|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **S.NO** | **DATE** | **NAME** | **PAGE NO** | **MARKS** | **SIGNATURE** |
| 1 |  | BASIC UNIX COMMANDS | 2 |  |  |
| 2 A |  | SHELL PROGRAM: ODD OR EVEN | 4 |  |  |
| B |  | SHELL PROGRAM: BIGGEST OF TWO NUMBERS | 5 |  |  |
| C |  | SHELL PROGRAM: GENERATE 1 TO N NUMBERS | 6 |  |  |
| 3 |  | CPU SCHEDULING ALGORITHMS | 8 |  |  |
| 4 |  | Implementation of Semaphores | 16 |  |  |
| 5 |  | Implementation of Shared memory and IPC | 18 |  |  |
| 6 |  | Bankers Algorithm for Deadlock Avoidance | 23 |  |  |
| 7 |  | Implementation of Deadlock Detection Algorithm | 29 |  |  |
| 8 |  | implement Threading & Synchronization Applications | 33 |  |  |
| 9A |  | First Fit | 35 |  |  |
| B |  | Worst Fit | 38 |  |  |
| C |  | Best Fit | 41 |  |  |
| 10 |  | Implementation of Paging Technique of Memory Management | 44 |  |  |
| 11 |  | Implementation of Page Replacement Algorithms | 48 |  |  |
| 12 |  | Implementation of the various File Organization Techniques | 55 |  |  |
| 13 |  | Implementation of File Allocation Strategies | 66 |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

**Ex. No.1 BASIC UNIX COMMANDS**

**Date:**

**Aim:**

To implement basic UNIX commands.

**Directory Commands:**

|  |  |
| --- | --- |
| **Command** | **Description** |
| ls | Gives you a short list of the files in the directory where you are currently working. |
| cd | Moves you to another directory. |
| mkdir | Creates a new subdirectory inside of the directory where  you are currently working |

**File Related Commands:**

|  |  |
| --- | --- |
| **Command** | **Description** |
| cat >FILE | Creates a file |
| more FILE | Display contents of FILE, page by page. |
| cp | Type cp followed by the name of an existing file and the name of the new file.  Eg:  cp sourcefile destfile  To copy a file to a different directory specify the directory  instead of filename.  Eg:  cp newfile testdir  To copy a file to a different directory and create a new file  name, you need to specify a directory/a new file name.  Eg:  cp newfile testdir/newerfile  cp newfile ../newerfile  The .. represents one directory up in the hierarchy. |
| **rm** | Type **rm** followed by the name of a file to remove the file.  Eg:  **rm newfile**  Use the wildcard character to remove several files at once.  Eg:  **rm n\***  This command removes all files beginning with n. |

**General Purpose Commands:**

|  |  |
| --- | --- |
| **Command** | **Description** |
| **date** | print or set the system date and time to set date and time  **date -s "11/20/2003 12:48:00" -** Set the date to the date and time shown.  **date '+DATE: %m/%d/%y%nTIME:%H:%M:%S' -** Would list the time  and date in the below format:  DATE: 02/08/01  TIME:16:44:55 |
| **cal** | Calendar for the month and the year.  **cal [month] [year]**  month Specifies the month for you want the calendar to be  displayed. Must be the numeric representation of the  month. For example: January is 1 and December is 12.  year Specifies the year that you want to be displayed.  **Cal** - Would give you the calendar for this month.  **Cal 12 2000** - Would give you the calendar for December of 2000. |
| **clear** | Clears the screen |

**RESULT:**

The basic UNIX commands are executed successfully.

**Ex. No.: 2a SHELL PROGRAM: ODD OR EVEN**

**Date :**

**AIM:**

To write a shell program to check whether the given number is odd or even.

**ALGORITHM:**

STEP 1: Start the program.

STEP 2: Enter the number.

STEP 3: Check the equation as num %2.

STEP 4: If the value is 0, the number is declared as even

STEP 5: Else the number is odd

STEP 6: Stop the program

**PROGRAM:**

echo "Enter the number"

read num

temp=$(($num%2))

if [ $temp -eq 0 ]

then

echo "even"

else

echo "odd"

fi

**OUTPUT:**

Enter the number

7

odd

**RESULT:**

Thus the shell program to check the number is odd or even is implemented and the output is verified.

**Ex. No.: 2b SHELL PROGRAM: BIGGEST OF TWO NUMBERS**

**Date** :

**AIM:**

To write a shell program to find the biggest of two numbers.

**ALGORITHM:**

STEP 1: Start the program

STEP 2: Get the two numbers

STEP 3: Check whether ‘a’ is greater or ‘b’ is greater

STEP 4: If ‘a’ is greater, declare ‘a’ as the greater number

STEP 5: If ‘b’ is greater, declare ‘b’ as the greater number

STEP 6: Stop the program

**PROGRAM:**

echo "Enter number a"

read a

echo "Enter number b"

read b

if [ $a -gt $b ]

then

echo "a is greater"

else

echo "b is greater"

fi

**OUTPUT:**

Enter number a

8

Enter number b

13

b is greater

**RESULT:**

Thus the program to find the biggest of two numbers was executed and the output was verified.

**Ex. No.:2c SHELL PROGRAM: GENERATE 1 TO N NUMBERS**

**Date :**

**AIM:**

To write a shell program to generate 1 to n numbers

**ALGORITHM:**

STEP 1:Start the program

STEP 2:Initialize the values of num1,num2 as 0and 1.

STEP 3:read the limit.

STEP 4: Generate fibonacci series by adding the current value with the previous value.

STEP 5:Display the output.

STEP6:Stop the program

**PROGRAM:**

echo "Enter a number"

read n

count=0

echo "Series:"

while [ $count -lt $n ]

do

count=$((count+1))

echo $count

done

**OUTPUT:**

Enter a number

5

Series:

1

2

3

4

5

**RESULT:**

Thus the program to generate 1 to N numbers was executed and the output was verified.

**Ex. No. : 3 CPU SCHEDULING ALGORITHMS**

**Date :**

**AIM:**

To write a program to implement the CPU scheduling algorithms: FCFS, SJF, RR and priotity.

**ALGORITHM:**

Step1: Create the number of process.

Step2: Get burst time for each process.

Step3: For FCFS, Schedule the first process that comes in first.

Step 4: For SJF, sort the processes in increasing order of burst time and schedule as FCFS.

Step 4: For priority algorithm, sort the processes in decreasing order of priority and schedule as FCFS.

Step 5: For round robin, schedule every process as in FCFS, but run each process for quantum time.

Step 6: Calculate Waiting and Turnaround time of each process and find their averages.

**PROGRAM:**

#include<stdio.h>

void FCFS();

void SJF();

void priority\_np();

void roundrobin();

int burstTime[10]={0}, process[10];

int wt[10]={0}, tat[10]={0};

int clock, n;

void main()

{

int i;

printf("Enter the no. of processes: ");

scanf("%d", &n);

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

{

printf("Enter Burst Time of P%d: ", i);

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

process[i] = i;

}

clock = 0;

FCFS();

clock = 0;

SJF();

clock = 0;

roundrobin();

clock = 0;

priority\_np();

}

void **FCFS**()

{

int i, avg\_tat = 0, avg\_wt = 0;

printf("\nFCFS\n");

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

{

clock = clock + burstTime[i];

tat[i] = tat[i] + clock;

printf("P%d\t", i);

}

printf("\nProcess\tWaiting\tTurnAround\n");

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

{

wt[i] = tat[i] - burstTime[i];

avg\_wt = avg\_wt + wt[i];

avg\_tat = avg\_tat + tat[i];

printf("\nP%d\t%d\t%d", process[i], wt[i], tat[i]);

}

printf("\nAvg. Waiting Time: %d\nAvg. TurnAround Time: %d\n", avg\_wt/n, avg\_tat/n);

}

void **SJF**()

{

int i, j, small, smallIndex = 0, temp, avg\_wt = 0, avg\_tat = 0;

int burstCopy[10];

printf("\nSJF\n");

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

{

burstCopy[i] = burstTime[i];

tat[i] = 0;

}

//Sort the burst in ascending order

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

{

small = burstCopy[i];

smallIndex = i;

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

{

if (small > burstCopy[j])

{

small = burstCopy[j];

smallIndex = j;

}

}

temp = burstCopy[i];

burstCopy[i] = small;

burstCopy[smallIndex] = temp;

//sort process numbers also

temp = process[i];

process[i] = process[smallIndex];

process[smallIndex] = temp;

}

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

{

clock = clock + burstCopy[i];

tat[i] = tat[i] + clock;

printf("P%d\t", process[i]);

}

printf("\nProcess\tWaiting\tTurnAround\n");

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

{

wt[i] = tat[i] - burstCopy[i];

avg\_wt = avg\_wt + wt[i];

avg\_tat = avg\_tat + tat[i];

printf("\nP%d\t%d\t%d", process[i], wt[i], tat[i]);

}

printf("\nAvg. Waiting Time: %d\nAvg. TurnAround Time: %d\n", avg\_wt/n, avg\_tat/n);

}

void **priority\_np**()

{

int i, j, small, smallIndex, temp, avg\_wt = 0, avg\_tat = 0;

int burstCopy[10], priority[10];

printf("\nPRIORITY\n");

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

{

printf("\nEnter priority of P%d: ",i);

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

burstCopy[i] = burstTime[i];

process[i] = i; //reset the process numbers

tat[i] = 0;

}

//Sort the burst in ascending order of priority

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

{

small = priority[i];

smallIndex = i;

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

{

if (small > priority[j])

{

small = priority[j];

smallIndex = j;

}

}

temp = priority[i];

priority[i] = priority[smallIndex] ;

priority[smallIndex] = temp;

//sort process numbers & burst time also

temp = process[i];

process[i] = process[smallIndex];

process[smallIndex] = temp;

temp = burstCopy[i];

burstCopy[i] = burstCopy[smallIndex];

burstCopy[smallIndex] = temp;

}

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

{

clock = clock + burstCopy[i];

tat[i] = tat[i] + clock;

printf("P%d\t", process[i]);

}

printf("\nProcess\tWaiting\tTurnAround\n");

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

{

wt[i] = tat[i] - burstCopy[i];

avg\_wt = avg\_wt + wt[i];

avg\_tat = avg\_tat + tat[i];

printf("\nP%d\t%d\t%d", process[i], wt[i], tat[i]);

}

printf("\nAvg. Waiting Time: %d\nAvg. TurnAround Time: %d\n", avg\_wt/n, avg\_tat/n);

}

void **roundrobin**()

{

int i, burstCopy[10], avg\_tat = 0, avg\_wt = 0;

int count = 0, quantum = 0;

int flag[10] = {0};

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

{

burstCopy[i] = burstTime[i];

process[i] = i;

tat[i] = 0;

}

printf("Enter quantum: ");

scanf("%d", &quantum);

for (i = 0; i < n; i=(i+1)%n)

{

if (flag[i]==0)

{

if (burstCopy[i] <= quantum)

{

clock = clock + burstCopy[i];

printf("P%d\t", i);

burstCopy[i] = 0;

flag[i] = 1;

count++;

}

else

{

clock = clock + quantum;

printf("P%d\t", i);

burstCopy[i] = burstCopy[i] - quantum;

}

tat[i] = tat[i] + clock;

}

if (count == n)

{

break;

}

}

printf("\nProcess\tWaiting\tTurnAround\n");

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

{

wt[i] = tat[i] - burstTime[i];

avg\_wt = avg\_wt + wt[i];

avg\_tat = avg\_tat + tat[i];

printf("\nP%d\t%d\t%d", process[i], wt[i], tat[i]);

}

printf("\nAvg. Waiting Time: %d\nAvg. TurnAround Time: %d\n", avg\_wt/n, avg\_tat/n);

}

**OUTPUT**:

Enter the no. of processes: 3

Enter Burst Time of P0: 7

Enter Burst Time of P1: 3

Enter Burst Time of P2: 8

FCFS

P0 P1 P2

Process Waiting TurnAround

P0 0 7

P1 7 10

P2 10 18

Avg. Waiting Time: 5

Avg. TurnAround Time: 11

SJF

P1 P0 P2

Process Waiting TurnAround

P1 0 3

P0 3 10

P2 10 18

Avg. Waiting Time: 4

Avg. TurnAround Time: 10

Enter quantum: 2

P0 P1 P2 P0 P1 P2 P0 P2 P0 P2

Process Waiting TurnAround

P0 32 39

P1 10 13

P2 42 50

Avg. Waiting Time: 28

Avg. TurnAround Time: 34

PRIORITY

Enter priority of P0: 3

Enter priority of P1: 1

Enter priority of P2: 2

P1 P2 P0

Process Waiting TurnAround

P1 0 3

P2 3 11

P0 11 18

Avg. Waiting Time: 4

Avg. TurnAround Time: 10

**RESULT:**

Thus the program to implement CPU scheduling algorithms was written, executed and the output was verified successfully.

**EX.NO:4 Implementation of Semaphores**

**Date :**

**Aim:**

To implement semaphores using C / C++ program.

**Algorithm:**

Step1: Start the program.

Step2: include necessary header files

Step3: Declare the functions.

Step4: use inbuilt functions to implement semaphore

Step5:stop

**Program:**

#include <stdio.h>

#include <pthread.h>

#include <semaphore.h>

#include <unistd.h>

sem\_t mutex;

void\* thread(void\* arg)

{

//wait

sem\_wait(&mutex);

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

//critical section

sleep(4);

//signal

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

sem\_post(&mutex);

}

int main()

{

sem\_init(&mutex, 0, 1);

pthread\_t t1,t2;

pthread\_create(&t1,NULL,thread,NULL);

sleep(2);

pthread\_create(&t2,NULL,thread,NULL);

pthread\_join(t1,NULL);

pthread\_join(t2,NULL);

sem\_destroy(&mutex);

return 0;

}

**Output:**

Entered..

Just Exiting...

Entered..

Just Exiting...

**Result**:

Thus the program to implement semaphores executed successfully.

**Ex.no:5 Implementation of Shared memory and IPC**

**Date :**

**AIM:**

To demonstrate communication between process using shared memory.

**ALGORITHM:**

**Server**

Step1.Initialize size of shared memory shmsize to 27.

Step 2.Initialize Key.to 2013 (some random value).

Step 3.Create a shared memory segment using shmget with key & IPC\_CREAT as

parameter.

a.If shared memory identifier shmid is -1, then stop.

Step 4.Display shmid

Step 5.Attach server process to the shared memory using shmmat with shmid as parameter.

a.If pointer to the shared memory is not obtained, then stop.

Step 6.Clear contents of the shared region using memset function.

Step 7.Write a–z onto the shared memory.

Step 8.Wait till client reads the shared memory contents

Step 9.Detatch process from the shared memory using shmdt system call.

Step 10.Remove shared memory from the system using shmctl with IPC\_RMID

argument

Step 11.Stop the program.

**Client**

Step 1.Initialize size of shared memory shmsize to 27.

Step 2.Initialize key to 2013 (same value as in server).

Step 3.Obtain access to the same shared memory segment using same key

a.If obtained then display the shmid else print "Server not started"

Step 4. Attach client process to the shared memory using shmmat with shmid as parameter.

a.If pointer to the shared memory is not obtained, then stop.

Step 5.Read contents of shared memory and print it.

Step 6.After reading, modify the first character of shared memory to '\*'

Step 7.Stop the program.

.

**PROGRAM :**

**Server**

#include <stdio.h>

#include <stdlib.h>

#include <sys/un.h>

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#define shmsize 27

main()

{

char c;

int shmid;

key\_t key = 2013;

char \*shm, \*s;

if ((shmid = shmget(key, shmsize, IPC\_CREAT|0666)) < 0)

{

perror("shmget");

exit(1);

}

printf("Shared memory id : %d\n", shmid);

if ((shm = shmat(shmid, NULL, 0)) == (char \*) -1)

{

perror("shmat");

exit(1);

}

memset(shm, 0, shmsize);

s = shm;

printf("Writing (a-z) onto shared memory\n");

for (c = 'a'; c <= 'z'; c++)

{

\*s++ = c;

}

\*s = '\0';

while (\*shm != '\*');

printf("Client finished reading\n");

if(shmdt(shm) != 0)

{

fprintf(stderr, "Could not close memory segment.\n");

}

shmctl(shmid, IPC\_RMID, 0);

}

**Client**

#include <stdio.h>

#include <stdlib.h>

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#define shmsize 27

main()

{

int shmid;

key\_t key = 2013;

char \*shm, \*s;

if ((shmid = shmget(key, shmsize, 0666)) < 0)

{

printf("Server not started\n");

exit(1);

}

else

{

printf("Accessing shared memory id : %d\n",shmid);

}

if ((shm = shmat(shmid, NULL, 0)) == (char \*) -1)

{

perror("shmat");

exit(1);

}

printf("Shared memory contents:\n");

for (s = shm; \*s != '\0'; s++)

{

putchar(\*s);

}

putchar('\n');

\*shm = '\*';

}

**OUTPUT**

**Server side**

[root@fosslab]# gcc ex8s.c -o server

[root@fosslab]# ./server

Shared memory id : 851987

Writing (a-z) onto shared memory

Client finished reading

**Client side**

[root@fosslab]# gcc ex8c.c -o client

[root@fosslab]# ./client

Accessing shared memory id : 851987

Shared memory contents:

abcdefghijklmnopqrstuvwxyz

**RESULT:**

Thus the program to implement shared memory was executed successfully

**Ex.no :6 Bankers Algorithm for Deadlock Avoidance**

**Date :**

**AIM:**

To simulate Banker;s algorithm for Dead Lock Avoidance .

**ALGORITHM:**

1. Initialize ***Work*** = ***Available*** and ***Finish[i] = false*** for 0≤ i ≤ n-1
2. Find ***i*** such that both
   1. ***Finish[i] = false***
   2. ***Needi ≤ Work***

If no such ***i*** exists, go to step 4

1. ***Work = Work + Allocation***

***Finish[i] = true***

Go to step 2

1. If ***Finish[i] = true*** for all ***i***, then system is in SAFE state.

**PROGRAM:**

#include <stdio.h>

#include <stdlib.h>

void main()

{

int Max[10][10], need[10][10], alloc[10][10], avail[10], completed[10], safeSequence[10];

int p, r, i, j, process, count;

count = 0;

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

scanf("%d", &p);

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

{

completed[i] = 0;

}

printf("\nEnter the no of resources : ");

scanf("%d", &r);

printf("\nEnter the Max Matrix for each process : ");

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

{

printf("\nFor process %d : ", i + 1);

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

{

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

}

}

printf("\nEnter the allocation for each process : ");

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

{

printf("\nFor process %d : ",i + 1);

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

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

}

printf("\nEnter the Available Resources : ");

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

{

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

}

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

{

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

{

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

}

}

do

{

printf("\n Max matrix:\tAllocation matrix:\n");

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

{

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

{

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

}

printf("\t\t");

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

{

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

}

printf("\n");

}

process = -1;

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

{

if(completed[i] == 0)//if not completed

{

process = i ;

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

{

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

{

process = -1;

break;

}

}

}

if(process != -1)

{

break;

}

}

if(process != -1)

{

printf("\nProcess %d runs to completion!", process + 1);

safeSequence[count] = process + 1;

count++;

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

{

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

alloc[process][j] = 0;

Max[process][j] = 0;

completed[process] = 1;

}

}

}

while(count != p && process != -1);

if(count == p)

{

printf("\nThe system is in a safe state!!\n");

printf("Safe Sequence : < ");

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

{

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

}

printf(">\n");

}

else

{

printf("\nThe system is in an unsafe state!!");

}

}

**OUTPUT:**

Enter the no of processes : 5

Enter the no of resources : 3

Enter the Max Matrix for each process :

For process 1 : 7 5 3

For process 2 : 3 2 2

For process 3 : 9 0 2

For process 4 : 2 2 2

For process 5 : 4 3 3

Enter the allocation for each process :

For process 1 : 0 1 0

For process 2 : 2 0 0

For process 3 : 3 0 2

For process 4 : 2 1 1

For process 5 : 0 0 2

Enter the Available Resources : 3 3 2

Max matrix: Allocation matrix:

7 5 3 0 1 0

3 2 2 2 0 0

9 0 2 3 0 2

2 2 2 2 1 1

4 3 3 0 0 2

Process 2 runs to completion!

Max matrix: Allocation matrix:

7 5 3 0 1 0

0 0 0 0 0 0

9 0 2 3 0 2

2 2 2 2 1 1

4 3 3 0 0 2

Process 4 runs to completion!

Max matrix: Allocation matrix:

7 5 3 0 1 0

0 0 0 0 0 0

9 0 2 3 0 2

0 0 0 0 0 0

4 3 3 0 0 2

Process 1 runs to completion!

Max matrix: Allocation matrix:

0 0 0 0 0 0

0 0 0 0 0 0

9 0 2 3 0 2

0 0 0 0 0 0

4 3 3 0 0 2

Process 3 runs to completion!

Max matrix: Allocation matrix:

0 0 0 0 0 0

0 0 0 0 0 0

0 0 0 0 0 0

0 0 0 0 0 0

4 3 3 0 0 2

Process 5 runs to completion!

The system is in a safe state!!

Safe Sequence : < 2 4 1 3 5 >

**RESULT:**

Thus the program to implement Bankers Algorithm was written, executed and the output was verified successfully.

**Ex. No.: 7 Implementation of Deadlock Detection Algorithm**

**Date :**

**AIM:**

To simulate Deadlock Detection algorithm.

**ALGORITHM:**

1. Initialize ***Work*** = ***Available*** and if Allocationi ≠ 0, then ***Finish[i] = false*** for 0≤ i ≤ n-1
2. Find ***i*** such that both
   1. ***Finish[i] = false***
   2. ***Requesti ≤ Work***

If no such ***i*** exists, go to step 4

1. ***Work = Work + Allocation***

***Finish[i] = true***

Go to step 2

1. If ***Finish[i] = true*** for all ***i***, then system is in SAFE state, else UNSAFE.

**PROGRAM :**

#include <stdio.h>

#include <stdlib.h>

void main()

{

int request[10][10], alloc[10][10], avail[10], completed[10], safeSequence[10];

int p, r, i, j, process, count;

count = 0;

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

scanf("%d", &p);

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

{

completed[i] = 0;

}

printf("\nEnter the no of resources : ");

scanf("%d", &r);

printf("\nEnter the allocation for each process : ");

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

{

printf("\nFor process %d : ",i + 1);

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

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

}

printf("\nEnter the current Request for each process : ");

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

{

printf("\nFor process %d : ", i + 1);

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

{

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

}

}

printf("\nEnter the Available Resources : ");

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

{

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

}

do

{

process = -1;

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

{

if(completed[i] == 0)//if not completed

{

process = i ;

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

{

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

{

process = -1;

break;

}

}

}

if(process != -1)

{

break;

}

}

if(process != -1)

{

printf("\nProcess %d runs to completion!", process + 1);

safeSequence[count] = process + 1;

count++;

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

{

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

alloc[process][j] = 0;

completed[process] = 1;

}

}

}

while(count != p && process != -1);

if(count == p)

{

printf("\nThe system is in a safe state!!\n");

printf("Safe Sequence : < ");

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

{

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

}

printf(">\n");

}

else

{

printf("\nThe system is deadlocked!");

}

}

**OUTPUT:**

Enter the no of processes : 5

Enter the no of resources : 3

Enter the allocation for each process :

For process 1 : 0 1 0

For process 2 : 2 0 0

For process 3 : 3 0 3

For process 4 : 2 1 1

For process 5 : 0 0 2

Enter the current Request for each process :

For process 1 : 0 0 0

For process 2 : 2 0 2

For process 3 : 0 0 1

For process 4 : 1 0 0

For process 5 : 0 0 2

Enter the Available Resources : 0 0 0

Process 1 runs to completion!

The system is deadlocked!

**RESULT:**

Thus a program to implement deadlock detection was verified.

**Ex.no:8 implement Threading & Synchronization Applications**

**Date :**

**Aim:**

To Write C program to implement Threading & Synchronization Applications.

**Algorithm:**

Step1: Start the program

Step2: include necessary header files

Step3: initialize all global variables

Step4: using inbuilt thread functions perform thread and synchronization

Operations

Step5: print the output

Step 6: Stop the program

**Program:**

#include<stdio.h>

#include<string.h>

#include<pthread.h>

#include<stdlib.h>

#include<unistd.h>

pthread\_t tid[2];

int counter;

void\* trythis(void \*arg)

{

unsigned long i = 0;

counter += 1;

printf("\n Job %d has started\n", counter);

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

printf("\n Job %d has finished\n", counter);

return NULL;

}

int main(void)

{

int i = 0;

int error;

while(i < 2)

{

error = pthread\_create(&(tid[i]), NULL, &trythis, NULL);

if (error != 0)

printf("\nThread can't be created : [%s]",strerror(error));

i++;

}

pthread\_join(tid[0], NULL);

pthread\_join(tid[1], NULL);

return 0;

}

**OUTPUT:**

Job 1 has started

Job 2 has started

Job 2 has finished

Job 2 has finished

**Result:**

Thus the program to implement threading and synchronization is executed successfully and the output is verified.

**Ex.no:9(a) First Fit**

**Date:**

**Aim:**

Implement of first fit Memory Allocation Methods for fixed partition

**Algorithm:**

1. Get no. of Processes and no. of blocks.
2. After that get the size of each block and process requests.
3. Now allocate processes  
   if(block size >= process size)  
   //allocate the process  
   else  
   //move on to next block
4. Display the processes with the blocks that are allocated to a respective process.
5. Stop.

**Program:**

#include<bits/stdc++.h>

using namespace std;

// Function to allocate memory to

// blocks as per First fit algorithm

void firstFit(int blockSize[], int m,

int processSize[], int n)

{

// Stores block id of the

// block allocated to a process

int allocation[n];

// Initially no block is assigned to any process

memset(allocation, -1, sizeof(allocation));

// pick each process and find suitable blocks

// according to its size ad assign to it

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

{

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

{

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

{

// allocate block j to p[i] process

allocation[i] = j;

// Reduce available memory in this block.

blockSize[j] -= processSize[i];

break;

}

}

}

cout << "\nProcess No.\tProcess Size\tBlock no.\n";

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

{

cout << " " << i+1 << "\t\t"

<< processSize[i] << "\t\t";

if (allocation[i] != -1)

cout << allocation[i] + 1;

else

cout << "Not Allocated";

cout << endl;

}

}

// Driver code

int main()

{

int blockSize[] = {100, 500, 200, 300, 600};

int processSize[] = {212, 417, 112, 426};

int m = sizeof(blockSize) / sizeof(blockSize[0]);

int n = sizeof(processSize) / sizeof(processSize[0]);

firstFit(blockSize, m, processSize, n);

return 0 ;

}

**output:**

Input :

blockSize[] = {100, 500, 200, 300, 600};

processSize[] = {212, 417, 112, 426};

Output:

Process No. Process Size Block no.

1 212 2

2 417 5

3 112 2

4 426 Not Allocated

**Result:**

Thus first fit Memory Allocation Methods for fixed partition is executed successfully.

**Ex.no:9(b) Worst Fit**

**Date :**

**Aim:**

Implement of worst fit Memory Allocation Methods for fixed partition

**Algorithm:**

1- Input memory blocks and processes with sizes.

2- Initialize all memory blocks as free.

3- Start by picking each process and find the maximum block size that can be assigned tocurrent process

i.e., find max(bockSize[1],

blockSize[2],.....blockSize[n]) >

processSize[current]

if found then assign it to the current process.

4- If not then leave that process and keep checking the further processes.

**Program:**

// C++ implementation of worst - Fit algorithm

#include<bits/stdc++.h>

using namespace std;

// Function to allocate memory to blocks as per worst fit

// algorithm

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

{

// Stores block id of the block allocated to a

// process

int allocation[n];

// Initially no block is assigned to any process

memset(allocation, -1, sizeof(allocation));

// pick each process and find suitable blocks

// according to its size ad assign to it

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

{

// Find the best fit block for current process

int wstIdx = -1;

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

{

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

{

if (wstIdx == -1)

wstIdx = j;

else if (blockSize[wstIdx] < blockSize[j])

wstIdx = j;

}

}

// If we could find a block for current process

if (wstIdx != -1)

{

// allocate block j to p[i] process

allocation[i] = wstIdx;

// Reduce available memory in this block.

blockSize[wstIdx] -= processSize[i];

}

}

cout << "\nProcess No.\tProcess Size\tBlock no.\n";

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

{

cout << " " << i+1 << "\t\t" << processSize[i] << "\t\t";

if (allocation[i] != -1)

cout << allocation[i] + 1;

else

cout << "Not Allocated";

cout << endl;

}

}

// Driver code

int main()

{

int blockSize[] = {100, 500, 200, 300, 600};

int processSize[] = {212, 417, 112, 426};

int m = sizeof(blockSize)/sizeof(blockSize[0]);

int n = sizeof(processSize)/sizeof(processSize[0]);

worstFit(blockSize, m, processSize, n);

return 0 ;

}

OUTPUT:

Input : blockSize[] = {100, 500, 200, 300, 600};

processSize[] = {212, 417, 112, 426};

Output:

Process No. Process Size Block no.

1 212 5

2 417 2

3 112 5

4 426 Not Allocated

**Result:**

Thus worst fit Memory Allocation Methods for fixed partition is executed successfully.

**Ex.no:9(C) Best Fit**

**Date :**

**Aim:**

Implement of best fit Memory Allocation Methods for fixed partition

**Algorithm:**

1. Get no. of Processes and no. of blocks.
2. After that get the size of each block and process requests.
3. Then select the best memory block that can be allocated using the above definition.
4. Display the processes with the blocks that are allocated to a respective process.
5. Value of Fragmentation is optional to display to keep track of wasted memory.
6. Stop.

**Program:**

// C++ implementation of Best - Fit algorithm

#include<bits/stdc++.h>

using namespace std;

// Function to allocate memory to blocks as per Best fit

// algorithm

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

{

// Stores block id of the block allocated to a

// process

int allocation[n];

// Initially no block is assigned to any process

memset(allocation, -1, sizeof(allocation));

// pick each process and find suitable blocks

// according to its size ad assign to it

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

{

// Find the best fit block for current process

int bestIdx = -1;

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

{

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

{

if (bestIdx == -1)

bestIdx = j;

else if (blockSize[bestIdx] > blockSize[j])

bestIdx = j;

}

}

// If we could find a block for current process

if (bestIdx != -1)

{

// allocate block j to p[i] process

allocation[i] = bestIdx;

// Reduce available memory in this block.

blockSize[bestIdx] -= processSize[i];

}

}

cout << "\nProcess No.\tProcess Size\tBlock no.\n";

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

{

cout << " " << i+1 << "\t\t" << processSize[i] << "\t\t";

if (allocation[i] != -1)

cout << allocation[i] + 1;

else

cout << "Not Allocated";

cout << endl;

}

}

// Driver code

int main()

{

int blockSize[] = {100, 500, 200, 300, 600};

int processSize[] = {212, 417, 112, 426};

int m = sizeof(blockSize)/sizeof(blockSize[0]);

int n = sizeof(processSize)/sizeof(processSize[0]);

bestFit(blockSize, m, processSize, n);

return 0 ;

}

**OUTPUT:**

Input :

blockSize[] = {100, 500, 200, 300, 600};

processSize[] = {212, 417, 112, 426};

Output:

Process No. Process Size Block no.

1 212 4

2 417 2

3 112 3

4 426 5

**Result:**

Thus Best Fit Memory Allocation Methods for fixed partition is executed successfully.

**Ex. No:10** PAGING TECHNIQUE OF MEMORY MANAGEMENT

Date :

**AIM:**

To simulate paging to convert logical address to physical address

**ALGORITHM:**

Step 1: Read all the necessary input from the keyboard.

Step 2: Pages - Logical memory is broken into fixed - sized blocks.

Step 3: Frames – Physical memory is broken into fixed – sized blocks.

Step 4: Calculate the physical address using the following

Physical address = ( Frame number \* Frame size ) + offset

Step 5: Display the physical address.

Step 6: Stop the process.

**PROGRAM**

#include <stdio.h>

#include <math.h>

void main()

{

int memsize = 0, pgsize = 0, memaddrBits, offsetBits;

int pgCount = 0, pgNoBits;

int i, j, pgTable[32], logicalAddr[32];

int frameNo = 0, pgNo = 0, physicalAddr = 0, offset = 0;

printf("\nEnter memory size (as a power of 2): ");

scanf("%d", &memaddrBits);

printf("\nEnter page size (as a power of 2): ");

scanf("%d", &offsetBits);

pgNoBits = memaddrBits - offsetBits;

memsize = pow(2, memaddrBits);

pgsize = pow(2, offsetBits);

pgCount = pow(2, pgNoBits);

printf("\n\t\t\tNumber of Address bits");

printf("\nMemory\t\t%d Bytes\t%d", memsize, memaddrBits);

printf("\nPage\t\t%d Bytes\t\t%d", pgsize, pgNoBits);

printf("\nNo. of pages:\t%d pages\t\t%d", pgCount, offsetBits);

printf("\nEnter the page table entry for %d pages:\n", pgCount);

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

{

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

}

printf("\nEnter logical address in binary form(%d bits): ",memaddrBits );

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

{

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

}

for (i = (pgNoBits-1), j = 0; i >= 0; i--, j++ )

{

pgNo = pgNo + logicalAddr[i]\*pow(2, j);

}

printf("\nPAGE TABLE");

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

{

printf("\n%d-%d", i, pgTable[i]);

}

printf("\n\nPgNo:%d", pgNo);

frameNo = pgTable[pgNo];

printf("\nFrame Number: %d", frameNo);

for ( i = (memaddrBits - 1), j = 0; i >= pgNoBits; i--, j++)

{

offset = offset + logicalAddr[i]\*pow(2,j);

}

printf("\nOffset: %d", offset);

physicalAddr = (frameNo \* pgsize) + offset;

printf("\nPhysical Address: %d", physicalAddr);

}

**OUTPUT**

[root@fosslab]# gcc ex9.c -lm

[root@fosslab]# ./a.out

Enter memory size (as a power of 2): 4

Enter page size (as a power of 2): 2

Number of Address bits

Memory 16 Bytes 4

Page 4 Bytes 2

No. of pages: 4 pages 2

Enter the page table entry for 4 pages:

5

6

1

2

Enter logical address in binary form(4 bits): 0 1 1 1

PAGE TABLE

0-5

1-6

2-1

3-2

PgNo: 1

Frame Number: 6

Offset: 3

Physical Address: 27

**RESULT:**

Thus the program to simulate paging was implemented

**Ex. No :11**  PAGE REPLACEMENT ALGORITHMS

**Date  :**

**AIM:**

To implement page replacement algorithms FIFO, LRU and LFU.

**ALGORITHM:**

FIFO

* Choose the page that first came in as the victim page.

LRU

* Increment a clock variable for every reference.
* Copy the clock to a counter variable for every frame.
* Choose the frame with least counter value.

LFU

* Increment a counter for each page.
* Choose the page with least counter.

**PROGRAM :**

#include<stdio.h>

int frameCount, refCount, memRef[30], frames[10], frCounter[30];

int refSearch(int);

void FIFO();

void LRU();

void LFU();

void main()

{

int i;

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

scanf("%d", &frameCount);

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

scanf("%d", &refCount);

printf("Enter the memory reference string: ");

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

{

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

}

FIFO();

LRU();

LFU();

}

void **FIFO**()

{

int i, pgFault = 0;

int j = 0, k;

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

{

frames[i] = -1;

}

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

{

if (refSearch(memRef[i]) == 0)

{

frames[j] = memRef[i];

j = (j+1) % frameCount;

pgFault++;

}

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

{

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

}

printf("\n");

}

printf("\nPage Fault Count: %d", pgFault);

}

void **LRU**()

{

int i, j, k, pgFault = 0, searchIndex, least = 0, leastIndex = 0, clock = 0;

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

{

frames[i] = -1;

frCounter[i] = 0;

}

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

{

searchIndex = refSearch(memRef[i]);

if (searchIndex == 0)

{

//Find the least value in frCounter

least = frCounter[0];

leastIndex = 0;

for (j = 1; j < frameCount; j++)

{

if (least > frCounter[j])

{

least = frCounter[j];

leastIndex = j;

}

}

frames[leastIndex] = memRef[i];

clock++;

frCounter[leastIndex] = clock;

pgFault++;

}

else

{

clock++;

frCounter[searchIndex] = clock;

}

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

{

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

}

printf("\n");

}

printf("\nPage Fault Count: %d", pgFault);

}

int **refSearch**(int ref)

{

int i;

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

{

if (ref == frames[i])

{

return i;

}

}

return 0;

}

void **LFU**()

{

int i, j, k, pgFault = 0, searchIndex, least = 0, leastIndex = 0;

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

{

frames[i] = -1;

frCounter[i] = 0;

}

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

{

searchIndex = refSearch(memRef[i]);

if (searchIndex == 0)

{

//Find the least value in frCounter

least = frCounter[0];

leastIndex = 0;

for (j = 1; j < frameCount; j++)

{

if (least > frCounter[j])

{

least = frCounter[j];

leastIndex = j;

}

}

frames[leastIndex] = memRef[i];

frCounter[leastIndex] = 1;

pgFault++;

}

else

{

frCounter[searchIndex]++;

}

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

{

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

}

printf("\n");

}

printf("\nPage Fault Count: %d", pgFault);

}

**OUTPUT:**

Enter the number of frames: 3

Enter the number of memory references: 20

Enter the memory reference string: 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

FIFO

7 -1 -1

7 0 -1

7 0 1

2 0 1

2 0 1

2 3 1

2 3 0

4 3 0

4 2 0

4 2 3

0 2 3

0 2 3

0 2 3

0 1 3

0 1 2

0 1 2

0 1 2

0 7 2

0 7 0

1 7 0

Page Fault Count: 16

LRU

7 -1 -1

7 0 -1

7 0 1

2 0 1

2 0 1

2 0 3

2 0 3

4 0 3

4 0 2

4 3 2

0 3 2

0 3 2

0 3 2

1 3 2

1 3 2

1 0 2

1 0 2

1 0 7

1 0 7

1 0 7

Page Fault Count: 14

LFU

7 -1 -1

7 0 -1

7 0 1

2 0 1

2 0 1

3 0 1

3 0 1

4 0 1

2 0 1

3 0 1

3 0 1

3 0 1

2 0 1

2 0 1

2 0 1

2 0 1

2 0 1

7 0 1

7 0 1

7 0 1

Page Fault Count: 12

**RESULT:**

Thus, the program to implement page allocation algorithms was executed successfully.

**Ex. No :12**  **Implementation of the various File Organization Techniques**

**Date  :**

**Aim:**

To Implementation of the various File Organization Techniques

a. Single level Directory

b. Two level Directory

c. Hierarchical level Directory

**Hierarchical level Directory**

**Algorithm:**

Step 1: Start the Program

Step 2: Define structure and declare structure variables

Step 3: start main and declare variables

Step 4: Check a directory treestructure

Step 5: Display the directory tree in graphical mood

Step 6: Stop the execution

**Program:**

#include <stdio.h>

#include <graphics.h>

struct tree\_element

{

char name[20];

int x,y,ftype,lx,rx,nc,level;

struct tree\_element \*link[5];

};

typedef struct tree\_element node;

void main()

{

int gd=DETECT,gm;

node \*root;

root=NULL;

clrscr();

create(&root,0,"root",0,639,320);

clrscr();

initgraph(&gd,&gm,"c:\tc\BGI");

display(root);

getch();

closegraph();

}

create(node \*\*root,int lev,char \*dname,int lx,int rx,int x)

{

int i,gap;

if(\*root==NULL)

{

(\*root)=(node \*)malloc(sizeof(node));

printf("Enter name of dir/file(under %s) : ",dname);

fflush(stdin);

root)->name);

printf("enter 1 for Dir/2 for file :");

scanf("%d",&(\*root)->ftype);

(\*root)->level=lev;

(\*root)->y=50+lev\*50;

(\*root)->x=x;

(\*root)->lx=lx;

(\*root)->rx=rx;

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

(\*root)->link[i]=NULL;

if((\*root)->ftype==1)

{

printf("No of sub directories/files(for %s):",(\*root)->name);

scanf("%d",&(\*root)>nc);

if((\*root)->nc==0)

gap=rx-lx;

else

gap=(rx-lx)/(\*root)->nc;

for(i=0;i< (\*root)->nc;i++) create(&((\*root)>link[i]),lev+1,(\*root)>name,lx+gap\*i,lx+gap\*i+gap,lx+gap\*i+gap/2);

}

else

(\*root)->nc=0;

}

}

display(node \*root)

{

int i;

settextstyle(2,0,4);

settextjustify(1,1);

setfillstyle(1,BLUE);

setcolor(14);

if(root !=NULL)

{

for(i=0;i< root->nc;i++)

{

line(root->x,root->y,root->link[i]->x,root->link[i]->y);

}

if(root->ftype==1)

bar3d(root->x-20,root->y-10,root->x+20,root>y+10,0,0);

else

fillellipse(root->x,root->y,20,20);

outtextxy(root->x,root->y,root->name);

for(i=0;inc;i++)

{

display(root->link[i]);

}

}

}

**Output:**

**Two level Directory:**

**ALGORITHM:**

Step 1: Start the Program

Step 2: Initialize structure elements

Step 3: Start main function

Step 4: Set variables gd =DETECT, gm;

Step 5: Create structure using create (&root,0,”null”,0,639,320);

Step 6: initgraph(&gd,&gm,”c:\tc\bgi”);

Step 7: Stop the execution

**Program:**

#include<stdio.h>

#include <graphics.h>

struct tree\_element

{

char name[20];

int x,y,ftype,lx,rx,nc,level;

struct tree\_element \*link[5];

};

typedef truct tree\_element node;

void main()

{

int gd=DETECT,gm;

node \*root;

root=NULL;

clrscr();

create(&root,0,"null",0,639,320);

clrscr();

initgraph(&gd,&gm,"c:\tc\bgi");

display(root);

getch();

closegraph();

}

create(node \*\*root,int lev,char \*dname,int lx,int rx,int x)

{

int i,gap;

if(\*root==NULL)

{

(\*root)=(node\*)malloc(sizeof(node));

printf("enter name of dir/file(under %s):",dname);

fflush(stdin);

gets((\*root)->name);

if(lev==0||lev==1)

(\*root)->ftype=1;

else

(\*root)->ftype=2;

(\*root)->level=lev;

(\*root)->y=50+lev\*50;

(\*root)->x=x;

(\*root)->lx=lx;

(\*root)->rx=rx;

for(i=0;ilink[i]=NULL;

if((\*root)->ftype==1)

{

if(lev==0||lev==1)

{

if((\*root)->level==0)

printf("How many users");

else

printf("hoe many files");

printf("(for%s):",(\*root)->name);

scanf("%d",&(\*root)->nc);

}

else

(\*root)->nc=0;

if((\*root)->nc==0)

gap=rx-lx;

else

gap=(rx-lx)/(\*root)->nc;

for(i=0;inc;i++)

create(&((\*root)>link[i]),lev+1,(\*root)>name,lx+gap\*i,lx+gap\*i+gap,lx+gap\*i+gap/2);

}

Else

(\*root)->nc=0;

}

}

display(node \*root)

{

int i;

settextstyle(2,0,4);

settextjustify(1,1);

setfillstyle(1,BLUE);

35

setcolor(14);

if(root!=NULL)

{

for(i=0;i< root->nc;i++)

{

line(root->x,root->y,root->link[i]->x,root->link[i]->y);

}

if(root->ftype==1)

bar3d(root->x-20,root->y-10,root->x+20,roo>y+10,0,0);

else

fillellipse(root->x,root->y,20,20);

outtextxy(root->x,root->y,root->name);

for(i=0;i<root->nc;i++)

{

display(root->link[i]);

}

}

}

**Output:**

**Single level Directory**

**ALGORITHM**:

Step 1: Start the Program

Step 2: Initialize values gd=DETECT,gm,count,i,j,mid,cir\_x.

Step 3: Initialize graph function

Step 4: Set back ground color with setbkcolor();

Step 5: Read number of files in variable count.

Step 6: check i<count; mid=640/count;

Step 7: Stop the execution

**Program**:

#include<stdio.h>

#include<conio.h>

#include <stdlib.h>

#include<graphics.h>

void main()

{

int gd=DETECT,gm,count,i,j,mid,cir\_x;

char fname[10][20];

clrscr();

initgraph(&gd,&gm,"c:\tc\bgi");

cleardevice();

setbkcolor(GREEN);

puts("Enter no of files do u have?");

scanf("%d",&count);

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

{

cleardevice();

setbkcolor(GREEN);

printf("Enter file %d name",i+1);

scanf("%s",fname[i]);

setfillstyle(1,MAGENTA);

mid=640/count;

cir\_x=mid/3;

bar3d(270,100,370,150,0,0);

settextstyle(2,0,4);

settextjustify(1,1);

outtextxy(320,125,"Root Directory");

setcolor(BLUE);

for(j=0;j< =i;j++,cir\_x+=mid)

{

line(320,150,cir\_x,250);

fillellipse(cir\_x,250,30,30);

outtextxy(cir\_x,250,fname[j]);

} getch();

} }

**Output:**

**Result:**

Thus the programs was executed successfully

**Ex. No :13**  **Implementation of the following File Allocation Strategies**

**Date  :**

**Aim :**

Implementation of the following File Allocation Strategies

a) Sequential

b) Indexed

c) Linked

**OBJECTIVE:**

To implement sequential file allocation technique.

**ALGORITHM:**

Step 1: Start the program.

Step 2: Get the number of files.

Step 3: Get the memory requirement of each file.

Step 4: Allocate the required locations to each in sequential order.

a). Randomly select a location from available location s1= random(100);

b). Check whether the required locations are free from the selected location.

c). Allocate and set flag=1 to the allocated locations.

Step 5: Print the results fileno, length , Blocks allocated.

Step 6: Stop the program.

**Program**:

#include<stdio.h>

main()

{

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

clrscr();

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

f[i]=0;

X:

printf("\n Enter the starting block & length of file"); scanf("%d%d",&st,&len);

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

if(f[j]==0)

{

f[j]=1;

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

}

else

{

printf("Block already allocated");

break;

}

if(j==(st+len))

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

printf("\n if u want to enter more files?(y-1/n-0)");

scanf("%d",&c);

if(c==1)

goto X;

else

exit();

getch();

}

Output: Enter the starting block & length of file 4 10

4->1

5->1

6->1

7->1

8->1

9->1

10->1

11->1

12->1

13->1

The file is allocated to disk

If you want to enter more files? (Y-1/N-0)

**OBJECTIVE:**

To write a C program to implement File Allocation concept using the technique Linked List Technique.

**ALGORITHM:**

Step 1: Start the Program

Step 2: Get the number of files.

Step 3: Allocate the required locations by selecting a location randomly

Step 4: Check whether the selected location is free.

Step 5: If the location is free allocate and set flag =1 to the allocated locations.

Step 6: Print the results file no, length, blocks allocated.

Step 7: Stop the execution

**Program:**

#include<stdio.h>

main()

{

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

clrscr();

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

f[i]=0;

printf("Enter how many blocks that are already allocated"); scanf("%d",&p);

printf("\nEnter the blocks no.s that are already allocated"); for(i=0;i<p;i++)

{

scanf("%d",&a);

f[a]=1;

}

X:

printf("Enter the starting index block & length"); scanf("%d%d",&st,&len);

k=len;

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

{

if(f[j]==0)

{

f[j]=1;

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

}

else

{

printf("\n %d->file is already allocated",j);

k++;

}

} printf("\n If u want to enter one more file? (yes-1/no-0)");

scanf("%d",&c);

if(c==1)

goto X;

else

exit();

getch( );

}

**OUTPUT:**

Enter how many blocks are already allocated 3

Enter the blocks no’s that are already allocated 4 7 9 Enter the starting index block & length 3

7

3-> 1

4-> File is already allocated

5->1

6->1

7-> File is already allocated

8->1

9-> File is already allocated

10->1

11->1

12->1

If u want to enter one more file? (yes-1/no-0)

**OBJECTIVE:**

To write a C program to implement File Allocation concept using the technique indexed allocation Technique

**ALGORITHM:**

Step 1: Start the Program

Step 2: Get the number of files.

Step 3: Get the memory requirement of each file.

Step 4: Allocate the required locations by selecting a location randomly.

Step 5: Print the results file no,length, blocks allocated.

Step 7: Stop the execution.

**Program:**

#include<stdio.h>

int f[50],i,k,j,inde[50],n,c,count=0,p;

main()

{

clrscr();

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

f[i]=0;

x:

printf("enter index block\t");

scanf("%d",&p);

if(f[p]==0)

{

f[p]=1;

printf("enter no of files on index\t");

scanf("%d",&n);

}

else

{

printf("Block already allocated\n");

goto x;

}

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

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

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

if(f[inde[i]]==1)

{

printf("Block already allocated");

goto x;

}

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

f[inde[j]]=1;

printf("\n allocated");

printf("\n file indexed");

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

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

printf(" Enter 1 to enter more files and 0 to exit\t"); scanf("%d",&c);

if(c==1)

goto x;

else

exit();

getch();

}

**OUTPUT:**

Enter index block 9

Enter no of files on index 3

1 2 3

Allocated

File indexed

9-> 1:1

9-> 2:1

9->3:1

Enter 1 to enter more files and 0 to exit

**Result:**

Thus the programs are executed successfully.