CSA0470 –

(OPERATING SYSTEMS with design principles)

K.Vaishnavi(192110431).

LABORATORY RECORD

INDEX

|  |  |  |  |
| --- | --- | --- | --- |
| **SL.NO** | **DATE** | **PROGRAM NAME** | **PAGE NO** |
|  | **26-09-22** | **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.** |  |
|  | **26-09-22** | **Identify the system calls to copy the content of one file to another and illustrate the same using a C program.** |  |
|  | **26-09-22** | **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.** |  |
|  | **26-09-22** | **Construct a scheduling program with C that selects the waiting process with the smallest execution time to execute next.** |  |
| **14** | **28-09-22** | **Illustrate the deadlock avoidance concept by simulating Banker’s algorithm with C.** |  |
| **15** | **28-09-22** | **Construct a C program to simulate producer-consumer problem using semaphores.** |  |
| **16** | **28-09-22** | **Construct a program to simulate the First in First Out paging technique of memory management. When a page must be replaced, the oldest page is chosen.** |  |
| **17** | **28-09-22** | **Construct a program to simulate the Least Recently Used paging technique of memory management. When a page must be replaced, the oldest page is chosen** |  |
| **18** | **28-09-22** | **Construct a program to simulate the optimal paging technique of memory management. The operating system replaces the page that will not be used for the longest period of time in future** |  |
| **19** | **28-09-22** | **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 program to simulate the file allocation strategy.** |  |
| **20** | **28-09-22** | **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 program to simulate the file allocation strategy.** |  |
| **21** | **28-09-22** | **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 program to simulate the file allocation strategy.** |  |
| **22** | **28-09-22** | **Construct a C program to simulate the First Come First Served disk scheduling algorithm.** |  |
| **25** | **28-09-22** | **Illustrate the various File Access Permission and different types users in Linux** |  |

|  |
| --- |
| EXPERIMENT 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 |

AIM:

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.

ALGORITHM:

Step 1 : Start program.

Step 2 : An existing process can create a new one by calling the fork() function.

Step 3 : The new process created by fork() is called the child process.

Step 4 : We are using her getpid()to get the process id.

Step 5 : Stop program..

PROGRAM:

#include <stdio.h>

#include <stdlib.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <unistd.h>

int main(void) {

pid\_t pid = fork();

if(pid == 0) {

printf("Child => PPID: %d PID: %d\n", getppid(), getpid());

exit(EXIT\_SUCCESS);

}

else if(pid > 0) {

printf("Parent => PID: %d\n", getpid());

printf("Waiting for child process to finish.\n");

wait(NULL);

printf("Child process finished.\n");

}

else {

printf("Unable to create child process.\n");

}

return EXIT\_SUCCESS;

}

OUTPUT:

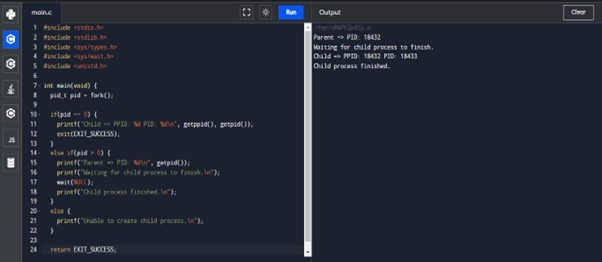
Parent => PID: 18432

Waiting for child process to finish.

Child => PPID: 18432 PID: 18433

Child process finished.

INPUT AND OUTPUT:



RESULT:

Thus the program has been successfully implemented using system call

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

Aim:

To identify the system calls to copy the content of one file to another and illustrate the same using a C program.

ALGORITHM:

Step 1 : Start program.

Step 2 :type the file name that file is readable

Step 3 : type the file name that the content should be copied in the other file

Step 4 : the coping of the content is done

Step 5 : Stop program.

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);

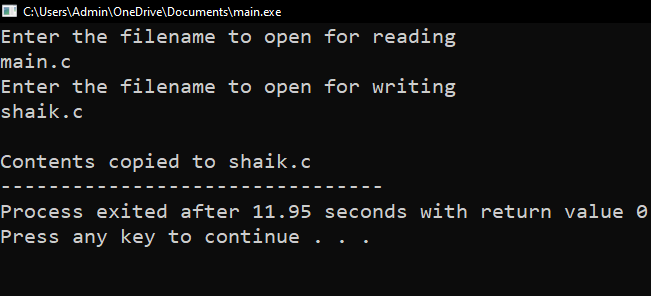
fclose(fptr1);

fclose(fptr2);

return 0;

}

INPUT AND OUTPUT:



Result: Thus the program copying one file to another has been successfully implemented using system calls.

|  |
| --- |
| EXPERIMENT 3:  Desig n 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. |

AIM:

ToDesign 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.

AlGORITHM:

Step 1: start program  
Step 2: Inside the structure declare the variables.

Step 3: Declare the variable i, j as integer, to time and to time is equal to zero.

Step 4: Get the value of „n‟ assign pid as I and get the value of p[i].b time.

Step 5: Assign p[0] wtime as zero and tot time as btime and inside the loop calculate wait time and turnaround time.

Step 6: Calculate total wait time and total turnaround time by dividing by total number of process. Step 7: Print total wait time and total turnaround time.

Step 8: Stop the program

PROGRAM:

#include<stdio.h>

void main()

{

int n,bt[20],wt[20],tat[20],i,j; float avwt=0,avtat=0;printf("Enter total number of processes(maximum 20):");scanf("%d",&n);

printf("\nEnter Process Burst Time\n");for(i=0;i<n;i++)

{

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

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

} wt[0]=0;

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

{ wt[i]=0;for(j=0;j<i;j++)

wt[i]+=bt[j];

}

printf("\nProcess\t\tBurst Time\tWaiting Time\tTurnaround Time"); for(i=0;i<n;i++)

{

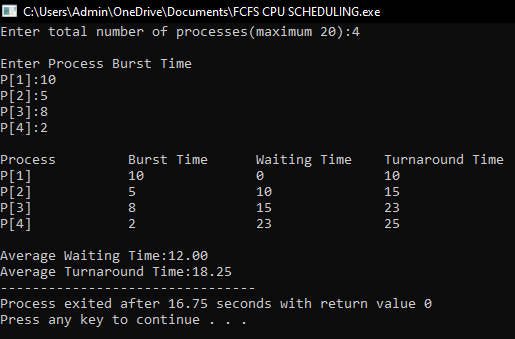
tat[i]=bt[i]+wt[i]; avwt+=wt[i]; avtat+=tat[i];printf("\nP[%d]\t\t%d\t\t%d\t\t%d",i+1,bt[i],wt[i],tat[i]);

} avwt/=i; avtat/=i;printf("\n\nAverage Waiting Time:%.2f",avwt);

printf("\nAverage Turnaround Time:%.2f",avtat);

}

INPUT AND OUTPUT:



RESULT:

Thus the program of cpu scheduling by using “frist come first serve”technique is implemented successfully..

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

AIM:

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

ALGORITHM:

Step 1: Inside the structure declare the variables.

Step 2: Declare the variable i, j as integer, totwtime and totttime is equal to zero.

Step 3: Get the value of „n‟ assign pid as I and get the value of p[i].btime.

Step 4: Assign p[0] w time as zero and tot time as btime and inside the loop calculate wait time and turnaround time.

Step 5: Calculate total wait time and total turnaround time by dividing by total number of process. Step 6: Print total wait time and total turnaround time.

Step 7: Stop the program.

PROGRAM:

#include<stdio.h>

int main()

{

int bt[20],p[20],wt[20],tat[20],i,j,n,total=0,pos,temp,floatavg\_wt,avg\_tat,avg\_wt;

printf("Enter number of process:");

scanf("%d",&n);

printf("\n Enter 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("\nProcess\t Burst Time \tWaiting Time\tTurnaround Time");for(i=0;i<n;i++)

{

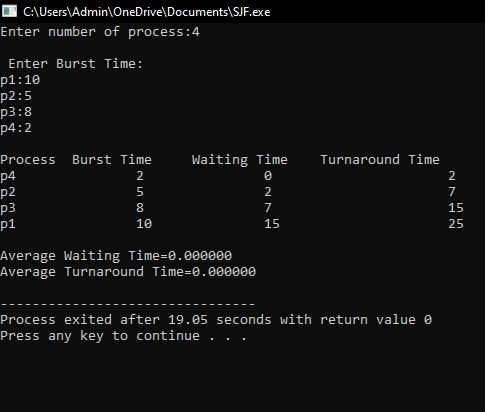
tat[i]=bt[i]+wt[i]; total+=tat[i];printf("\np%d\t\t %d\t\t %d\t\t\t%d",p[i],bt[i],wt[i],tat[i]);

}avg\_tat=(float)total/n;printf("\n\nAverage Waiting Time=%f",avg\_wt);

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

}

OUTPUT:



RESULT: Thus the c program for **scheduling program with C that selects the waiting process with the smallest execution time to execute next successfully implemented.**

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

AIM: To **illustrate the deadlock avoidance concept by simulating Banker’s algorithm with C.**

**ALGORITHM:**

Step-1: Start the program.

Step-2: Declare the memory for the process.

Step-3: Read the number of process, resources, allocation matrix and available matrix.

Step-4: Compare each and every process using the banker‟s algorithm.

Step-5: If the process is in safe state then it is a not a deadlock process otherwise it is a deadlock process

Step-6: produce the result of state of process

Step-7: Stop the program.

PROGRAM:

**#include<stdio.h>**

**int main()**

**{**

**int count = 0, m, n, process, temp, resource; int allocation\_table[5] = {0, 0, 0, 0, 0}; int available[5], current[5][5], maximum\_claim[5][5]; int maximum\_resources[5], running[5], safe\_state = 0; printf("\nEnter The Total Number Of Processes:\t"); scanf("%d", &process); for(m=0;m<process;m++)**

**{ running[m]=1; count++;**

**}**

**printf("\nEnter The Total Number Of Resources To Allocate:\t"); scanf("%d",&resource);**

**printf("\nEnter The Claim Vector:\t"); for(m=0;m<resource;m++)**

**{**

**scanf("%d",&maximum\_resources[m]);**

**}**

**printf("\nEnter Allocated Resource Table:\n"); for(m=0;m<process;m++)**

**{**

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

**{**

**scanf("%d",&current[m][n]);**

**}**

**}**

**printf("\nEnter The Maximum Claim Table:\n");for(m=0;m<process;m++)**

**{**

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

**{**

**scanf("%d",&maximum\_claim[m][n]);**

**} }**

**printf("\nThe Claim Vector \n");**

**for(m=0;m<resource;m++)**

**{printf("\t%d ",maximum\_resources[m]);**

**}**

**printf("\n The Allocated Resource Table\n"); for(m=0;m<process;m++)**

**{for(n=0;n<resource;n++)**

**{printf("\t%d",current[m][n]);**

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

**}printf("\nThe Maximum Claim Table \n"); for(m=0;m<process;m++)**

**{for(n=0;n<resource;n++)**

**{printf("\t%d",maximum\_claim[m][n]);**

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

**}for(m=0;m<process;m++)**

**{for(n=0;n<resource;n++)**

**{allocation\_table[n]=allocation\_table[n]+current[m][n];**

**}}**

**printf("\nAllocated Resources \n"); for(m=0;m<resource;m++)**

**{printf("\t%d",allocation\_table[m]);**

**}for(m=0;m<resource;m++)**

**{**

**available[m]=maximum\_resources[m]-allocation\_table[m];**

**}**

**printf("\nAvailable Resources:");**

**for(m=0;m<resource;m++)**

**{**

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

**} printf("\n"); while(count!=0)**

**{ safe\_state=0;**

**for(m=0;m<process;m++)**

**{**

**if(running[m])**

**{ temp=1;**

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

**{**

**if(maximum\_claim[m][n]-current[m][n]>available[n])**

**{ temp=0; break;**

**}**

**} if(temp)**

**{**

**printf("\nProcess %d Is In Execution \n", m + 1); running[m]=0; count--; safe\_state=1;**

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

**{available[n]=available[n]+current[m][n];**

**} break;**

**}}}if(!safe\_state)**

**{printf("\nThe Processes Are In An Unsafe State \n"); break; } else**

**{printf("\nThe Process Is In A Safe State \n"); printf("\nAvailable Vector\n");**

**for(m=0;m<resource;m++)**

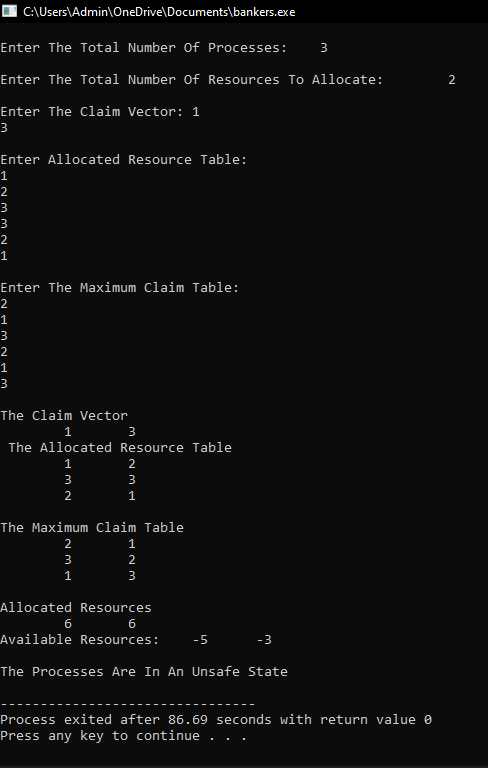
**{printf("\t%d",available[m]);**

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

**}}**

**}**

**OUTPUT:**



**RESULT:**

**Thus the program to illustrate the deadlock avoidance concept by simulating Banker’s algorithm is successfully implemented.**

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

**AIM:**

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

**ALGORITHM:**

**Step 1:Start the program**

**Writer requests the entry to critical section.**

**Step 2: If allowed wait() gives a true value, it enters and performs write. If not it exits.**

**Step 3: Reader requests the entry to critical section.**

**Step 4: If allowed, it increments the count of the number of readers inside the critical section**

**Step 5: If not it keeps on waiting**

**Step 6: Stop the program.**

**PROGRAM:**

**#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);**

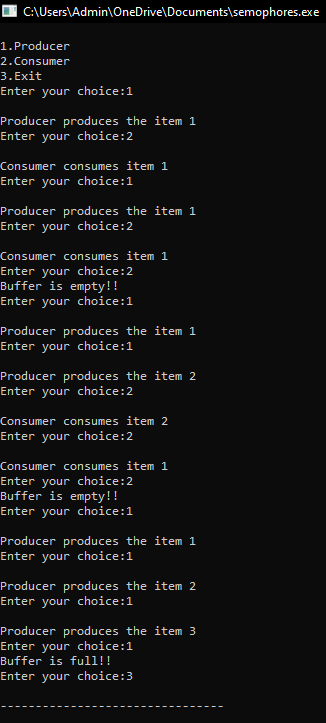
**printf("\nConsumer consumes item %d",x);**

**x--;**

**mutex=signal(mutex);**

**}**

**OUTPUT:**



**RESULT:**

**Thus the C program to simulate producer-consumer problem using semaphores is successfully implemented.**

|  |
| --- |
| **EXPERIMENT 16:**  **Construct a program to simulate the First in First Out paging technique of memory management. When a page must be replaced, the oldest page is chosen.** |

**AIM:  
 To construct a program to simulate the Least Recently Used paging technique of memory management. When a page must be replaced, the oldest page is chosen.**

**ALGORITHM:**

Step 1: Start the program.

Step 2: Declare the necessary variables.

Step 3: Enter the number of frames.

Step 4: Enter the reference string end with zero.

Step 5: FIFO page replacement selects the page that has been in memory the longest time and when the page must be replaced the oldest page is chosen. Step 6: When a page is brought into memory, it is inserted at the tail of the queue.

Step 7: Initially all the three frames are empty.

Step 8: The page fault range increases as the no of allocated frames also increases.

Step 9: Print the total number of page faults.

Step 10: Stop the program.

PROGRAM:

#include<stdio.h>

Int main ()

{

int i,j,n,a[50],frame[10],nf,k,avail,count=0;

printf("\n ENTER THE NUMBER OF PAGES:\n");

scanf("%d",&n);

printf("\n ENTER THE PAGE

NUMBER:\n");for(i=1;i<=n;i++)scanf("%d",&a[i]);

printf("\n ENTER THE NUMBER OF FRAMES :");

scanf("%d",&nf);for(i=0;i<nf;i++)frame[i]= -1;j=0;

printf("\tref string\t page frames\n");for(i=1;i<=n;i++)

{

printf("%d\t\t",a[i]);avail=0;for(k=0;

k<nf;k++)

if(frame[k]==a[i])avail=1;

if (avail==0)

{

frame[j]=a[i];j=(j+1)%nf;c

ount++;for(k=0;k<nf;k++)

printf("%d\t",frame[k]);

}

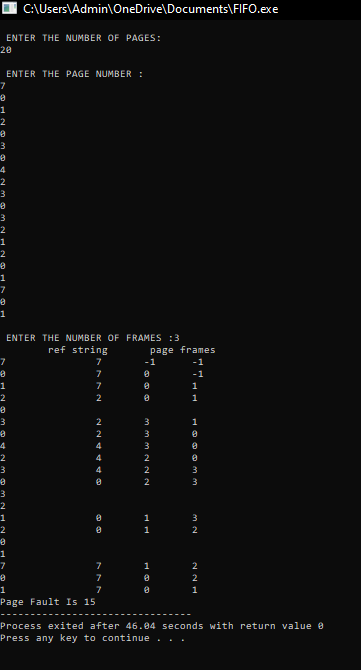
printf("\n");

}

printf("Page Fault Is %d",count);

}

OUTPUT:



RESULT:

Thus the c program to simulate the First in First Out paging technique of memory management. When a page must be replaced, the oldest page is chosen is successfully implemented.

|  |
| --- |
| **EXPERIMENT 17:**  **Construct a program to simulate the Least Recently Used paging technique of memory management. When a page must be replaced, the oldest page is chosen** |

AIM:

To Construct a program to simulate the Least Recently Used paging technique of memory management. When a page must be replaced, the oldest page is chosen

ALGORITHM:

Step 1: Start the program.

Step 2: Declare the necessary variables.

Step 3: Enter the number of frames.

Step 4: Enter the reference string end with zero.

Step 5: FIFO page replacement selects the page that has been in memory the longest time and when the page must be replaced the oldest page is chosen. Step 6: When a page is brought into memory, it is inserted at the tail of the queue.

Step 7: Initially all the three frames are empty.

Step 8: The page fault range increases as the no of allocated frames also increases.

Step 9: Print the total number of page faults.

Step 10: Stop the program.

**PROGRAM:**

#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){

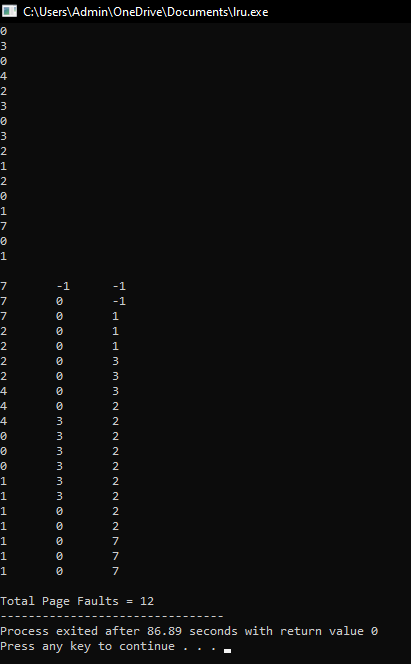
printf("%d\t", frames[j]);

}}

printf("\n\nTotal Page Faults = %d", faults); return 0;

}

OUTPUT:



RESULT: Thus the c program to simulate the Least Recently Used paging technique of memory management. When a page must be replaced, the oldest page is chosen is successfully implemented.

|  |
| --- |
| **EXPERIMENT 18:**  **Construct a program to simulate the optimal paging technique of memory management. The operating system replaces the page that will not be used for the longest period of time in future** |

AIM:

To Construct a program to simulate the optimal paging technique of memory management. The operating system replaces the page that will not be used for the longest period of time in future.

ALGORITHM:

Step 1: Start the program.

Step 2: Declare the necessary variables.

Step 3: Enter the number of frames.

Step 4: Enter the reference string end with zero.

Step 5: FIFO page replacement selects the page that has been in memory the longest time and when the page must be replaced the oldest page is chosen. Step 6: When a page is brought into memory, it is inserted at the tail of the queue.

Step 7: Initially all the three frames are empty.

Step 8: The page fault range increases as the no of allocated frames also increases.

Step 9: Print the total number of page faults.

Step 10: Stop the program.

**PROGRAM:**

#include<stdio.h>

void 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]);

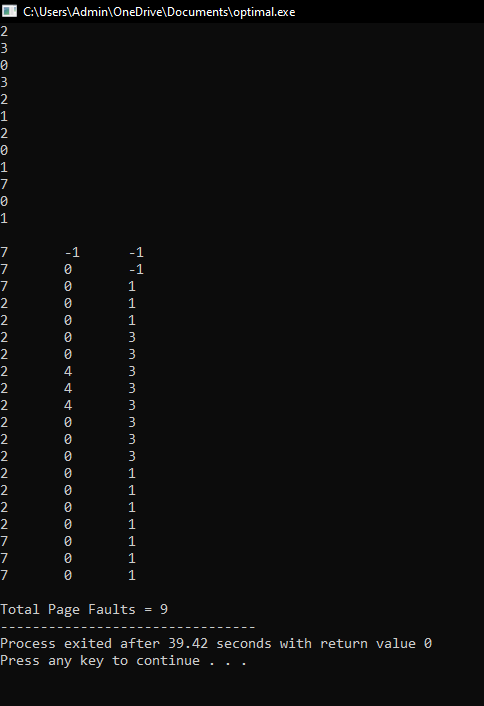
}

}

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

}

OUTPUT:



RESULT:  
 Thus the c program to simulate the optimal paging technique of memory management. The operating system replaces the page that will not be used for the longest period of time in future is successfully implemented.

|  |
| --- |
| **EXPERIMENT 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 program to simulate the file allocation strategy.** |

AIM:

To 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 program to simulate the file allocation strategy.

ALGORITHM:

STEP 1: Start the program.

STEP 2: Gather information about the number of files.

STEP 3: Gather the memory requirement of each file.

STEP 4: Allocate the memory to the file in a sequential manner.

STEP 5: Select any random location from the available location.

STEP 6: Check if the location that is selected is free or not.

STEP 7: If the location is allocated set the flag = 1.

STEP 8: Print the file number, length, and the block allocated.

STEP 9: Gather information if more files have to be stored.

STEP 10: If yes, then go to STEP 2.

STEP 11: If no, Stop the program.

PROGRAM:

#include <stdio.h>

#include <conio.h>

#include <stdlib.h>

void recurse(int files[]){

int flag = 0, startBlock, len, j, k, ch;

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

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

for (j=startBlock; j<(startBlock+len); j++){

if (files[j] == 0)

flag++;

}

if(len == flag){

for (int k=startBlock; k<(startBlock+len); k++){

if (files[k] == 0){

files[k] = 1;

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

}

}

if (k != (startBlock+len-1))

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

}

else

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

printf("Do you want to enter more files?\n");

printf("Press 1 for YES, 0 for NO: ");

scanf("%d", &ch);

if (ch == 1)

recurse(files);

else

exit(0);

return;

}

int main()

{

int files[50];

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

files[i]=0;

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

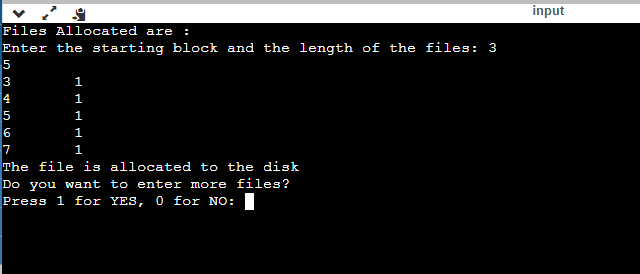
recurse(files);

getch();

return 0;

}

OUTPUT:



RESULT: Thus the c program to 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 program to simulate the file allocation strategy is successfully implemented.

|  |
| --- |
| **EXPERIMENT 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 program to simulate the file allocation strategy.** |

AIM:

To 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 program to simulate the file allocation strategy.

ALGORITHM:

STEP 1: Start the program.

STEP 2: Get information about the number of files.

STEP 3: Get the memory requirement of each file.

STEP 4: Allocate the memory to the file by selecting random locations.

STEP 5: Check if the location that is selected is free or not.

STEP 6: If the location is allocated set the flag = 1, and if free set flag = 0.

STEP 7: Print the file number, length, and the block allocated.

STEP 8: Gather information if more files have to be stored.

STEP 9: If yes, then go to STEP 2.

STEP 10: If no, Stop the program.

PROGRAM:

#include <stdio.h>

#include <conio.h>

#include <stdlib.h>

int files[50], indexBlock[50], indBlock, n;

void recurse1();

void recurse2();

void recurse1(){

printf("Enter the index block: ");

scanf("%d", &indBlock);

if (files[indBlock] != 1){

printf("Enter the number of blocks and the number of files needed for the index %d on the disk: ", indBlock);

scanf("%d", &n);

}

else{

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

recurse1();

}

recurse2();

}

void recurse2(){

int ch;

int flag = 0;

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

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

if (files[indexBlock[i]] == 0)

flag++;

}

if (flag == n){

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

files[indexBlock[j]] = 1;

}

printf("Allocated\n");

printf("File Indexed\n");

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

{

printf("%d ------> %d : %d\n", indBlock, indexBlock[k], files[indexBlock[k]]);

}

}

else{

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

printf("Enter another indexed file\n");

recurse2();

}

printf("Do you want to enter more files?\n");

printf("Enter 1 for Yes, Enter 0 for No: ");

scanf("%d", &ch);

if (ch == 1)

recurse1();

else

exit(0);

return;

}

int main()

{

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

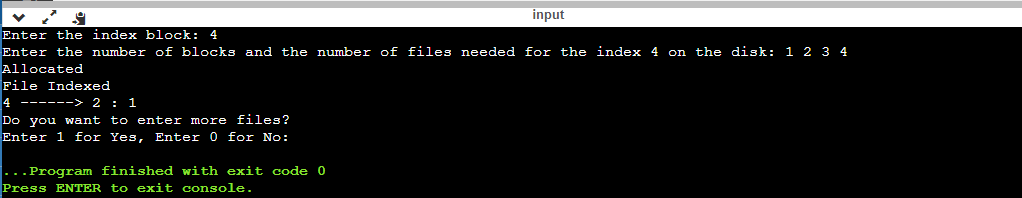
files[i]=0;

recurse1();

return 0;

}

OUTPUT:



RESULT:

Thus the c program to 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 program to simulate the file allocation strategy is successfully implemented.

|  |
| --- |
| **EXPERIMENT 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 program to simulate the file allocation strategy.** |

AIM:

To do 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 program to simulate the file allocation strategy.

ALGORITHM:

STEP 1: Start the program.

STEP 2: Gather information about the number of files.

STEP 3: Allocate random locations to the files.

STEP 4: Check if the location that is selected is free or not.

STEP 5: If the location is free set the flag=0 a location is allocated set the flag .

STEP 6: Print the file number, length, and the block allocated.

STEP 7: Gather information if more files have to be stored.

STEP 8: If yes, then go to STEP 2.

STEP 9: If no, Stop the program.

PROGRAM:

#include <stdio.h>

#include <conio.h>

#include <stdlib.h>

void recursivePart(int pages[]){

int st, len, k, c, j;

printf("Enter the index of the starting block and its length: ");

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

k = len;

if (pages[st] == 0){

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

if (pages[j] == 0){

pages[j] = 1;

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

}

else {

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

k++;

}

}

}

else

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

printf("Do you want to enter more files? \n");

printf("Enter 1 for Yes, Enter 0 for No: ");

scanf("%d", &c);

if (c==1)

recursivePart(pages);

else

exit(0);

return;

}

int main(){

int pages[50], p, a;

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

pages[i] = 0;

printf("Enter the number of blocks already allocated: ");

scanf("%d", &p);

printf("Enter the blocks already allocated: ");

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

scanf("%d", &a);

pages[a] = 1;

}

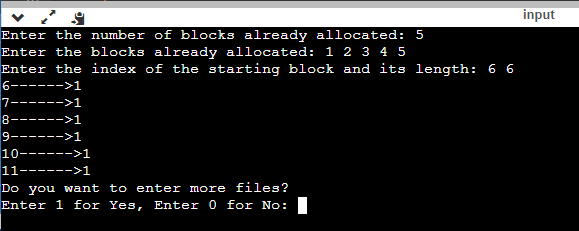
recursivePart(pages);

getch();

return 0;

}

OUTPUT:



RESULT:

Thus the c program to do 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 program to simulate the file allocation strategy is successfully implemented.

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

AIM:

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

ALGORITHM:

Step 1 : start the program.

Step 2 : Let request array represents an array storing indexes of tracks that have been requested in ascending order of their time of arraival “head “ is position of disk head.

Step 3: Let us one by one take the tacks in default order and calculate the absolute distance of the track from the head.

Step 4 : increment the total seek count with this distance.

Step 5:Currently serviced track position now becomes the new head position.

Step 6:Go to step 2 until all tracks in request array have not been serviced.

Step 7:Stop the program.

PROGRAM:

#include<stdio.h>

#include<stdlib.h>

int main()

{

int RQ[100],i,n,TotalHeadMoment=0,initial;

printf("Enter the number of Requests\n");

scanf("%d",&n);

printf("Enter the Requests sequence\n");

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

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

printf("Enter initial head position\n");

scanf("%d",&initial);

// logic for FCFS disk scheduling

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

{

TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);

initial=RQ[i];

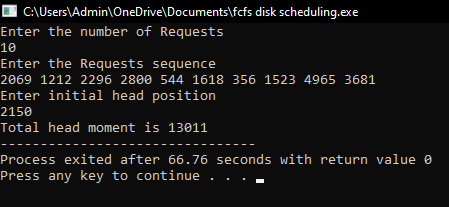
}

printf("Total head moment is %d",TotalHeadMoment);

return 0;

}

OUTPUT:



RESULT: Thus the C program to simulate the First Come First Served disk scheduling algorithm is successfully implemented.

|  |
| --- |
| **EXPERIMENT 25:**  **Illustrate the various File Access Permission and different types users in Linux** |

AIM:

To illustrate the various File Access Permission and different types users in Linux.

1. C program to writing into a file.

AIM:

To write a simple c program to write contents into a file.

ALGORITHM:

Step 1: Start.

Step 2: Open an input file in write mode.

Step 3: Get the contents from the user.

Step 4: Write the contents into the file.

Step 5: Close the file.

Step 6: Stop.

PROGRAM:

# include <stdio.h>

# include <string.h>

void main( )

{ FILE \*fp ;

char data[50];

printf( "Opening the file test.c in write mode" ) ;

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

if ( fp == NULL )

{

printf( "Could not open file test.txt" ) ;

} printf( "\n Enter some text from keyboard to write in the file test.txt" ) ; while ( strlen ( gets( data ) ) > 0 )

{ fputs(data, fp) ;

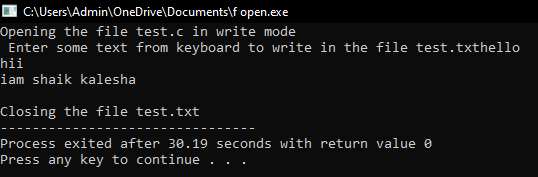
fputs("\n", fp) ;

} printf("Closing the file test.txt") ;

fclose(fp) ;

}

OUTPUT:



2. C program to read the file.

AIM:

To write a simple c program to read the contents of a file and display the same.

ALGORITHM:

Step 1: Start.

Step 2: Open an input file in read mode.

Step 3: Read the contents from the file.

Step 4: Display the contents in the screen.

Step 5: Close the file.

Step 6: Stop.

PROGRAM:

# include <stdio.h>

void main( )

{ FILE \*fp ;

char data[50] ;

printf( "Opening the file test.txt in read mode" ) ;

fp = fopen( "test.txt", "r" ) ;

if ( fp == NULL )

{ printf( "Could not open file test.txt" ) ;

} printf( "Reading the file test.txt" ) ;

while( fgets ( data, 50, fp ) != NULL )

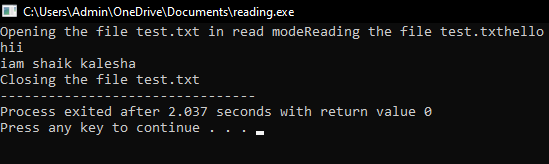
printf( "%s" , data ) ;

printf("Closing the file test.txt") ;

fclose(fp) ;

}

OUT PUT:



3. C program to reversing a file.

AIM:

To write a simple c program for reversing the file contents.

ALGORITHM:

STEP 1:START

STEP 2:Create a file named reverse.c

STEP 3:Get the data from the user.

STEP 4:Store the data in the file.

STEP 5:Reverse the data.

STEP 6:STOP.

PROGRAM:

#include <stdio.h>

void main()

{

FILE \*fp;

char ch;

int i,pos;

fp=fopen("test.txt","r");

if(fp==NULL)

{

printf("file doesnot exist \n");

}

fseek(fp,0,SEEK\_END);

pos=ftell(fp);

i=0;

while(i<pos)

{

i++;

fseek(fp,-i,SEEK\_END);

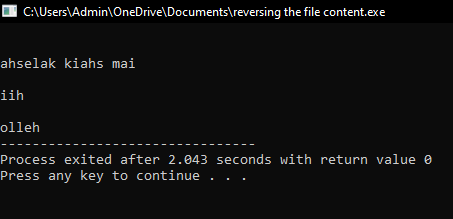
ch=fgetc(fp);

printf("%c",ch);

}

}

OUT PUT:



RESULT: Thus the illustration of the various file access permission and different types users in linux is successfully implemented.