**LIST OF PROGRAMS**

**1.Create a new process by invoking the appropriate system call. Get the process identifier of the currently running process and its respective parent using system calls and display the same using a C program.**

#include<stdio.h>

#include<unistd.h>

int main()

{

printf("Process ID: %d\n", getpid() );

printf("Parent Process ID: %d\n", getpid() );

return 0;

}

**2. Identify the system calls to copy the content of one file to another and illustrate the same using a C program.**

#include <stdio.h>

#include <stdlib.h>

int main()

{

FILE \*fptr1, \*fptr2;

char filename[100], c;

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

scanf("%s", filename);

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);

fptr2 = fopen(filename, "w");

if (fptr2 == NULL)

{

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

exit(0);

}

c = fgetc(fptr1);

while (c != EOF)

{

fputc(c, fptr2);

c = fgetc(fptr1);

}

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

fclose(fptr1);

fclose(fptr2);

return 0;

}

**3. Design a CPU scheduling program with C using First Come First Served technique with the following considerations.**

**a. All processes are activated at time 0.**

**b. Assume that no process waits on I/O devices.**

#include <stdio.h>

int main()

{

int A[100][4];

int i, j, n, total = 0, index, temp;

float avg\_wt, avg\_tat;

printf("Enter number of process: ");

scanf("%d", &n);

printf("Enter Burst Time:\n");

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

printf("P%d: ", i + 1);

scanf("%d", &A[i][1]);

A[i][0] = i + 1;

}

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

index = i;

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

if (A[j][1] < A[index][1])

index = j;

temp = A[i][1];

A[i][1] = A[index][1];

A[index][1] = temp;

temp = A[i][0];

A[i][0] = A[index][0];

A[index][0] = temp;

}

A[0][2] = 0;

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

A[i][2] = 0;

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

A[i][2] += A[j][1];

total += A[i][2];

}

avg\_wt = (float)total / n;

total = 0;

printf("P BT WT TAT\n");

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

A[i][3] = A[i][1] + A[i][2];

total += A[i][3];

printf("P%d %d %d %d\n", A[i][0],A[i][1], A[i][2], A[i][3]);

}

avg\_tat = (float)total / n;

printf("Average Waiting Time= %f", avg\_wt);

printf("\nAverage Turnaround Time= %f", avg\_tat);

}

**4. Construct a scheduling program with C that selects the waiting process with the smallest execution time to execute next.**

#include<stdio.h>

int main()

{

int bt[20],p[20],wt[20],tat[20],i,j,n,total=0,pos,temp;

float avg\_wt,avg\_tat;

printf("Enter number of process:");

scanf("%d",&n);

printf("nEnter Burst Time:n");

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

{

printf("p%d:",i+1);

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

p[i]=i+1;

}

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

{

pos=i;

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

{

if(bt[j]<bt[pos])

pos=j;

}

temp=bt[i];

bt[i]=bt[pos];

bt[pos]=temp;

temp=p[i];

p[i]=p[pos];

p[pos]=temp;

}

wt[0]=0;

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

{

wt[i]=0;

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

wt[i]+=bt[j];

total+=wt[i];

}

avg\_wt=(float)total/n;

total=0;

printf("nProcesst Burst Time tWaiting TimetTurnaround Time");

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

{

tat[i]=bt[i]+wt[i];

total+=tat[i];

printf("np%dtt %dtt %dttt%d",p[i],bt[i],wt[i],tat[i]);

}

avg\_tat=(float)total/n;

printf("nnAverage Waiting Time=%f",avg\_wt);

printf("nAverage Turnaround Time=%fn",avg\_tat);

}

**5. Construct a scheduling program with C that selects the waiting process with the highest priority to execute next.**

#include<stdio.h>

struct priority\_scheduling {

char process\_name;

int burst\_time;

int waiting\_time;

int turn\_around\_time;

int priority;

};

int main() {

int number\_of\_process;

int total = 0;

struct priority\_scheduling temp\_process;

int ASCII\_number = 65;

int position;

float average\_waiting\_time;

float average\_turnaround\_time;

printf("Enter the total number of Processes: ");

scanf("%d", & number\_of\_process);

struct priority\_scheduling process[number\_of\_process];

printf("\nPlease Enter the Burst Time and Priority of each process:\n");

for (int i = 0; i < number\_of\_process; i++) {

process[i].process\_name = (char) ASCII\_number;

printf("\nEnter the details of the process %c \n", process[i].process\_name);

printf("Enter the burst time: ");

scanf("%d", & process[i].burst\_time);

printf("Enter the priority: ");

scanf("%d", & process[i].priority);

ASCII\_number++;

}

for (int i = 0; i < number\_of\_process; i++) {

position = i;

for (int j = i + 1; j < number\_of\_process; j++) {

if (process[j].priority > process[position].priority)

position = j;

}

temp\_process = process[i];

process[i] = process[position];

process[position] = temp\_process;

}

process[0].waiting\_time = 0;

for (int i = 1; i < number\_of\_process; i++) {

process[i].waiting\_time = 0;

for (int j = 0; j < i; j++) {

process[i].waiting\_time += process[j].burst\_time;

}

total += process[i].waiting\_time;

}

average\_waiting\_time = (float) total / (float) number\_of\_process;

total = 0;

printf("\n\nProcess\_name \t Burst Time \t Waiting Time \t Turnaround Time\n");

printf("------------------------------------------------------------\n");

for (int i = 0; i < number\_of\_process; i++) {

process[i].turn\_around\_time = process[i].burst\_time + process[i].waiting\_time;

total += process[i].turn\_around\_time;

printf("\t %c \t\t %d \t\t %d \t\t %d", process[i].process\_name, process[i].burst\_time, process[i].waiting\_time, process[i].turn\_around\_time);

printf("\n-----------------------------------------------------------\n");

}

average\_turnaround\_time = (float) total / (float) number\_of\_process;

printf("\n\n Average Waiting Time : %f", average\_waiting\_time);

printf("\n Average Turnaround Time: %f\n", average\_turnaround\_time);

return 0;

}

**6. Construct a C program to simulate Round Robin scheduling algorithm with C.**

#include<stdio.h>

#include<conio.h>

int main()

{

int i, NOP, sum=0,count=0, y, quant, wt=0, tat=0, at[10], bt[10], temp[10];

float avg\_wt, avg\_tat;

printf(" Total number of process in the system: ");

scanf("%d", &NOP);

y = NOP;

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

{

printf("\n Enter the Arrival and Burst time of the Process[%d]\n", i+1);

printf(" Arrival time is: \t");

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

printf(" \nBurst time is: \t");

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

temp[i] = bt[i];

}

printf("Enter the Time Quantum for the process: \t");

scanf("%d", &quant);

printf("\n Process No \t\t Burst Time \t\t TAT \t\t Waiting Time ");

for(sum=0, i = 0; y!=0; )

{

if(temp[i] <= quant && temp[i] > 0)

{

sum = sum + temp[i];

temp[i] = 0;

count=1;

}

else if(temp[i] > 0)

{

temp[i] = temp[i] - quant;

sum = sum + quant;

}

if(temp[i]==0 && count==1)

{

y--;

printf("\nProcess No[%d] \t\t %d\t\t\t\t %d\t\t\t %d", i+1, bt[i], sum-at[i], sum-at[i]-bt[i]);

wt = wt+sum-at[i]-bt[i];

tat = tat+sum-at[i];

count =0;

}

if(i==NOP-1)

{

i=0;

}

else if(at[i+1]<=sum)

{

i++;

}

else

{

i=0;

}

}

avg\_wt = wt \* 1.0/NOP;

avg\_tat = tat \* 1.0/NOP;

printf("\n Average Turn Around Time: \t%f", avg\_wt);

printf("\n Average Waiting Time: \t%f", avg\_tat);

getch();

}

**7. Illustrate the concept of inter-process communication using shared memory with a C program.**

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#include<sys/shm.h>

#include<string.h>

int main()

{

int i;

void \*shared\_memory;

char buff[100];

int shmid;

shmid=shmget((key\_t)2345, 1024, 0666|IPC\_CREAT);

printf("Key of shared memory is %d\n",shmid);

shared\_memory=shmat(shmid,NULL,0);

printf("Process attached at %p\n",shared\_memory);

printf("Enter some data to write to shared memory\n");

read(0,buff,100);

strcpy(shared\_memory,buff);

printf("You wrote : %s\n",(char \*)shared\_memory);

}

**8. Illustrate the concept of multithreading using a C program.**

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#include<pthread.h>

void \*myThreadFun(void \*vargp)

{

sleep(1);

printf("Printing GeeksQuiz from Thread \n");

return NULL;

}

int main()

{

pthread\_t thread\_id;

printf("Before Thread\n");

pthread\_create(&thread\_id, NULL, myThreadFun, NULL);

pthread\_join(thread\_id, NULL);

printf("After Thread\n");

exit(0);

}

**9. Design a C program to simulate the concept of Dining-Philosophers problem**

#include<stdio.h>

#include<stdlib.h>

#include<pthread.h>

#include<semaphore.h>

#include<unistd.h>

sem\_t room;

sem\_t chopstick[5];

void \* philosopher(void \*);

void eat(int);

int main()

{

int i,a[5];

pthread\_t tid[5];

sem\_init(&room,0,4);

for(i=0;i<5;i++)

sem\_init(&chopstick[i],0,1);

for(i=0;i<5;i++){

a[i]=i;

pthread\_create(&tid[i],NULL,philosopher,(void \*)&a[i]);

}

for(i=0;i<5;i++)

pthread\_join(tid[i],NULL);

}

void \* philosopher(void \* num)

{

int phil=\*(int \*)num;

sem\_wait(&room);

printf("\nPhilosopher %d has entered room",phil);

sem\_wait(&chopstick[phil]);

sem\_wait(&chopstick[(phil+1)%5]);

eat(phil);

sleep(2);

printf("\nPhilosopher %d has finished eating",phil);

sem\_post(&chopstick[(phil+1)%5]);

sem\_post(&chopstick[phil]);

sem\_post(&room);

}

void eat(int phil)

{

printf("\nPhilosopher %d is eating",phil);

}

**10. Construct a C program for implementation of memory allocation using first fit strategy.**

#include<stdio.h>

int main()

{

int bsize[10], psize[10], bno, pno, flags[10], allocation[10], i, j;

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

{

flags[i] = 0;

allocation[i] = -1;

}

printf("Enter no. of blocks: ");

scanf("%d", &bno);

printf("\nEnter size of each block: ");

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

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

printf("\nEnter no. of processes: ");

scanf("%d", &pno);

printf("\nEnter size of each process: ");

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

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

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

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

if(flags[j] == 0 && bsize[j] >= psize[i])

{

allocation[j] = i;

flags[j] = 1;

break;

}

printf("\nBlock no.\tsize\t\tprocess no.\t\tsize");

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

{

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

if(flags[i] == 1)

printf("%d\t\t\t%d",allocation[i]+1,psize[allocation[i]]);

else

printf("Not allocated");

}

}

**11. Construct a C program to organize the file using single level directory.**

#include<stdio.h>

#include<conio.h>

#include<string.h>

int main()

{

int nf=0,i=0,j=0,ch;

char mdname[10],fname[10][10],name[10];

printf("Enter the directory name:");

scanf("%s",mdname);

printf("Enter the number of files:");

scanf("%d",&nf);

do

{

printf("Enter file name to be created:");

scanf("%s",name);

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

{

if(!strcmp(name,fname[i]))

break;

}

if(i==nf)

{

strcpy(fname[j++],name);

nf++;

}

else

printf("There is already %s\n",name);

printf("Do you want to enter another file(yes - 1 or no - 0):");

scanf("%d",&ch);

}

while(ch==1);

printf("Directory name is:%s\n",mdname);

printf("Files names are:");

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

printf("\n%s",fname[i]);

getch();

}

**12. Design a C program to organize the file using two level directory structure.**

#include<stdio.h>

#include<conio.h>

struct st

{

char dname[10];

char sdname[10][10];

char fname[10][10][10];

int ds,sds[10];

}dir[10];

int main()

{

int i,j,k,n;

printf("enter number of directories:");

scanf("%d",&n);

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

{

printf("enter directory %d names:",i+1);

scanf("%s",&dir[i].dname);

printf("enter size of directories:");

scanf("%d",&dir[i].ds);

for(j=0;j<dir[i].ds;j++)

{

printf("enter subdirectory name and size:");

scanf("%s",&dir[i].sdname[j]);

scanf("%d",&dir[i].sds[j]);

for(k=0;k<dir[i].sds[j];k++)

{

printf("enter file name:");

scanf("%s",&dir[i].fname[j][k]);

}

}

}

printf("\ndirname\t\tsize\tsubdirname\tsize\tfiles");

printf("\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

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

{

printf("%s\t\t%d",dir[i].dname,dir[i].ds);

for(j=0;j<dir[i].ds;j++)

{

printf("\t%s\t\t%d\t",dir[i].sdname[j],dir[i].sds[j]);

for(k=0;k<dir[i].sds[j];k++)

printf("%s\t",dir[i].fname[j][k]);

printf("\n\t\t");

}

printf("\n");

}

getch();

}

**13. Develop a C program for implementing random access file for processing the employee details.**

**14. Illustrate the deadlock avoidance concept by simulating Banker’s algorithm with C.**

#include<stdio.h>

#include<conio.h>

int max[100][100];

int alloc[100][100];

int need[100][100];

int avail[100];

int n,r;

void input();

void show();

void cal();

int main()

{

int i,j;

printf("\*\*\*\*\*\*\*\*\*\* Banker's Algo \*\*\*\*\*\*\*\*\*\*\*\*\n");

input();

show();

cal();

getch();

return 0;

}

void input()

{

int i,j;

printf("Enter the no of Processes\t");

scanf("%d",&n);

printf("Enter the no of resources instances\t");

scanf("%d",&r);

printf("Enter the Max Matrix\n");

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

{

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

{

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

}

}

printf("Enter the Allocation Matrix\n");

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

{

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

{

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

}

}

printf("Enter the available Resources\n");

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

{

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

}

}

void show()

{

int i,j;

printf("Process\t Allocation\t Max\t Available\t");

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

{

printf("\nP%d\t ",i+1);

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

{

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

}

printf("\t");

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

{

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

}

printf("\t");

if(i==0)

{

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

printf("%d ",avail[j]);

}

}

}

void cal()

{

int finish[100],temp,need[100][100],flag=1,k,c1=0;

int safe[100];

int i,j;

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

{

finish[i]=0;

}

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

{

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

{

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

}

}

printf("\n");

while(flag)

{

flag=0;

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

{

int c=0;

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

{

if((finish[i]==0)&&(need[i][j]<=avail[j]))

{

c++;

if(c==r)

{

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

{

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

finish[i]=1;

flag=1;

}

printf("P%d->",i);

if(finish[i]==1)

{

i=n;

}

}

}

}

}

}

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

{

if(finish[i]==1)

{

c1++;

}

else

{

printf("P%d->",i);

}

}

if(c1==n)

{

printf("\n The system is in safe state");

}

else

{

printf("\n Process are in dead lock");

printf("\n System is in unsafe state");

}

}

**15 Construct a C program to simulate producer-consumer problem using semaphores.**

#include<stdio.h>

#include<stdlib.h>

int mutex=1,full=0,empty=3,x=0;

int main()

{

int n;

void producer();

void consumer();

int wait(int);

int signal(int);

printf("\n1.Producer\n2.Consumer\n3.Exit");

while(1)

{

printf("\nEnter your choice:");

scanf("%d",&n);

switch(n)

{

case 1: if((mutex==1)&&(empty!=0))

producer();

else

printf("Buffer is full!!");

break;

case 2: if((mutex==1)&&(full!=0))

consumer();

else

printf("Buffer is empty!!");

break;

case 3:

exit(0);

break;

}

}

return 0;

}

int wait(int s)

{

return (--s);

}

int signal(int s)

{

return(++s);

}

void producer()

{

mutex=wait(mutex);

full=signal(full);

empty=wait(empty);

x++;

printf("\nProducer produces the item %d",x);

mutex=signal(mutex);

}

void consumer()

{

mutex=wait(mutex);

full=wait(full);

empty=signal(empty);

printf("\nConsumer consumes item %d",x);

x--;

mutex=signal(mutex);

}

**16. Construct a C program to simulate the First in First Out paging technique of memory management.**

#include <stdio.h>

int main()

{

int incomingStream[] = {4, 1, 2, 4, 5};

int pageFaults = 0;

int frames = 3;

int m, n, s, pages;

pages = sizeof(incomingStream)/sizeof(incomingStream[0]);

printf("Incoming \t Frame 1 \t Frame 2 \t Frame 3");

int temp[frames];

for(m = 0; m < frames; m++)

{

temp[m] = -1;

}

for(m = 0; m < pages; m++)

{

s = 0;

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

{

if(incomingStream[m] == temp[n])

{

s++;

pageFaults--;

}

}

pageFaults++;

if((pageFaults <= frames) && (s == 0))

{

temp[m] = incomingStream[m];

}

else if(s == 0)

{

temp[(pageFaults - 1) % frames] = incomingStream[m];

}

printf("\n");

printf("%d\t\t\t",incomingStream[m]);

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

{

if(temp[n] != -1)

printf(" %d\t\t\t", temp[n]);

else

printf(" - \t\t\t");

}

}

printf("\nTotal Page Faults:\t%d\n", pageFaults);

return 0;

}

**17. Construct a C program to simulate the Least Recently Used paging technique of memory management.**

#include<stdio.h>

int findLRU(int time[], int n){

int i, minimum = time[0], pos = 0;

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

if(time[i] < minimum){

minimum = time[i];

pos = i;

}

}

return pos;

}

int main()

{

int no\_of\_frames, no\_of\_pages, frames[10], pages[30], counter = 0, time[10], flag1, flag2, i, j, pos, faults = 0;

printf("Enter number of frames: ");

scanf("%d", &no\_of\_frames);

printf("Enter number of pages: ");

scanf("%d", &no\_of\_pages);

printf("Enter reference string: ");

for(i = 0; i < no\_of\_pages; ++i){

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

}

for(i = 0; i < no\_of\_frames; ++i){

frames[i] = -1;

}

for(i = 0; i < no\_of\_pages; ++i){

flag1 = flag2 = 0;

for(j = 0; j < no\_of\_frames; ++j){

if(frames[j] == pages[i]){

counter++;

time[j] = counter;

flag1 = flag2 = 1;

break;

}

}

if(flag1 == 0){

for(j = 0; j < no\_of\_frames; ++j){

if(frames[j] == -1){

counter++;

faults++;

frames[j] = pages[i];

time[j] = counter;

flag2 = 1;

break;

}

}

}

if(flag2 == 0){

pos = findLRU(time, no\_of\_frames);

counter++;

faults++;

frames[pos] = pages[i];

time[pos] = counter;

}

printf("\n");

for(j = 0; j < no\_of\_frames; ++j){

printf("%d\t", frames[j]);

}

}

printf("\n\nTotal Page Faults = %d", faults);

return 0;

}

**18. Construct a C program to simulate the optimal paging technique of memory management**

#include<stdio.h>

int main()

{

int no\_of\_frames, no\_of\_pages, frames[10], pages[30], temp[10], flag1, flag2, flag3, i, j, k, pos, max, faults = 0;

printf("Enter number of frames: ");

scanf("%d", &no\_of\_frames);

printf("Enter number of pages: ");

scanf("%d", &no\_of\_pages);

printf("Enter page reference string: ");

for(i = 0; i < no\_of\_pages; ++i){

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

}

for(i = 0; i < no\_of\_frames; ++i){

frames[i] = -1;

}

for(i = 0; i < no\_of\_pages; ++i){

flag1 = flag2 = 0;

for(j = 0; j < no\_of\_frames; ++j){

if(frames[j] == pages[i]){

flag1 = flag2 = 1;

break;

}

}

if(flag1 == 0){

for(j = 0; j < no\_of\_frames; ++j){

if(frames[j] == -1){

faults++;

frames[j] = pages[i];

flag2 = 1;

break;

}

}

}

if(flag2 == 0){

flag3 =0;

for(j = 0; j < no\_of\_frames; ++j){

temp[j] = -1;

for(k = i + 1; k < no\_of\_pages; ++k){

if(frames[j] == pages[k]){

temp[j] = k;

break;

}

}

}

for(j = 0; j < no\_of\_frames; ++j){

if(temp[j] == -1){

pos = j;

flag3 = 1;

break;

}

}

if(flag3 ==0){

max = temp[0];

pos = 0;

for(j = 1; j < no\_of\_frames; ++j){

if(temp[j] > max){

max = temp[j];

pos = j;

}

}

}

frames[pos] = pages[i];

faults++;

}

printf("\n");

for(j = 0; j < no\_of\_frames; ++j){

printf("%d\t", frames[j]);

}

}

printf("\n\nTotal Page Faults = %d", faults);

return 0;

}

**19. Consider a file system where the records of the file are stored one after another both physically and logically. A record of the file can only be accessed by reading all the previous records.  Design a C program to simulate the file allocation strategy.**

#include<stdio.h>

#include<conio.h>

#include<stdlib.h>

int main()

{

int f[50], i, st, len, j, c, k, count = 0;

for(i=0;i<50;i++)

f[i]=0;

printf("Files Allocated are : \n");

x : count=0;

printf("Enter starting block and length of files: ");

scanf("%d%d", &st,&len);

for(k=st;k<(st+len);k++)

if(f[k]==0)

count++;

if(len==count)

{

for(j=st;j<(st+len);j++)

if(f[j]==0)

{

f[j]=1;

printf("%d\t%d\n",j,f[j]);

}

if(j!=(st+len-1))

printf("The file is allocated to disk\n");

}

else

printf("The file is not allocated \n");

printf("Do you want to enter more file(Yes - 1/No - 0)");

scanf("%d", &c);

if(c==1)

goto x;

else

exit(0);

getch();

}

**20. Consider a file system that brings all the file pointers together into an index block. The ith entry in the index block points to the ith block of the file. Design a C program to simulate the file allocation strategy.**

#include<stdio.h>

#include<conio.h>

#include<stdlib.h>

int main()

{

int f[50], index[50],i, n, st, len, j, c, k, ind,count=0;

for(i=0;i<50;i++)

f[i]=0;

x:printf("Enter the index block: ");

scanf("%d",&ind);

if(f[ind]!=1)

{

printf("Enter no of blocks needed and no of files for the index %d on the disk : \n", ind);

scanf("%d",&n);

}

else

{

printf("%d index is already allocated \n",ind);

goto x;

}

y: count=0;

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

{

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

if(f[index[i]]==0)

count++;

}

if(count==n)

{

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

f[index[j]]=1;

printf("Allocated\n");

printf("File Indexed\n");

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

printf("%d-------->%d : %d\n",ind,index[k],f[index[k]]);

}

else

{

printf("File in the index is already allocated \n");

printf("Enter another file indexed");

goto y;

}

printf("Do you want to enter more file(Yes - 1/No - 0)");

scanf("%d", &c);

if(c==1)

goto x;

else

exit(0);

getch();

}

**21. With linked allocation, each file is a linked list of disk blocks; the disk blocks may be scattered anywhere on the disk. The directory contains a pointer to the first and last blocks of the file.  Each block contains a pointer to the next block. Design a C program to simulate the file allocation strategy.**

**#include<stdio.h>**

#include<conio.h>

#include<stdlib.h>

void main()

{

int f[50], p,i, st, len, j, c, k, a;

clrscr();

for(i=0;i<50;i++)

f[i]=0;

printf("Enter how many blocks already allocated: ");

scanf("%d",&p);

printf("Enter blocks already allocated: ");

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

{

scanf("%d",&a);

f[a]=1;

}

x: printf("Enter index starting block and length: ");

scanf("%d%d", &st,&len);

k=len;

if(f[st]==0)

{

for(j=st;j<(st+k);j++)

{

if(f[j]==0)

{

f[j]=1;

printf("%d-------->%d\n",j,f[j]);

}

else

{

printf("%d Block is already allocated \n",j);

k++;

}

}

}

else

printf("%d starting block is already allocated \n",st);

printf("Do you want to enter more file(Yes - 1/No - 0)");

scanf("%d", &c);

if(c==1)

goto x;

else

exit(0);

getch();

}

**22. Construct a C program to simulate the First Come First Served disk scheduling algorithm.**

#include<stdio.h>

#include<stdlib.h>

int main()

{

int ReadyQueue[100],i,n,TotalHeadMov=0,initial;

scanf("%d",&n);

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

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

}

scanf("%d",&initial);

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

{

TotalHeadMov=TotalHeadMov+abs(ReadyQueue[i]-initial);

initial=ReadyQueue[i];

}

printf("Total Head Movement=%d",TotalHeadMov);

}

**23. Design a C program to simulate SCAN disk scheduling algorithm.**

**24.. Develop a C program to simulate C-SCAN disk scheduling algorithm.**

**25. Illustrate the various File Access Permission and different types users in Linux.**

**Banker's algorithm**

**#include<stdio.h>**

**#include<conio.h>**

**int max[100][100];**

**int alloc[100][100];**

**int need[100][100];**

**int avail[100];**

**int n,r;**

**void input();**

**void show();**

**void cal();**

**int main()**

**{**

**int i,j;**

**printf("\*\*\*\*\*\*\*\*\*\* Banker's Algo \*\*\*\*\*\*\*\*\*\*\*\*\n");**

**input();**

**show();**

**cal();**

**getch();**

**return 0;**

**}**

**void input()**

**{**

**int i,j;**

**printf("Enter the no of Processes\t");**

**scanf("%d",&n);**

**printf("Enter the no of resources instances\t");**

**scanf("%d",&r);**

**printf("Enter the Max Matrix\n");**

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

**{**

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

**{**

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

**}**

**}**

**printf("Enter the Allocation Matrix\n");**

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

**{**

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

**{**

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

**}**

**}**

**printf("Enter the available Resources\n");**

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

**{**

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

**}**

**}**

**void show()**

**{**

**int i,j;**

**printf("Process\t Allocation\t Max\t Available\t");**

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

**{**

**printf("\nP%d\t ",i+1);**

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

**{**

**printf("%d ",alloc[i][j]);**

**}**

**printf("\t");**

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

**{**

**printf("%d ",max[i][j]);**

**}**

**printf("\t");**

**if(i==0)**

**{**

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

**printf("%d ",avail[j]);**

**}**

**}**

**}**

**void cal()**

**{**

**int finish[100],temp,need[100][100],flag=1,k,c1=0;**

**int safe[100];**

**int i,j;**

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

**{**

**finish[i]=0;**

**}**

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

**{**

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

**{**

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

**}**

**}**

**printf("\n");**

**while(flag)**

**{**

**flag=0;**

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

**{**

**int c=0;**

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

**{**

**if((finish[i]==0)&&(need[i][j]<=avail[j]))**

**{**

**c++;**

**if(c==r)**

**{**

**for(k=0;k<r;k++)**

**{**

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

**finish[i]=1;**

**flag=1;**

**}**

**printf("P%d->",i);**

**if(finish[i]==1)**

**{**

**i=n;**

**}**

**}**

**}**

**}**

**}**

**}**

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

**{**

**if(finish[i]==1)**

**{**

**c1++;**

**}**

**else**

**{**

**printf("P%d->",i);**

**}**

**}**

**if(c1==n)**

**{**

**printf("\n The system is in safe state");**

**}**

**else**

**{**

**printf("\n Process are in dead lock");**

**printf("\n System is in unsafe state");**

**}**

}

**Best Fit Algorithm**

#include <stdio.h>

#define MAX\_BLOCKS 100

#define MAX\_PROCESS 100

void bestFit(int blockSize[], int m, int processSize[], int n) {

int allocation[MAX\_PROCESS];

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

allocation[i] = -1; // Initialize allocation array to -1 (indicating not allocated)

}

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

int bestIndex = -1;

for (int j = 0; j < m; j++) {

if (blockSize[j] >= processSize[i]) {

if (bestIndex == -1 || blockSize[j] < blockSize[bestIndex]) {

bestIndex = j;

}

}

}

if (bestIndex != -1) {

allocation[i] = bestIndex;

blockSize[bestIndex] -= processSize[i];

}

}

// Display the allocation result

printf("Process No.\tProcess Size\tBlock No.\n");

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

printf("%d\t\t%d\t\t", i + 1, processSize[i]);

if (allocation[i] != -1) {

printf("%d\n", allocation[i] + 1);

} else {

printf("Not Allocated\n");

}

}

}

int main() {

int m; // Number of memory blocks

int n; // Number of processes

printf("Enter the number of memory blocks: ");

scanf("%d", &m);

int blockSize[MAX\_BLOCKS];

printf("Enter the size of each memory block:\n");

for (int i = 0; i < m; i++) {

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

}

printf("Enter the number of processes: ");

scanf("%d", &n);

int processSize[MAX\_PROCESS];

printf("Enter the size of each process:\n");

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

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

}

bestFit(blockSize, m, processSize, n);

return 0;

}

1. **Scan DS algorithm**

#include <stdio.h>

#include <stdlib.h>

void sort(int arr[], int n) {

int i, j, temp;

for (i = 0; i < n - 1; i++) {

for (j = 0; j < n - i - 1; j++) {

if (arr[j] > arr[j + 1]) {

// Swap the elements if they are in the wrong order

temp = arr[j];

arr[j] = arr[j + 1];

arr[j + 1] = temp;

}

}

}

}

int main() {

int diskSize, currentPosition, totalRequests;

printf("Enter the size of the disk: ");

scanf("%d", &diskSize);

printf("Enter the current position of the disk arm: ");

scanf("%d", &currentPosition);

printf("Enter the total number of disk requests: ");

scanf("%d", &totalRequests);

int requests[totalRequests];

int i;

printf("Enter the disk request track numbers:\n");

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

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

}

// Sort the disk requests in ascending order

sort(requests, totalRequests);

int seekCount = 0;

// C-SCAN towards the right

for (i = currentPosition; i <= diskSize; i++) {

printf("Servicing track %d\n", i);

seekCount += abs(currentPosition - i);

currentPosition = i;

}

// Jump to the beginning of the disk

printf("Jump to the beginning of the disk\n");

seekCount += currentPosition;

currentPosition = 0;

// C-SCAN towards the right again

for (i = currentPosition; i <= diskSize; i++) {

printf("Servicing track %d\n", i);

seekCount += abs(currentPosition - i);

currentPosition = i;

}

printf("Total seek count: %d\n", seekCount);

return 0;

}

**CPU Scheduling**

#include <stdio.h>

int main()

{

int A[100][4];

int i, j, n, total = 0, index, temp;

float avg\_wt, avg\_tat;

printf("Enter number of process: ");

scanf("%d", &n);

printf("Enter Burst Time:\n");

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

printf("P%d: ", i + 1);

scanf("%d", &A[i][1]);

A[i][0] = i + 1;

}

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

index = i;

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

if (A[j][1] < A[index][1])

index = j;

temp = A[i][1];

A[i][1] = A[index][1];

A[index][1] = temp;

temp = A[i][0];

A[i][0] = A[index][0];

A[index][0] = temp;

}

A[0][2] = 0;

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

A[i][2] = 0;

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

A[i][2] += A[j][1];

total += A[i][2];

}

avg\_wt = (float)total / n;

total = 0;

printf("P BT WT TAT\n");

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

A[i][3] = A[i][1] + A[i][2];

total += A[i][3];

printf("P%d %d %d %d\n", A[i][0],A[i][1], A[i][2], A[i][3]);

}

avg\_tat = (float)total / n;

printf("Average Waiting Time= %f", avg\_wt);

printf("\nAverage Turnaround Time= %f", avg\_tat);

}

**Dining-Philosophers problem**

#include<stdio.h>

#include<stdlib.h>

#include<pthread.h>

#include<semaphore.h>

#include<unistd.h>

sem\_t room;

sem\_t chopstick[5];

void \* philosopher(void \*);

void eat(int);

int main()

{

int i,a[5];

pthread\_t tid[5];

sem\_init(&room,0,4);

for(i=0;i<5;i++)

sem\_init(&chopstick[i],0,1);

for(i=0;i<5;i++){

a[i]=i;

pthread\_create(&tid[i],NULL,philosopher,(void \*)&a[i]);

}

for(i=0;i<5;i++)

pthread\_join(tid[i],NULL);

}

void \* philosopher(void \* num)

{

int phil=\*(int \*)num;

sem\_wait(&room);

printf("\nPhilosopher %d has entered room",phil);

sem\_wait(&chopstick[phil]);

sem\_wait(&chopstick[(phil+1)%5]);

eat(phil);

sleep(2);

printf("\nPhilosopher %d has finished eating",phil);

sem\_post(&chopstick[(phil+1)%5]);

sem\_post(&chopstick[phil]);

sem\_post(&room);

}

void eat(int phil)

{

printf("\nPhilosopher %d is eating",phil);

}

**FCFS Disk Scheduling Algorithm**

#include<stdio.h>

#include<stdlib.h>

int main()

{

int ReadyQueue[100],i,n,TotalHeadMov=0,initial;

scanf("%d",&n);

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

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

}

scanf("%d",&initial);

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

{

TotalHeadMov=TotalHeadMov+abs(ReadyQueue[i]-initial);

initial=ReadyQueue[i];

}

printf("Total Head Movement=%d",TotalHeadMov);

}

**FIFO Paging Technique**

#include <stdio.h>

#include <stdlib.h>

#define PAGE\_FRAMES 3

#define PAGE\_REFERENCE\_STRING\_SIZE 10

void displayPageTable(int pageTable[], int frames) {

printf("Page Table: ");

for (int i = 0; i < frames; i++) {

if (pageTable[i] == -1) {

printf("[ ] ");

} else {

printf("[%d] ", pageTable[i]);

}

}

printf("\n");

}

int isPageInMemory(int page, int pageTable[], int frames) {

for (int i = 0; i < frames; i++) {

if (pageTable[i] == page) {

return 1; // Page found in memory

}

}

return 0; // Page not found in memory

}

int findFIFOPage(int pageTable[], int frames) {

return 0; // FIFO replaces the page at the front of the queue

}

void simulateFIFO(int pageReferenceString[], int frames, int references) {

int pageTable[frames];

int pageQueue[frames];

int pageFaults = 0;

int queueIndex = 0;

for (int i = 0; i < frames; i++) {

pageTable[i] = -1; // Initialize page table with -1 (indicating an empty frame)

pageQueue[i] = -1; // Initialize page queue with -1 (indicating an empty slot)

}

printf("Simulating FIFO Paging:\n");

for (int i = 0; i < references; i++) {

int page = pageReferenceString[i];

if (!isPageInMemory(page, pageTable, frames)) {

pageFaults++;

int replaceIndex = findFIFOPage(pageTable, frames);

// Update the page table and queue

pageTable[replaceIndex] = page;

pageQueue[queueIndex] = replaceIndex;

queueIndex = (queueIndex + 1) % frames;

displayPageTable(pageTable, frames);

}

}

printf("Total Page Faults: %d\n", pageFaults);

}

int main() {

int pageReferenceString[PAGE\_REFERENCE\_STRING\_SIZE] = {1, 2, 3, 4, 1, 2, 5, 1, 2, 3};

int references = PAGE\_REFERENCE\_STRING\_SIZE;

simulateFIFO(pageReferenceString, PAGE\_FRAMES, references);

return 0;

}

**File allocation stratergy**

#include<stdio.h>

#include<conio.h>

#include<stdlib.h>

int main()

{

int f[50], i, st, len, j, c, k, count = 0;

for(i=0;i<50;i++)

f[i]=0;

printf("Files Allocated are : \n");

x : count=0;

printf("Enter starting block and length of files: ");

scanf("%d%d", &st,&len);

for(k=st;k<(st+len);k++)

if(f[k]==0)

count++;

if(len==count)

{

for(j=st;j<(st+len);j++)

if(f[j]==0)

{

f[j]=1;

printf("%d\t%d\n",j,f[j]);

}

if(j!=(st+len-1))

printf("The file is allocated to disk\n");

}

else

printf("The file is not allocated \n");

printf("Do you want to enter more file(Yes - 1/No - 0)");

scanf("%d", &c);

if(c==1)

goto x;

else

exit(0);

getch();

}

**File management Operations**

#include <stdio.h>

#include <stdlib.h>

int main() {

FILE \*file;

char fileName[] = "example.txt";

// Open file for writing (creates a new file or truncates an existing file)

file = fopen(fileName, "w");

if (file == NULL) {

perror("Error opening file for writing");

exit(EXIT\_FAILURE);

}

// Write data to the file

fprintf(file, "Hello, File Management!\n");

// Close the file

fclose(file);

// Open file for reading

file = fopen(fileName, "r");

if (file == NULL) {

perror("Error opening file for reading");

exit(EXIT\_FAILURE);

}

// Read and print data from the file

char buffer[256];

while (fgets(buffer, sizeof(buffer), file) != NULL) {

printf("%s", buffer);

}

// Close the file

fclose(file);

return 0;

}

**File pointers together an index block**

#include<stdio.h>

#include<conio.h>

#include<stdlib.h>

int main()

{

int f[50], index[50],i, n, st, len, j, c, k, ind,count=0;

for(i=0;i<50;i++)

f[i]=0;

x:printf("Enter the index block: ");

scanf("%d",&ind);

if(f[ind]!=1)

{

printf("Enter no of blocks needed and no of files for the index %d on the disk : \n", ind);

scanf("%d",&n);

}

else

{

printf("%d index is already allocated \n",ind);

goto x;

}

y: count=0;

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

{

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

if(f[index[i]]==0)

count++;

}

if(count==n)

{

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

f[index[j]]=1;

printf("Allocated\n");

printf("File Indexed\n");

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

printf("%d-------->%d : %d\n",ind,index[k],f[index[k]]);

}

else

{

printf("File in the index is already allocated \n");

printf("Enter another file indexed");

goto y;

}

printf("Do you want to enter more file(Yes - 1/No - 0)");

scanf("%d", &c);

if(c==1)

goto x;

else

exit(0);

getch();

}

**First in First out**

#include <stdio.h>

int main()

{

int incomingStream[] = {4, 1, 2, 4, 5};

int pageFaults = 0;

int frames = 3;

int m, n, s, pages;

pages = sizeof(incomingStream)/sizeof(incomingStream[0]);

printf("Incoming \t Frame 1 \t Frame 2 \t Frame 3");

int temp[frames];

for(m = 0; m < frames; m++)

{

temp[m] = -1;

}

for(m = 0; m < pages; m++)

{

s = 0;

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

{

if(incomingStream[m] == temp[n])

{

s++;

pageFaults--;

}

}

pageFaults++;

if((pageFaults <= frames) && (s == 0))

{

temp[m] = incomingStream[m];

}

else if(s == 0)

{

temp[(pageFaults - 1) % frames] = incomingStream[m];

}

printf("\n");

printf("%d\t\t\t",incomingStream[m]);

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

{

if(temp[n] != -1)

printf(" %d\t\t\t", temp[n]);

else

printf(" - \t\t\t");

}

}

printf("\nTotal Page Faults:\t%d\n", pageFaults);

return 0;

}

**GREP Unix Command**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define MAX\_LINE\_LENGTH 1024

void grepSimulation(FILE \*file, const char \*pattern) {

char line[MAX\_LINE\_LENGTH];

while (fgets(line, MAX\_LINE\_LENGTH, file) != NULL) {

if (strstr(line, pattern) != NULL) {

// Pattern found in the line, print the line

printf("%s", line);

}

}

}

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

if (argc < 2) {

fprintf(stderr, "Usage: %s <pattern> [file]\n", argv[0]);

exit(EXIT\_FAILURE);

}

const char \*pattern = argv[1];

FILE \*file;

if (argc == 2) {

// No file specified, read from standard input

grepSimulation(stdin, pattern);

} else {

// Read from the specified file

file = fopen(argv[2], "r");

if (file == NULL) {

perror("Error opening file");

exit(EXIT\_FAILURE);

}

grepSimulation(file, pattern);

// Close the file

fclose(file);

}

return 0;

}

**I&O System calls of unix**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <fcntl.h>

#include <sys/types.h>

#include <sys/stat.h>

int main() {

// Open a source file for reading

int source\_fd = open("source.txt", O\_RDONLY);

if (source\_fd == -1) {

perror("Failed to open source.txt");

exit(1);

}

// Create or open a destination file for writing

int dest\_fd = open("destination.txt", O\_WRONLY | O\_CREAT | O\_TRUNC, 0644);

if (dest\_fd == -1) {

perror("Failed to open destination.txt");

close(source\_fd); // Close the source file

exit(1);

}

// Read from the source file and write to the destination file

char buffer[4096]; // A buffer to hold data

ssize\_t nread;

while ((nread = read(source\_fd, buffer, sizeof(buffer))) > 0) {

if (write(dest\_fd, buffer, nread) != nread) {

perror("Write error");

break;

}

}

// Check if there was an error during reading

if (nread < 0) {

perror("Read error");

}

// Close both files

close(source\_fd);

close(dest\_fd);

return 0;

}

**LRU Paging Technique**

#include <stdio.h>

#include <stdlib.h>

#define PAGE\_FRAMES 3

#define PAGE\_REFERENCE\_STRING\_SIZE 10

void displayPageTable(int pageTable[], int frames) {

printf("Page Table: ");

for (int i = 0; i < frames; i++) {

if (pageTable[i] == -1) {

printf("[ ] ");

} else {

printf("[%d] ", pageTable[i]);

}

}

printf("\n");

}

int isPageInMemory(int page, int pageTable[], int frames) {

for (int i = 0; i < frames; i++) {

if (pageTable[i] == page) {

return 1; // Page found in memory

}

}

return 0; // Page not found in memory

}

int findLRUPage(int pageAccessOrder[], int frames) {

int minIndex = 0;

for (int i = 1; i < frames; i++) {

if (pageAccessOrder[i] < pageAccessOrder[minIndex]) {

minIndex = i;

}

}

return minIndex;

}

void simulateLRU(int pageReferenceString[], int frames, int references) {

int pageTable[frames];

int pageAccessOrder[frames];

int pageFaults = 0;

for (int i = 0; i < frames; i++) {

pageTable[i] = -1; // Initialize page table with -1 (indicating an empty frame)

pageAccessOrder[i] = 0; // Initialize page access order

}

printf("Simulating LRU Paging:\n");

for (int i = 0; i < references; i++) {

int page = pageReferenceString[i];

if (!isPageInMemory(page, pageTable, frames)) {

pageFaults++;

int replaceIndex = findLRUPage(pageAccessOrder, frames);

pageTable[replaceIndex] = page;

pageAccessOrder[replaceIndex] = i + 1; // Update page access order

displayPageTable(pageTable, frames);

} else {

// Page hit, update access order

for (int j = 0; j < frames; j++) {

if (pageTable[j] == page) {

pageAccessOrder[j] = i + 1; // Update page access order

break;

}

}

}

}

printf("Total Page Faults: %d\n", pageFaults);

}

int main() {

int pageReferenceString[PAGE\_REFERENCE\_STRING\_SIZE] = {1, 2, 3, 4, 1, 2, 5, 1, 2, 3};

int references = PAGE\_REFERENCE\_STRING\_SIZE;

simulateLRU(pageReferenceString, PAGE\_FRAMES, references);

return 0;

}

**Least Recently Used**

#include<stdio.h>

int findLRU(int time[], int n){

int i, minimum = time[0], pos = 0;

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

if(time[i] < minimum){

minimum = time[i];

pos = i;

}

}

return pos;

}

int main()

{

int no\_of\_frames, no\_of\_pages, frames[10], pages[30], counter = 0, time[10], flag1, flag2, i, j, pos, faults = 0;

printf("Enter number of frames: ");

scanf("%d", &no\_of\_frames);

printf("Enter number of pages: ");

scanf("%d", &no\_of\_pages);

printf("Enter reference string: ");

for(i = 0; i < no\_of\_pages; ++i){

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

}

for(i = 0; i < no\_of\_frames; ++i){

frames[i] = -1;

}

for(i = 0; i < no\_of\_pages; ++i){

flag1 = flag2 = 0;

for(j = 0; j < no\_of\_frames; ++j){

if(frames[j] == pages[i]){

counter++;

time[j] = counter;

flag1 = flag2 = 1;

break;

}

}

if(flag1 == 0){

for(j = 0; j < no\_of\_frames; ++j){

if(frames[j] == -1){

counter++;

faults++;

frames[j] = pages[i];

time[j] = counter;

flag2 = 1;

break;

}

}

}

if(flag2 == 0){

pos = findLRU(time, no\_of\_frames);

counter++;

faults++;

frames[pos] = pages[i];

time[pos] = counter;

}

printf("\n");

for(j = 0; j < no\_of\_frames; ++j){

printf("%d\t", frames[j]);

}

}

printf("\n\nTotal Page Faults = %d", faults);

return 0;

}

**Linked Allocation**

#include<stdio.h>

#include<conio.h>

#include<stdlib.h>

int main()

{

int f[50], p,i, st, len, j, c, k, a;

for(i=0;i<50;i++)

f[i]=0;

printf("Enter how many blocks already allocated: ");

scanf("%d",&p);

printf("Enter blocks already allocated: ");

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

{

scanf("%d",&a);

f[a]=1;

}

x: printf("Enter index starting block and length: ");

scanf("%d%d", &st,&len);

k=len;

if(f[st]==0)

{

for(j=st;j<(st+k);j++)

{

if(f[j]==0)

{

f[j]=1;

printf("%d-------->%d\n",j,f[j]);

}

else

{

printf("%d Block is already allocated \n",j);

k++;

}

}

}

else

printf("%d starting block is already allocated \n",st);

printf("Do you want to enter more file(Yes - 1/No - 0)");

scanf("%d", &c);

if(c==1)

goto x;

else

exit(0);

getch();

}

**Non-preemtive SJF**

#include <stdio.h>

#include <stdlib.h>

// Process structure to store process details

struct Process {

int processID;

int arrivalTime;

int burstTime;

int waitingTime;

int turnaroundTime;

};

// Function to perform SJF scheduling

void sjfScheduling(struct Process processes[], int n) {

int currentTime = 0; // Current time

int totalWaitingTime = 0; // Total waiting time for all processes

int totalTurnaroundTime = 0; // Total turnaround time for all processes

// Sort processes based on arrival time

for (int i = 0; i < n - 1; i++) {

for (int j = 0; j < n - i - 1; j++) {

if (processes[j].arrivalTime > processes[j + 1].arrivalTime) {

// Swap processes if not in the correct order

struct Process temp = processes[j];

processes[j] = processes[j + 1];

processes[j + 1] = temp;

}

}

}

// Perform SJF scheduling

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

// Wait until the process arrives

while (currentTime < processes[i].arrivalTime) {

currentTime++;

}

// Update waiting time for the current process

processes[i].waitingTime = currentTime;

// Update turnaround time for the current process

processes[i].turnaroundTime = processes[i].waitingTime + processes[i].burstTime;

// Update total waiting and turnaround times

totalWaitingTime += processes[i].waitingTime;

totalTurnaroundTime += processes[i].turnaroundTime;

// Move to the next process

currentTime += processes[i].burstTime;

}

// Display results

printf("Process\tArrival Time\tBurst Time\tWaiting Time\tTurnaround Time\n");

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

printf("%d\t%d\t\t%d\t\t%d\t\t%d\n", processes[i].processID, processes[i].arrivalTime,

processes[i].burstTime, processes[i].waitingTime, processes[i].turnaroundTime);

}

// Calculate and display average waiting and turnaround times

float avgWaitingTime = (float)totalWaitingTime / n;

float avgTurnaroundTime = (float)totalTurnaroundTime / n;

printf("Average Waiting Time: %.2f\n", avgWaitingTime);

printf("Average Turnaround Time: %.2f\n", avgTurnaroundTime);

}

int main() {

// Number of processes

int n;

printf("Enter the number of processes: ");

scanf("%d", &n);

struct Process processes[n];

// Input process details

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

processes[i].processID = i + 1;

printf("Enter arrival time for process %d: ", i + 1);

scanf("%d", &processes[i].arrivalTime);

printf("Enter burst time for process %d: ", i + 1);

scanf("%d", &processes[i].burstTime);

}

// Perform SJF scheduling

sjfScheduling(processes, n);

return 0;

}

**Optimal paging technique**

#include<stdio.h>

int main()

{

int no\_of\_frames, no\_of\_pages, frames[10], pages[30], temp[10], flag1, flag2, flag3, i, j, k, pos, max, faults = 0;

printf("Enter number of frames: ");

scanf("%d", &no\_of\_frames);

printf("Enter number of pages: ");

scanf("%d", &no\_of\_pages);

printf("Enter page reference string: ");

for(i = 0; i < no\_of\_pages; ++i){

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

}

for(i = 0; i < no\_of\_frames; ++i){

frames[i] = -1;

}

for(i = 0; i < no\_of\_pages; ++i){

flag1 = flag2 = 0;

for(j = 0; j < no\_of\_frames; ++j){

if(frames[j] == pages[i]){

flag1 = flag2 = 1;

break;

}

}

if(flag1 == 0){

for(j = 0; j < no\_of\_frames; ++j){

if(frames[j] == -1){

faults++;

frames[j] = pages[i];

flag2 = 1;

break;

}

}

}

if(flag2 == 0){

flag3 =0;

for(j = 0; j < no\_of\_frames; ++j){

temp[j] = -1;

for(k = i + 1; k < no\_of\_pages; ++k){

if(frames[j] == pages[k]){

temp[j] = k;

break;

}

}

}

for(j = 0; j < no\_of\_frames; ++j){

if(temp[j] == -1){

pos = j;

flag3 = 1;

break;

}

}

if(flag3 ==0){

max = temp[0];

pos = 0;

for(j = 1; j < no\_of\_frames; ++j){

if(temp[j] > max){

max = temp[j];

pos = j;

}

}

}

frames[pos] = pages[i];

faults++;

}

printf("\n");

for(j = 0; j < no\_of\_frames; ++j){

printf("%d\t", frames[j]);

}

}

printf("\n\nTotal Page Faults = %d", faults);

return 0;

}

**Producer consumer problem**

#include <stdio.h>

#include <stdlib.h>

struct Employee {

int empId;

char empName[50];

float empSalary;

};

void addRecord(FILE \*file) {

struct Employee emp;

printf("Enter Employee ID: ");

scanf("%d", &emp.empId);

printf("Enter Employee Name: ");

scanf("%s", emp.empName);

printf("Enter Employee Salary: ");

scanf("%f", &emp.empSalary);

fseek(file, (emp.empId - 1) \* sizeof(struct Employee), SEEK\_SET);

fwrite(&emp, sizeof(struct Employee), 1, file);

printf("Record added successfully!\n");

}

void displayAllRecords(FILE \*file) {

struct Employee emp;

rewind(file);

printf("\nEmployee Records:\n");

printf("ID\tName\tSalary\n");

while (fread(&emp, sizeof(struct Employee), 1, file) == 1) {

printf("%d\t%s\t%.2f\n", emp.empId, emp.empName, emp.empSalary);

}

}

void updateRecord(FILE \*file) {

struct Employee emp;

int empId;

printf("Enter Employee ID to update: ");

scanf("%d", &empId);

fseek(file, (empId - 1) \* sizeof(struct Employee), SEEK\_SET);

if (fread(&emp, sizeof(struct Employee), 1, file) != 1) {

printf("Employee not found!\n");

return;

}

printf("Enter new Employee Name: ");

scanf("%s", emp.empName);

printf("Enter new Employee Salary: ");

scanf("%f", &emp.empSalary);

fseek(file, -sizeof(struct Employee), SEEK\_CUR);

fwrite(&emp, sizeof(struct Employee), 1, file);

printf("Record updated successfully!\n");

}

void deleteRecord(FILE \*file) {

int empId;

struct Employee emp;

FILE \*tempFile;

printf("Enter Employee ID to delete: ");

scanf("%d", &empId);

tempFile = fopen("temp.dat", "wb");

rewind(file);

while (fread(&emp, sizeof(struct Employee), 1, file) == 1) {

if (emp.empId != empId) {

fwrite(&emp, sizeof(struct Employee), 1, tempFile);

}

}

fclose(file);

fclose(tempFile);

remove("employee.dat");

rename("temp.dat", "employee.dat");

printf("Record deleted successfully!\n");

}

int main() {

FILE \*file = fopen("employee.dat", "rb+");

if (file == NULL) {

file = fopen("employee.dat", "wb+");

}

if (file == NULL) {

printf("Error opening the file.\n");

return 1;

}

int choice;

do {

printf("\nEmployee Database\n");

printf("1. Add Record\n");

printf("2. Display All Records\n");

printf("3. Update Record\n");

printf("4. Delete Record\n");

printf("5. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

addRecord(file);

break;

case 2:

displayAllRecords(file);

break;

case 3:

updateRecord(file);

break;

case 4:

deleteRecord(file);

break;

case 5:

printf("Exiting program.\n");

break;

default:

printf("Invalid choice. Please try again.\n");

}

} while (choice != 5);

fclose(file);

return 0;

}

**Random access file**

#include <stdio.h>

#include <stdlib.h>

struct Employee {

int empId;

char empName[50];

float empSalary;

};

void addRecord(FILE \*file) {

struct Employee emp;

printf("Enter Employee ID: ");

scanf("%d", &emp.empId);

printf("Enter Employee Name: ");

scanf("%s", emp.empName);

printf("Enter Employee Salary: ");

scanf("%f", &emp.empSalary);

fseek(file, (emp.empId - 1) \* sizeof(struct Employee), SEEK\_SET);

fwrite(&emp, sizeof(struct Employee), 1, file);

printf("Record added successfully!\n");

}

void displayAllRecords(FILE \*file) {

struct Employee emp;

rewind(file);

printf("\nEmployee Records:\n");

printf("ID\tName\tSalary\n");

while (fread(&emp, sizeof(struct Employee), 1, file) == 1) {

printf("%d\t%s\t%.2f\n", emp.empId, emp.empName, emp.empSalary);

}

}

void updateRecord(FILE \*file) {

struct Employee emp;

int empId;

printf("Enter Employee ID to update: ");

scanf("%d", &empId);

fseek(file, (empId - 1) \* sizeof(struct Employee), SEEK\_SET);

if (fread(&emp, sizeof(struct Employee), 1, file) != 1) {

printf("Employee not found!\n");

return;

}

printf("Enter new Employee Name: ");

scanf("%s", emp.empName);

printf("Enter new Employee Salary: ");

scanf("%f", &emp.empSalary);

fseek(file, -sizeof(struct Employee), SEEK\_CUR);

fwrite(&emp, sizeof(struct Employee), 1, file);

printf("Record updated successfully!\n");

}

void deleteRecord(FILE \*file) {

int empId;

struct Employee emp;

FILE \*tempFile;

printf("Enter Employee ID to delete: ");

scanf("%d", &empId);

tempFile = fopen("temp.dat", "wb");

rewind(file);

while (fread(&emp, sizeof(struct Employee), 1, file) == 1) {

if (emp.empId != empId) {

fwrite(&emp, sizeof(struct Employee), 1, tempFile);

}

}

fclose(file);

fclose(tempFile);

remove("employee.dat");

rename("temp.dat", "employee.dat");

printf("Record deleted successfully!\n");

}

int main() {

FILE \*file = fopen("employee.dat", "rb+");

if (file == NULL) {

file = fopen("employee.dat", "wb+");

}

if (file == NULL) {

printf("Error opening the file.\n");

return 1;

}

int choice;

do {

printf("\nEmployee Database\n");

printf("1. Add Record\n");

printf("2. Display All Records\n");

printf("3. Update Record\n");

printf("4. Delete Record\n");

printf("5. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

addRecord(file);

break;

case 2:

displayAllRecords(file);

break;

case 3:

updateRecord(file);

break;

case 4:

deleteRecord(file);

break;

case 5:

printf("Exiting program.\n");

break;

default:

printf("Invalid choice. Please try again.\n");

}

} while (choice != 5);

fclose(file);

return 0;

}

**SP Waiting process**

#include<stdio.h>

int main()

{

int bt[20],p[20],wt[20],tat[20],i,j,n,total=0,pos,temp;

float avg\_wt,avg\_tat;

printf("Enter number of process:");

scanf("%d",&n);

printf("nEnter Burst Time:n");

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

{

printf("p%d:",i+1);

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

p[i]=i+1;

}

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

{

pos=i;

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

{

if(bt[j]<bt[pos])

pos=j;

}

temp=bt[i];

bt[i]=bt[pos];

bt[pos]=temp;

temp=p[i];

p[i]=p[pos];

p[pos]=temp;

}

wt[0]=0;

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

{

wt[i]=0;

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

wt[i]+=bt[j];

total+=wt[i];

}

avg\_wt=(float)total/n;

total=0;

printf("nProcesst Burst Time tWaiting TimetTurnaround Time");

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

{

tat[i]=bt[i]+wt[i];

total+=tat[i];

printf("np%dtt %dtt %dttt%d",p[i],bt[i],wt[i],tat[i]);

}

avg\_tat=(float)total/n;

printf("nnAverage Waiting Time=%f",avg\_wt);

printf("nAverage Turnaround Time=%fn",avg\_tat);

}

**Scan Disk Scheduling Algorithm**

#include <stdio.h>

#include <stdlib.h>

#define DIRECTION\_LEFT 0

#define DIRECTION\_RIGHT 1

void sort(int arr[], int n) {

int i, j, temp;

for (i = 0; i < n - 1; i++) {

for (j = 0; j < n - i - 1; j++) {

if (arr[j] > arr[j + 1]) {

// Swap the elements if they are in the wrong order

temp = arr[j];

arr[j] = arr[j + 1];

arr[j + 1] = temp;

}

}

}

}

int main() {

int diskSize, currentPosition, totalRequests;

printf("Enter the size of the disk: ");

scanf("%d", &diskSize);

printf("Enter the current position of the disk arm: ");

scanf("%d", &currentPosition);

printf("Enter the total number of disk requests: ");

scanf("%d", &totalRequests);

int requests[totalRequests];

int i;

printf("Enter the disk request track numbers:\n");

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

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

}

// Sort the disk requests in ascending order

sort(requests, totalRequests);

int direction;

printf("Enter the initial direction (0 for left, 1 for right): ");

scanf("%d", &direction);

int seekCount = 0;

if (direction == DIRECTION\_LEFT) {

// SCAN towards the left

for (i = currentPosition; i >= 0; i--) {

printf("Servicing track %d\n", i);

seekCount += abs(currentPosition - i);

currentPosition = i;

}

// Change direction

direction = DIRECTION\_RIGHT;

}

// SCAN towards the right

for (i = currentPosition; i <= diskSize; i++) {

printf("Servicing track %d\n", i);

seekCount += abs(currentPosition - i);

currentPosition = i;

}

// Simulate reverse direction towards the left

if (direction == DIRECTION\_RIGHT) {

for (i = currentPosition; i >= 0; i--) {

printf("Servicing track %d\n", i);

seekCount += abs(currentPosition - i);

currentPosition = i;

}

}

printf("Total seek count: %d\n", seekCount);

return 0;

}

**Shared Memory**

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#include<sys/shm.h>

#include<string.h>

int main()

{

int i;

void \*shared\_memory;

char buff[100];

int shmid;

shmid=shmget((key\_t)2345, 1024, 0666|IPC\_CREAT);

printf("Key of shared memory is %d\n",shmid);

shared\_memory=shmat(shmid,NULL,0);

printf("Process attached at %p\n",shared\_memory);

printf("Enter some data to write to shared memory\n");

read(0,buff,100);

strcpy(shared\_memory,buff);

printf("You wrote : %s\n",(char \*)shared\_memory);

}

**Two level directory structure**

#include<stdio.h>

#include<conio.h>

struct st

{

char dname[10];

char sdname[10][10];

char fname[10][10][10];

int ds,sds[10];

}dir[10];

int main()

{

int i,j,k,n;

printf("enter number of directories:");

scanf("%d",&n);

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

{

printf("enter directory %d names:",i+1);

scanf("%s",&dir[i].dname);

printf("enter size of directories:");

scanf("%d",&dir[i].ds);

for(j=0;j<dir[i].ds;j++)

{

printf("enter subdirectory name and size:");

scanf("%s",&dir[i].sdname[j]);

scanf("%d",&dir[i].sds[j]);

for(k=0;k<dir[i].sds[j];k++)

{

printf("enter file name:");

scanf("%s",&dir[i].fname[j][k]);

}

}

}

printf("\ndirname\t\tsize\tsubdirname\tsize\tfiles");

printf("\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

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

{

printf("%s\t\t%d",dir[i].dname,dir[i].ds);

for(j=0;j<dir[i].ds;j++)

{

printf("\t%s\t\t%d\t",dir[i].sdname[j],dir[i].sds[j]);

for(k=0;k<dir[i].sds[j];k++)

printf("%s\t",dir[i].fname[j][k]);

printf("\n\t\t");

}

printf("\n");

}

getch();

}

**UNIX Command**

#include <stdio.h>

#include <stdlib.h>

#include <dirent.h>

void lsSimulation(const char \*directoryPath) {

DIR \*dir;

struct dirent \*entry;

dir = opendir(directoryPath);

if (dir == NULL) {

perror("Error opening directory");

exit(EXIT\_FAILURE);

}

while ((entry = readdir(dir)) != NULL) {

printf("%s\n", entry->d\_name);

}

closedir(dir);

}

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

const char \*directoryPath;

if (argc < 2) {

// No directory specified, list the current directory

directoryPath = ".";

} else {

directoryPath = argv[1];

}

lsSimulation(directoryPath);

return 0;

}

**Worst Fit Algorithm**

#include <stdio.h>

#define MAX\_BLOCKS 100

#define MAX\_PROCESS 100

void worstFit(int blockSize[], int m, int processSize[], int n) {

int allocation[MAX\_PROCESS];

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

allocation[i] = -1; // Initialize allocation array to -1 (indicating not allocated)

}

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

int worstIndex = -1;

for (int j = 0; j < m; j++) {

if (blockSize[j] >= processSize[i]) {

if (worstIndex == -1 || blockSize[j] > blockSize[worstIndex]) {

worstIndex = j;

}

}

}

if (worstIndex != -1) {

allocation[i] = worstIndex;

blockSize[worstIndex] -= processSize[i];

}

}

// Display the allocation result

printf("Process No.\tProcess Size\tBlock No.\n");

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

printf("%d\t\t%d\t\t", i + 1, processSize[i]);

if (allocation[i] != -1) {

printf("%d\n", allocation[i] + 1);

} else {

printf("Not Allocated\n");

}

}

}

int main() {

int m; // Number of memory blocks

int n; // Number of processes

printf("Enter the number of memory blocks: ");

scanf("%d", &m);

int blockSize[MAX\_BLOCKS];

printf("Enter the size of each memory block:\n");

for (int i = 0; i < m; i++) {

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

}

printf("Enter the number of processes: ");

scanf("%d", &n);

int processSize[MAX\_PROCESS];

printf("Enter the size of each process:\n");

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

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

}

worstFit(blockSize, m, processSize, n);

return 0;

}

**classical process synchronization prblm**

#include<pthread.h>

#include<stdio.h>

#include<semaphore.h>

#include<unistd.h>

void \*fun1();

void \*fun2();

int shared=1; //shared variable

sem\_t s; //semaphore variable

int main()

{

int x;

sem\_wait(&s); //executes wait operation on s

x=shared;//thread1 reads value of shared variable

printf("Thread1 reads the value as %d\n",x);

x++; //thread1 increments its value

printf("Local updation by Thread1: %d\n",x);

sleep(1); //thread1 is preempted by thread 2

shared=x; //thread one updates the value of shared variable

printf("Value of shared variable updated by Thread1 is: %d\n",shared);

sem\_post(&s);

}

void \*fun2()

{

int y;

sem\_wait(&s);

y=shared;//thread2 reads value of shared variable

printf("Thread2 reads the value as %d\n",y);

y--; //thread2 increments its value

printf("Local updation by Thread2: %d\n",y);

sleep(1); //thread2 is preempted by thread 1

shared=y; //thread2 updates the value of shared variable

printf("Value of shared variable updated by Thread2 is: %d\n",shared);

sem\_post(&s);

}

**memory allocation using first fit starteg**

#include<stdio.h>

int main()

{

int bsize[10], psize[10], bno, pno, flags[10], allocation[10], i, j;

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

{

flags[i] = 0;

allocation[i] = -1;

}

printf("Enter no. of blocks: ");

scanf("%d", &bno);

printf("\nEnter size of each block: ");

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

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

printf("\nEnter no. of processes: ");

scanf("%d", &pno);

printf("\nEnter size of each process: ");

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

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

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

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

if(flags[j] == 0 && bsize[j] >= psize[i])

{

allocation[j] = i;

flags[j] = 1;

break;

}

printf("\nBlock no.\tsize\t\tprocess no.\t\tsize");

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

{

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

if(flags[i] == 1)

printf("%d\t\t\t%d",allocation[i]+1,psize[allocation[i]]);

else

printf("Not allocated");

}

}

**organizing file using single level directory**

#include<stdio.h>

#include<conio.h>

#include<string.h>

int main()

{

int nf=0,i=0,j=0,ch;

char mdname[10],fname[10][10],name[10];

printf("Enter the directory name:");

scanf("%s",mdname);

printf("Enter the number of files:");

scanf("%d",&nf);

do

{

printf("Enter file name to be created:");

scanf("%s",name);

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

{

if(!strcmp(name,fname[i]))

break;

}

if(i==nf)

{

strcpy(fname[j++],name);

nf++;

}

else

printf("There is already %s\n",name);

printf("Do you want to enter another file(yes - 1 or no - 0):");

scanf("%d",&ch);

}

while(ch==1);

printf("Directory name is:%s\n",mdname);

printf("Files names are:");

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

printf("\n%s",fname[i]);

getch();

}

**round robin scheduling**

#include<stdio.h>

#include<conio.h>

int main()

{

int i, NOP, sum=0,count=0, y, quant, wt=0, tat=0, at[10], bt[10], temp[10];

float avg\_wt, avg\_tat;

printf(" Total number of process in the system: ");

scanf("%d", &NOP);

y = NOP;

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

{

printf("\n Enter the Arrival and Burst time of the Process[%d]\n", i+1);

printf(" Arrival time is: \t");

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

printf(" \nBurst time is: \t");

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

temp[i] = bt[i];

}

printf("Enter the Time Quantum for the process: \t");

scanf("%d", &quant);

printf("\n Process No \t\t Burst Time \t\t TAT \t\t Waiting Time ");

for(sum=0, i = 0; y!=0; )

{

if(temp[i] <= quant && temp[i] > 0)

{

sum = sum + temp[i];

temp[i] = 0;

count=1;

}

else if(temp[i] > 0)

{

temp[i] = temp[i] - quant;

sum = sum + quant;

}

if(temp[i]==0 && count==1)

{

y--;

printf("\nProcess No[%d] \t\t %d\t\t\t\t %d\t\t\t %d", i+1, bt[i], sum-at[i], sum-at[i]-bt[i]);

wt = wt+sum-at[i]-bt[i];

tat = tat+sum-at[i];

count =0;

}

if(i==NOP-1)

{

i=0;

}

else if(at[i+1]<=sum)

{

i++;

}

else

{

i=0;

}

}

avg\_wt = wt \* 1.0/NOP;

avg\_tat = tat \* 1.0/NOP;

printf("\n Average Turn Around Time: \t%f", avg\_wt);

printf("\n Average Waiting Time: \t%f", avg\_tat);

getch();

}

**system calls to copy content**

#include <stdio.h>

#include <stdlib.h>

#include <fcntl.h>

#include <unistd.h>

#define BUF\_SIZE 4096

int main() {

int sourceFile, destinationFile;

ssize\_t bytesRead, bytesWritten;

char buffer[BUF\_SIZE];

// Open the source file for reading

sourceFile = open("source.txt", O\_RDONLY);

if (sourceFile == -1) {

perror("Error opening source file");

exit(EXIT\_FAILURE);

}

// Open the destination file for writing (create if not exists, truncate to 0 if exists)

if (destinationFile == -1) {

perror("Error opening destination file");

close(sourceFile);

exit(EXIT\_FAILURE);

}

// Read from the source file and write to the destination file

while ((bytesRead = read(sourceFile, buffer, BUF\_SIZE)) > 0) {

bytesWritten = write(destinationFile, buffer, bytesRead);

if (bytesWritten != bytesRead) {

perror("Error writing to destination file");

close(sourceFile);

close(destinationFile);

exit(EXIT\_FAILURE);

}

}

if (bytesRead == -1) {

perror("Error reading from source file");

close(sourceFile);

close(destinationFile);

exit(EXIT\_FAILURE);

}

// Close the files

close(sourceFile);

close(destinationFile);

printf("File copy successful!\n");

return 0;

}