

hello

46490 f1

**4.** **Design, Develop and Implement a Parent process – Child process Relationship.**

**Program:**

#include<stdio.h>

int main()

{

    for(int i=0;i<5;i++) // loop will run n times (n=5)

    {

        if(fork() == 0)

        {

            printf("[son] pid %d from [parent] pid %d\n",getpid(),getppid());

            exit(0);

        }

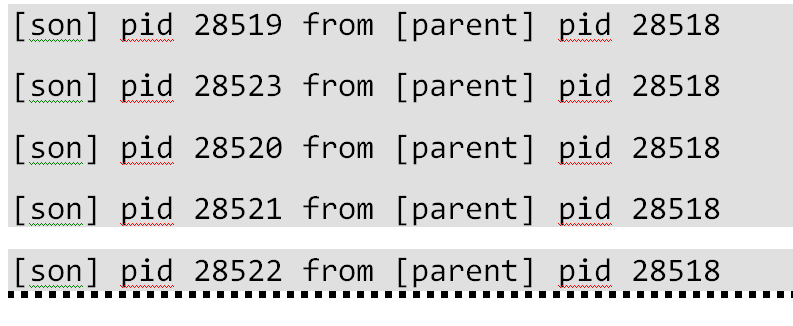
    }

    for(int i=0;i<5;i++) // loop will run n times (n=5)

    wait(NULL);

}

**Output:**



**5. Design, Develop and Implement a Program that creates a child process. Parent process writes data to pipe and child process reads the data from pipe and prints it on the screen.**

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#include<sys/types.h>

#include<string.h>

#include<sys/wait.h>

int main()

{

// We use two pipes

// First pipe to send input string from parent

// Second pipe to send concatenated string from child

int fd1[2]; // Used to store two ends of first pipe

int fd2[2]; // Used to store two ends of second pipe

char fixed\_str[] = "os\_labpgm5";

char input\_str[100];

pid\_t p;

if (pipe(fd1)==-1)

{

fprintf(stderr, "Pipe Failed" );

return 1;

}

if (pipe(fd2)==-1)

{

fprintf(stderr, "Pipe Failed" );

return 1;

}

scanf("%s", input\_str);

p = fork();

if (p < 0)

{

fprintf(stderr, "fork Failed" );

return 1;

}

// Parent process

else if (p > 0)

{

char concat\_str[100];

close(fd1[0]); // Close reading end of first pipe

// Write input string and close writing end of first

// pipe.

write(fd1[1], input\_str, strlen(input\_str)+1);

close(fd1[1]);

// Wait for child to send a string

wait(NULL);

close(fd2[1]); // Close writing end of second pipe

// Read string from child, print it and close

// reading end.

read(fd2[0], concat\_str, 100);

printf("Concatenated string %s\n", concat\_str);

close(fd2[0]);

}

// child process

else

{

close(fd1[1]); // Close writing end of first pipe

// Read a string using first pipe

char concat\_str[100];

read(fd1[0], concat\_str, 100);

// Concatenate a fixed string with it

int k = strlen(concat\_str);

int i;

for (i=0; i<strlen(fixed\_str); i++)

concat\_str[k++] = fixed\_str[i];

concat\_str[k] = '\0'; // string ends with '\0'

// Close both reading ends

close(fd1[0]);

close(fd2[0]);

// Write concatenated string and close writing end

write(fd2[1], concat\_str, strlen(concat\_str)+1);

close(fd2[1]);

exit(0);

}

}

**Output:**

Concatenated string os\_labpgm5

**6. Design, Develop and Implement a Program for Process system calls.**

**Program:**

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

void main(int argc,char \*arg[])

{

int pid; pid=fork();

if(pid<0)

{

printf("fork failed");

exit(1);

}

else if(pid==0)

{

execlp("whoami","ls",NULL);

exit(0);

}

else

{

printf("\n Process id is -%d\n",getpid());

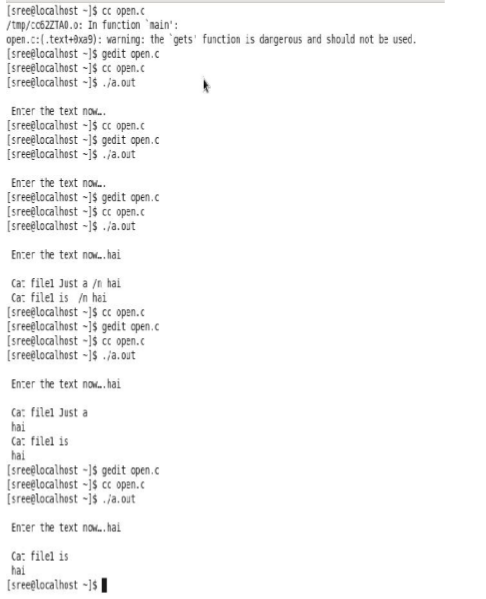
wait(NULL);

exit(0);

}

}

**Output:**

****

**PART B**

**7. Design, Develop and Implementation of CPU scheduling by using**

**a) Round Robin**

**b) FCFS**

**DESCRIPTION**

**ROUND ROBIN CPU SCHEDULING ALGORITHM:**  For round robin scheduling algorithm, read the number of processes/jobs in the system, their CPU burst times, and the size of the time slice. Time slices are assigned to each process in equal portions and in circular order, handling all processes execution. This allows every process to get an equal chance. Calculate the waiting time and turnaround time of each of the processes accordingly.

**FCFS CPU SCHEDULING ALGORITHM**: For FCFS scheduling algorithm, read the number of processes/jobs in the system, their CPU burst times. The scheduling is performed on the basis of arrival time of the processes irrespective of their other parameters. Each process will be executed according to its arrival time. Calculate the waiting time and turnaround time of each of the processes accordingly.

**Program :**

**Round Robin Scheduling:**

#include main()

{

int i,j,n,bu[10],wa[10],tat[10],t,ct[10],max;

float awt=0,att=0,temp=0;

clrscr();

printf("Enter the no of processes -- ");

scanf("%d",&n);

for(i=0;i<=t)

{

tat[i]=temp+bu[i

temp=temp+bu[i];

bu[i]=0;

}

else

{

bu[i]=bu[i]-t;

temp=temp+t;

}

for(i=0;i<n;i++)

{ wa[i]=tat[i]-ct[i];

att+=tat[i];

awt+=wa[i];

}

printf("\nThe Average Turnaround time is -- %f",att/n);

printf("\nThe Average Waiting time is -- %f ",awt/n);

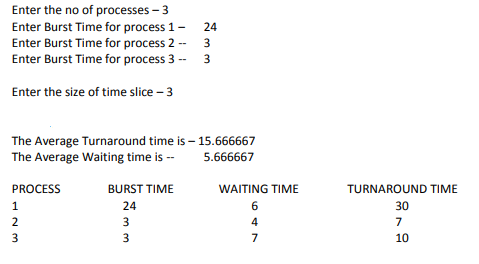
printf("\n\tPROCESS\t BURST TIME \t WAITING TIME\tTURNAROUND TIME\n");

for(i=0;i<n;i++)

printf("\t%d \t %d \t\t %d \t\t %d \n",i+1,ct[i],wa[i],tat[i]);

}

**Output:**

****

**FCFS Scheduling:**

#include<stdio.h>

#include<conio.h>

main()

{

int bt[20], wt[20], tat[20], i, n;

float wtavg, tatavg;

printf("\nEnter the number of processes -- ");

scanf("%d", &n);

for(i=0;i<n;i++)

{

printf("\nEnter Burst Time for Process %d -- ", i);

scanf("%d", &bt[i]);

}

wt[0] = wtavg = 0;

tat[0] = tatavg = bt[0];

for(i=1;i<n;i++)

{

wt[i] = wt[i-1] +bt[i-1];

tat[i] = tat[i-1] +bt[i];

wtavg = wtavg + wt[i];

tatavg = tatavg + tat[i];

}

printf("\t PROCESS \tBURST TIME \t WAITING TIME\t TURNAROUND TIME\n");

for(i=0;i<n;i++)

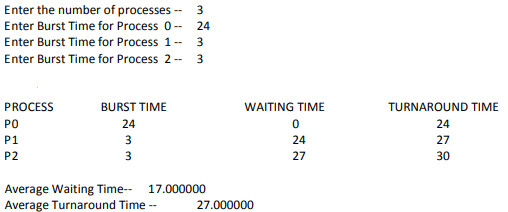
printf("\n\t P%d \t\t %d \t\t %d \t\t %d", i, bt[i], wt[i], tat[i]);

printf("\nAverage Waiting Time -- %f", wtavg/n);

printf("\nAverage Turnaround Time -- %f", tatavg/n);

}

**Output:**



**8. Design, Develop and Implement Implementation of CPU scheduling by using**

**a) Shortest job first**

**b) Priority**

**DESCRIPTION**

**SJF CPU SCHEDULING ALGORITHM:** For SJF scheduling algorithm, read the number of processes/jobs in the system, their CPU burst times. Arrange all the jobs in order with respect to their burst times. There may be two jobs in queue with the same execution time, and then FCFS approach is to be performed. Each process will be executed according to the length of its burst time. Then calculate the waiting time and turnaround time of each of the processes accordingly.

**PRIORITY CPU SCHEDULING ALGORITHM**: For priority scheduling algorithm, read the number of processes/jobs in the system, their CPU burst times, and the priorities. Arrange all the jobs in order with respect to their priorities. There may be two jobs in queue with the same priority, and then FCFS approach is to be performed. Each process will be executed according to its priority. Calculate the waiting time and turnaround time of each of the processes accordingly.

**SJF CPU SCHEDULING:**

#include<stdio.h>

#include<conio.h>

struct process

{

int pid;

int bt;

int wt;

int tt;

}p[10],temp;

int main()

{

int i,j,n,totwt,tottt;

float avg1,avg2;

clrscr();

printf("\nEnter the number of process:\t");

scanf("%d",&n);

for(i=1;i<=n;i++)

{

p[i].pid=i;

printf("\nEnter the burst time:\t");

scanf("%d",&p[i].bt);

}

for(i=1;i<n;i++){

for(j=i+1;j<=n;j++)

{

if(p[i].bt>p[j].bt)

{

temp.pid=p[i].pid;

p[i].pid=p[j].pid;

p[j].pid=temp.pid;

temp.bt=p[i].bt;p[i].bt=p[j].bt;

p[j].bt=temp.bt;

}}}

p[1].wt=0;

p[1].tt=p[1].bt+p[1].wt;

i=2;

while(i<=n){

p[i].wt=p[i-1].bt+p[i-1].wt;

p[i].tt=p[i].bt+p[i].wt;

i++;

}

i=1;

totwt=tottt=0;

printf("\nProcess id \tbt \twt \ttt");

while(i<=n){

printf("\n\t%d \t%d \t%d t%d\n",p[i].pid,p[i].bt,p[i].wt,p[i].tt);

totwt=p[i].wt+totwt;

tottt=p[i].tt+tottt;

i++;

} avg1=totwt/n;

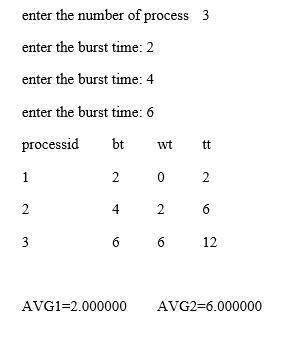
avg2=tottt/n;

printf("\nAVG1=%f\t AVG2=%f",avg1,avg2);

getch();

return 0; }

**Output:**

****

**PRIORITY CPU SCHEDULING**

#include<stdio.h>

#include<conio.h>

struct process

{

int pid;

int bt;

int wt;

int tt;

int prior;

}

p[10],temp;

int main()

{

int i,j,n,totwt,tottt,arg1,arg2;

clrscr();

printf("enter the number of process");

scanf("%d",&n);

for(i=1;i<=n;i++)

{

p[i].pid=i;

printf("enter the burst time");

scanf("%d",&p[i].bt);

printf("\n enter the priority");

scanf("%d",&p[i].prior);

}

for(i=1;i<n;i++)

{

for(j=i+1;j<=n;j++)

{

if(p[i].prior>p[j].prior)

{

temp.pid=p[i].pid;

p[i].pid=p[j].pid;

p[j].pid=temp.pid;

temp.bt=p[i].bt;

p[i].bt=p[j].bt;

p[j].bt=temp.bt;

temp.prior=p[i].prior;

p[i].prior=p[j].prior;

p[j].prior=temp.prior;

}

}

}

p[i].wt=0;

p[1].tt=p[1].bt+p[1].wt;

i=2;

while(i<=n)

{

p[i].wt=p[i-1].bt+p[i-1].wt;

p[i].tt=p[i].bt+p[i].wt;

i++;

}

i=1;

totwt=tottt=0;

printf("\n process to \t bt \t wt \t tt");

while(i<=n)

{

printf("\n%d\t %d\t %d\t %d\t",p[i].pid,p[i].bt,p[i].wt,p[i].tt);

totwt=p[i].wt+totwt;

tottt=p[i].tt+tottt;

i++;

}

arg1=totwt/n;

arg2=tottt/n;

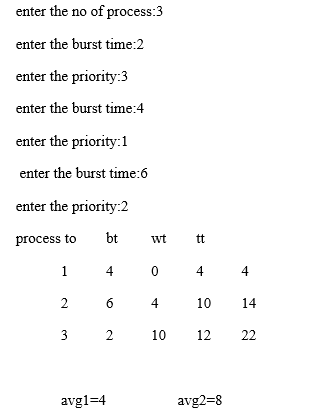
printf("\n arg1=%d \t arg2=%d\t",arg1,arg2);

getch();

return 0;

}

**Output:**



**9. Design, Develop and Implement File management system calls:**

**a). create a file**

**b). Copy one file to another**

**c). Linking a file**

**d). Delete a file.**

**Program:**

**Create a file**

#include <stdio.h>

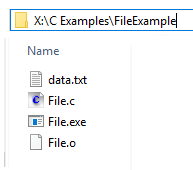
int main() {

FILE \*fp;

fp = fopen ("data.txt", "w");

}

**Output:**



**Copy one file to another**

#include <stdio.h>

#include <stdlib.h> // For exit()

int main()

{

    FILE \*fptr1, \*fptr2;

    char filename[100], c;

    printf("Enter the filename to open for reading \n");

    scanf("%s", filename);

    // Open one file for reading

    fptr1 = fopen(filename, "r");

    if (fptr1 == NULL)

    {

        printf("Cannot open file %s \n", filename);

        exit(0);

    }

    printf("Enter the filename to open for writing \n");

    scanf("%s", filename);

    // Open another file for writing

    fptr2 = fopen(filename, "w");

    if (fptr2 == NULL)

    {

        printf("Cannot open file %s \n", filename);

        exit(0);

    }

    // Read contents from file

    c = fgetc(fptr1);

    while (c != EOF)

    {

        fputc(c, fptr2);

        c = fgetc(fptr1);

    }

    printf("\nContents copied to %s", filename);

    fclose(fptr1);

    fclose(fptr2);

    return 0;

}

**Output:**

Enter the filename to open for reading

a.txt

Enter the filename to open for writing

b.txt

Contents copied to b.txt

**Delete a file**

#include<stdio.h>

int main()

{

   if (remove("abc.txt") == 0)

      printf("Deleted successfully");

   else

      printf("Unable to delete the file");

   return 0;

}

**Output:**



**10. Write a program that demonstrates how two processes can share a variable using semaphore**

**Program:**

#include<stdio.h>

void main()

{

int buffer[10], bufsize, in, out, produce, consume, choice=0;

in = 0;

out = 0;

bufsize = 10;

while(choice !=3)

{

printf("\n1. Produce \t 2. Consume \t3. Exit");

printf("\nEnter your choice: =");

scanf("%d", &choice);

switch(choice)

{

case 1: if((in+1)%bufsize==out)

printf("\nBuffer is Full");

else

{

printf("\nEnter the value: ");

scanf("%d", &produce);

buffer[in] = produce;

in = (in+1)%bufsize;

}

break;

case 2: if(in == out)

printf("\nBuffer is Empty");

else

{

consume = buffer[out];

printf("\nThe consumed value is %d", consume);

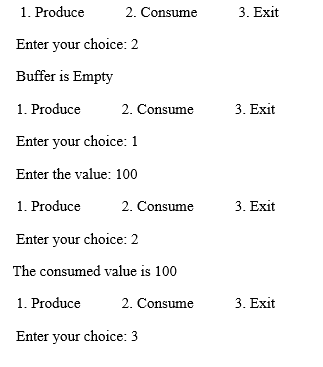
out = (out+1)%bufsize;

}

break;

} } }

**Output:**

****

**11. Design, Develop and Implement an Algorithm for Dead Lock Detection**

**Program:**

#include<stdio.h>

#include<conio.h>

int max[10][10], alloc[10][10], need[10][10];

int avail[10],i,j,p,r,finish[10]={0},flag=0;

int main()

{

clrscr( );

printf("\n\nSIMULATION OF DEADLOCK PREVENTION");

printf("Enter no. of processes, resources");

scanf("%d%d",&p,&r);printf("Enter allocation matrix");

for(i=0;i<p;i++)

for(j=0;j<r;j++)

scanf("%d",&alloc[i][j]);

printf("enter max matrix");

for(i=0;i<p;i++) /\*reading the maximum matrix and availale matrix\*/

for(j=0;j<r;j++)

scanf("%d",&max[i][j]);

printf("enter available matrix");

for(i=0;i<r;i++)

scanf("%d",&avail[i]);

for(i=0;i<p;i++)

for(j=0;j<r;j++)

need[i][j]=max[i][j]-alloc[i][j];

fun(); /\*calling function\*/

if(flag==0)

{

if(finish[i]!=1)

{

printf("\n\n Failing :Mutual exclusion");

for(j=0;j<r;j++)

{ /\*checking for mutual exclusion\*/

if(avail[j]<need[i][j])

avail[j]=need[i][j];

}fun();

printf("\n By allocating required resources to process %d dead lock is prevented ",i);

printf("\n\n lack of preemption");

for(j=0;j<r;j++)

{

if(avail[j]<need[i][j])

avail[j]=need[i][j];

alloc[i][j]=0;

}

fun( );

printf("\n\n daed lock is prevented by allocating needed resources");

printf(" \n \n failing:Hold and Wait condition ");

for(j=0;j<r;j++)

{

if(avail[j]<need[i][j])

avail[j]=need[i][j];

}

fun( );

printf("\n AVOIDING ANY ONE OF THE CONDITION, U CAN PREVENT DEADLOCK");

}

}

getch( ); return 0;

}

fun()

{

while(1)

{

for(flag=0,i=0;i<p;i++)

{

if(finish[i]==0)

{

for(j=0;j<r;j++)

{

if(need[i][j]<=avail[j])

continue;

else

break;

}

if(j==r)

{

for(j=0;j<r;j++)

avail[j]+=alloc[i][j];

flag=1;

finish[i]=1;

}

}

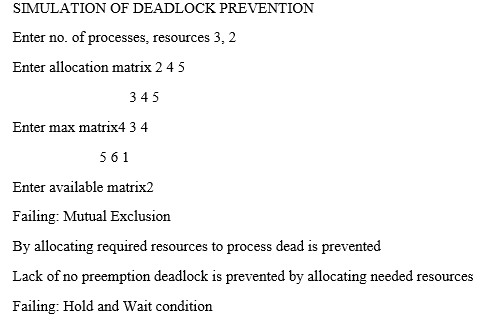
}

if(flag==0)

break;

}return 0;}

**Output:**



**12. Design, Develop and Implement a Program by using page replacement algorithms for virtual memory management**

**Program:**

#include<stdio.h>

#include<conio.h>

int i,j,nof,nor,flag=0,ref[50],frm[50],pf=0,victim=-1;

int recent[10],optcal[50],count=0;

int optvictim();

void main()

{

clrscr();

printf("\n OPTIMAL PAGE REPLACEMENT ALGORITHN");

printf("\n......................... ........");

printf("\nEnter the no.of frames");

scanf("%d",&nof);

printf("Enter the no.of reference string");

scanf("%d",&nor);

printf("Enter the reference string");

for(i=0;i<nor;i++)

scanf("%d",&ref[i]);

clrscr();

printf("\n OPTIMAL PAGE REPLACEMENT ALGORITHM");

printf("\n................................");

printf("\nThe given string");

printf("\n....................\n");

for(i=0;i<nor;i++)

printf("%4d",ref[i]);

for(i=0;i<nof;i++)

{

frm[i]=-1;

optcal[i]=0;

}

for(i=0;i<10;i++)

recent[i]=0;

printf("\n");

for(i=0;i<nor;i++)

{

flag=0;

printf("\n\tref no %d ->\t",ref[i]);

for(j=0;j<nof;j++)

{

if(frm[j]==ref[i])

{

flag=1;

break;

}

}

if(flag==0)

{

count++;

if(count<=nof)

victim++;

else

victim=optvictim(i);

pf++;

frm[victim]=ref[i];

for(j=0;j<nof;j++)

printf("%4d",frm[j]);

}

}

printf("\n Number of page faults: %d",pf);

getch();

}

int optvictim(int index)

{

int i,j,temp,notfound;

for(i=0;i<nof;i++)

{

notfound=1;

for(j=index;j<nor;j++)

if(frm[i]==ref[j])

{

notfound=0;

optcal[i]=j;

break;

}

if(notfound==1)

return i;

}

temp=optcal[0];

for(i=1;i<nof;i++)

if(temp<optcal[i])

temp=optcal[i];

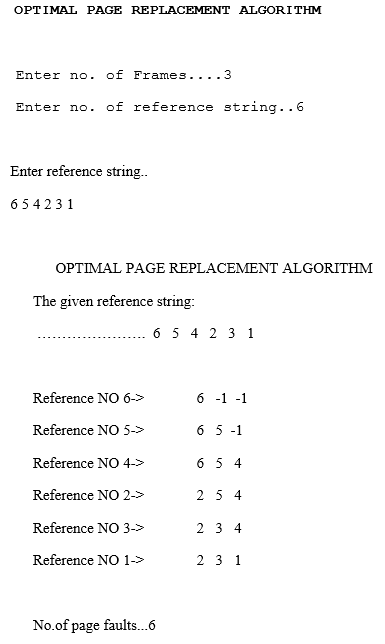
for(i=0;i<nof;i++)

if(frm[temp]==frm[i])

return i;

return 0;}

**Output:**

****