**9.a)**

#include <stdio.h>

int main() {

int a, i;

unsigned long long fact = 1; // Use long long for big factorials

printf("Enter a number: ");

scanf("%d", &a);

if (a < 0) {

printf("Factorial is not defined for negative numbers.\n");

} else {

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

fact \*= i;

}

printf("Factorial of %d = %llu\n", a, fact);

}

return 0;

}

**O/P**:-

Enter a number: 5

Factorial of 5 = 120

**9.b):-PARENT**

#include<stdlib.h>

#include<string.h>

int global=10;

void main()

{

int local=20;

pid\_t pid;

printf("before fork\n");

printf("pid=%d,global=%d,local=%d\n",getpid(),global,local);

pid=fork();

if(pid<0)

printf("failed to create child");

else if(pid==0)

{

printf("after fork\n");global++;

local++;

}

else if(pid>0)

sleep(2);

printf("cid=%d,global=%d,local=%d\n",getpid(),global,local);

exit(0);

}

**O/P:-**

before fork

pid=4437,global=10,local=20

after fork

cid=4438,global=11,local=21

cid=4437,global=10,local=20

**9.c):-PIPE**

#include <stdio.h>

#include<stdlib.h>

#include<unistd.h>

#include <sys/wait.h> /\* contains prototype for wait \*/

int main(void){

int pid;

int status;

printf("Hello World!\n");

pid = fork( );

if(pid == -1) /\* check for error in fork \*/

{

perror("bad fork");

exit(1);

}

if (pid == 0)

printf("I am the child process.\n");

else { wait(&status); /\* parent waits for child to finish \*/

printf("I am the parent process.\n");

return 0;

}

}

**o/p:-**

Hello World!

I am the child process.

I am the parent process**.**

**VIVA:-**

**1. What is Shell Script?**

A Shell Script is a text file that contains one or more commands.

**2. What are the different type of variables used in a shell Script?**

In Linux shell script we can use two types of variables :

o System defined variables

o User defined variables

**3. Why are we using fork() function call?**

System call fork() is used to create processes. It takes no arguments and returns a process ID. The purpose of fork() is to create a *new* process, which becomes the *child* process of the caller. After a new child process is created, *both* processes will execute the next instruction following the *fork()* system call.

4. Why do we create a child process?

Sometimes there is a need for a program to perform more than one function simultaneously. Since these jobs may be interrelated so two different programs to perform them cannot be created.

**5. Why do we use piping?**

A pipe is a form of redirection (transfer of standard output to some other destination) that is used in Linux and other Unix-like operating systems to send the output of one command/program/process to another command/program/process for further processing.

**6. what is IPC?**

Inter-process communication (IPC) is a mechanism that allows processes to communicate with each other and synchronize their actions. The communication between these processes can be seen as a method of co-operation between them.

**10.A):- FCFS**

#include <stdio.h>

int main() {

int pid[10], bt[10], wt[10], tat[10], n, twt = 0, ttat = 0, i;

float awt, atat;

printf("Enter number of processes: ");

scanf("%d", &n);

printf("Enter burst times for %d processes:\n", n);

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

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

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

}

wt[0] = 0;

tat[0] = bt[0];

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

wt[i] = tat[i - 1];

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

}

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

twt += wt[i];

ttat += tat[i];

}

printf("\nPID\tBT\tWT\tTAT\n");

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

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

}

awt = (float)twt / n;

atat = (float)ttat / n;

printf("\nAverage Waiting Time: %.2f\n", awt);

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

return 0;

}

**O/P:-**

Enter number of processes: 3

Enter burst times for 3 processes:

Process 1: 1

Process 2: 2

Process 3: 3

PID BT WT TAT

1 1 0 1

2 2 1 3

3 3 3 6

Average Waiting Time: 1.33

Average Turnaround Time: 3.33

**10.B):- PRIORITY**

#include <stdio.h>

int main() {

int pid[10], bt[10], pr[10], wt[10], tat[10], n;

int twt = 0, ttat = 0, i, j, temp;

float awt, atat;

printf("Enter number of processes: ");

scanf("%d", &n);

printf("Enter burst times:\n");

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

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

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

}

printf("Enter process IDs:\n");

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

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

}

printf("Enter priorities (lower number = higher priority):\n");

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

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

}

// Sort processes by priority using simple bubble sort

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

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

if (pr[i] > pr[j]) {

// Swap priority

temp = pr[i]; pr[i] = pr[j]; pr[j] = temp;

// Swap burst time

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

// Swap process ID

temp = pid[i]; pid[i] = pid[j]; pid[j] = temp;

}

}

}

// Calculate Waiting Time and Turnaround Time

wt[0] = 0;

tat[0] = bt[0];

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

wt[i] = tat[i - 1];

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

}

// Total WT and TAT

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

twt += wt[i];

ttat += tat[i];

}

// Display the result

printf("\nPID\tPriority\tBT\tWT\tTAT\n");

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

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

}

awt = (float)twt / n;

atat = (float)ttat / n;

printf("\nAverage Waiting Time: %.2f\n", awt);

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

return 0;

}

**O/P:-**

Enter number of processes: 2

Enter burst times:

P1: 1

P2: 2

Enter process IDs:

3

3

Enter priorities (lower number = higher priority):

2

3

PID Priority BT WT TAT

3 2 1 0 1

3 3 2 1 3

Average Waiting Time: 0.50

Average Turnaround Time: 2.00

**VIVA:-**

**1.What is an Operating system?**

Operating System (OS), program that manages a computer's resources, especially the allocation of those resources among other programs. Typical resources include the central processing unit (CPU), computer memory, file storage, input/output (I/O) devices, and network connections.

**2.What is a process ?**

A process is an ‘active’ entity, instead of a program, which is considered a ‘passive’ entity. A single program can create many processes when run multiple times; for example, when we open a .exe or binary file multiple times, multiple instances begin (multiple processes are created)

**3.What Are The Advantages of A Multiprocessor System?**

The advantages of the multiprocessing system are: Increased Throughput − By increasing the number of processors, more work can be completed in a unit time. Cost Saving − Parallel system shares the memory, buses, peripherals etc. Multiprocessor system thus saves money as compared to multiple single systems.

**4. Explain starvation and Aging**

Starvation is the problem that occurs when high priority processes keep executing and low priority processes get blocked for indefinite time. In heavily loaded computer system, a steady stream of higher-priority processes can prevent a low-priority process from ever getting the CPU.

Aging is a technique of gradually increasing the priority of processes that wait in the system for a long time. For example, if priority range from 127(low) to 0(high), we could increase the priority of a waiting process by 1 Every 15 minutes.

**5.What are the functions of operating system?**

An operating system has three main functions: (1) manage the computer's resources, such as the central processing unit, memory, disk drives, and printers, (2) establish a user interface, and (3) execute and provide services for applications software.

**11). Write C programs to simulate the following file allocation strategies**

**a) Contiguous**

**b) Linked**

**c) Indexed**

**a) Contiguous/Sequential**

#include <stdio.h>

int main() {

int n, i, j, b[20], sb[20], t[20], x, c[20][20];

printf("Enter number of files: ");

scanf("%d", &n);

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

printf("Enter number of blocks occupied by file %d: ", i + 1);

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

printf("Enter the starting block of file %d: ", i + 1);

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

t[i] = sb[i];

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

c[i][j] = sb[i]++;

}

printf("\nFilename\tStart block\tLength\n");

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

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

printf("Enter file number to display: ");

scanf("%d", &x);

if(x > 0 && x <= n) {

printf("\nFile name is: %d", x);

printf("\nLength is: %d", b[x - 1]);

printf("\nBlocks occupied:");

for(i = 0; i < b[x - 1]; i++)

printf(" %d", c[x - 1][i]);

printf("\n");

} else {

printf("Invalid file number!\n");

}

return 0;

}

**b) Linked**

#include <stdio.h>

struct file {

char fname[10];

int start;

int size;

int block[10];

};

int main() {

struct file f[10];

int i, j, n;

printf("Enter number of files: ");

scanf("%d", &n);

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

printf("Enter file name: ");

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

printf("Enter starting block: ");

scanf("%d", &f[i].start);

f[i].block[0] = f[i].start;

printf("Enter number of blocks: ");

scanf("%d", &f[i].size);

printf("Enter block numbers: ");

for(j = 1; j < f[i].size; j++) {

scanf("%d", &f[i].block[j]);

}

}

printf("\nFile\tStart\tSize\tBlocks\n");

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

printf("%s\t%d\t%d\t", f[i].fname, f[i].start, f[i].size);

for(j = 0; j < f[i].size - 1; j++) {

printf("%d--->", f[i].block[j]);

}

printf("%d\n", f[i].block[j]);

}

return 0;

}

**c) Indexed**

#include <stdio.h>

int main() {

int n, m[20], i, j, sb[20], b[20][20], x;

// Get number of files

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

scanf("%d", &n);

// Collect file info

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

printf("\nEnter index block of File %d: ", i + 1);

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

printf("Enter number of blocks for File %d: ", i + 1);

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

printf("Enter the block numbers for File %d:\n", i + 1);

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

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

}

}

// Display file index info

printf("\nFile\tIndex Block\tLength\n");

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

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

}

// Select a file to display its blocks

printf("\nEnter the file number to display its blocks (1 to %d): ", n);

scanf("%d", &x);

if (x < 1 || x > n) {

printf("Invalid file number entered.\n");

return 1;

}

printf("\nFile Number: %d", x);

printf("\nIndex Block: %d", sb[x - 1]);

printf("\nBlocks occupied:");

for (j = 0; j < m[x - 1]; j++) {

printf(" %d", b[x - 1][j]);

}

printf("\n");

return 0;

}

**VIVA**

1. **List File access methods**?

There are three ways to access a file into a computer system: Sequential-Access, Direct Access, Index sequential Method.

· Sequential Access – It is the simplest access method. ..

. Direct Access – Another method is direct access method also known as relative access method.

. Index sequential method

1. **Explain Sequential file allocation method?**

In this allocation strategy, each file occupies a set of contiguously blocks on the disk. This strategy is best suited. For sequential files, the file allocation table consists of a single entry for each file. It shows the filenames, starting block of the file and size of the file.

1. **Explain Linked file allocation method?**

Each file is a linked list of disk blocks which need not be contiguous. The disk blocks canbe scattered anywhere on the disk. The directory entry contains a pointer to the starting and the ending file block. Each block contains a pointer to the next block occupied by the file.

1. **Explain Indexed file allocation method?**

Indexed allocation scheme stores all the disk pointers in one of the blocks called as indexed block. Indexed block doesn't hold the file data, but it holds the pointers to all the disk blocks allocated to that particular file. Directory entry will only contain the index block address.

1. **What is file system mounting?**

Mounting a file system attaches that file system to a directory (mount point) and makes it available to the system. The root ( / ) file system is always mounted. Any other file system can be connected or disconnected from the root ( / ) file system.

**11). Round Robin Algorithm.**

**#include <stdio.h>**

**int main() {**

**int ts, bt1[10], wt[10], tat[10], i, j = 0, n, bt[10], ttat = 0, twt = 0, tot = 0;**

**float awt, atat;**

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

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

**printf("Enter the timeslice: ");**

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

**printf("Enter the burst time for each process:\n");**

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

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

**scanf("%d", &bt1[i]);**

**bt[i] = bt1[i];**

**}**

**while (j < n) {**

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

**if (bt[i] > 0) {**

**if (bt[i] >= ts) {**

**tot += ts;**

**bt[i] -= ts;**

**if (bt[i] == 0) {**

**j++;**

**tat[i] = tot;**

**}**

**} else {**

**tot += bt[i];**

**bt[i] = 0;**

**j++;**

**tat[i] = tot;**

**}**

**}**

**}**

**}**

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

**wt[i] = tat[i] - bt1[i];**

**twt += wt[i];**

**ttat += tat[i];**

**}**

**awt = (float)twt / n;**

**atat = (float)ttat / n;**

**printf("\nPID\tBT\tWT\tTAT\n");**

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

**printf("P%d\t%d\t%d\t%d\n", i + 1, bt1[i], wt[i], tat[i]);**

**}**

**printf("\nAverage Waiting Time = %.2f", awt);**

**printf("\nAverage Turnaround Time = %.2f\n", atat);**

**return 0;**

**11 b). Shortest job first algorithm.**

**#include <stdio.h>**

**int main() {**

**int pid[10], bt[10], wt[10], tat[10];**

**int n, twt = 0, ttat = 0, i, j, t;**

**float awt, atat;**

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

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

**printf("Enter burst times for each process:\n");**

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

**printf("Burst time for process %d: ", i + 1);**

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

**}**

**printf("Enter process IDs:\n");**

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

**printf("PID for process %d: ", i + 1);**

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

**}**

**// Sort processes by burst time using bubble sort**

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

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

**if (bt[i] > bt[j]) {**

**// Swap burst times**

**t = bt[i];**

**bt[i] = bt[j];**

**bt[j] = t;**

**// Swap corresponding PIDs**

**t = pid[i];**

**pid[i] = pid[j];**

**pid[j] = t;**

**}**

**}**

**}**

**// Compute waiting and turnaround times**

**wt[0] = 0;**

**tat[0] = bt[0];**

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

**wt[i] = tat[i - 1];**

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

**}**

**// Totals for averages**

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

**twt += wt[i];**

**ttat += tat[i];**

**}**

**awt = (float)twt / n;**

**atat = (float)ttat / n;**

**// Display results**

**printf("\nPID\tBT\tWT\tTAT\n");**

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

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

**}**

**printf("\nAverage Waiting Time: %.2f", awt);**

**printf("\nAverage Turnaround Time: %.2f\n", atat);**

**return 0;**

**}**

**}**

**VIVA:**

**1. List different CPU Scheduling algorithms.**

The different CPU algorithms are:

· First Come First Serve.

· Shortest Job First.

· Shortest Remaining Time First.

· Round Robin Scheduling.

· Priority Scheduling.

· Multilevel Queue Scheduling.

· Multilevel Feedback Queue Scheduling.

**2. What is FCFS Scheduling?**

First Come First Serve (FCFS) is an operating system scheduling algorithm that automatically executes queued requests and processes in order of their arrival. It is the easiest and simplest CPU scheduling algorithm. In this type of algorithm, processes which requests the CPU first get the CPU allocation first.

**3. What is SJF Scheduling?**

Shortest Job First (SJF) is an algorithm in which the process having the smallest execution time is chosen for the next execution. This scheduling method can be preemptive or non-preemptive. It significantly reduces the average waiting time for other processes awaiting execution.

**4. What is RR Scheduling?**

Round Robin is a CPU scheduling algorithm where each process is assigned a fixed time

slot in a cyclic way. It is simple, easy to implement, and starvation-free as all processes get

fair share of CPU. One of the most commonly used technique in CPU scheduling as a core.

**5. What is Priority Scheduling?**

Priority Scheduling is a method of scheduling processes that is based on priority. In this

algorithm, the scheduler selects the tasks to work as per the priority. The processes with

higher priority should be carried out first, whereas jobs with equal priorities are carried out

on a round-robin or FCFS basis.

**12A).**

**#include <stdio.h>**

**#include <stdbool.h>**

**#define MAX\_FILES 20**

**#define MAX\_BLOCKS 100**

**int main() {**

**int n, i, j;**

**int length[MAX\_FILES], start[MAX\_FILES], blocks[MAX\_FILES][MAX\_BLOCKS];**

**bool allocated[MAX\_BLOCKS] = {false}; // Track if a block is already allocated**

**int fileNo;**

**printf("Enter number of files (max %d): ", MAX\_FILES);**

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

**if(n > MAX\_FILES || n <= 0) {**

**printf("Invalid number of files.\n");**

**return 1;**

**}**

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

**printf("\nEnter details for File %d:\n", i + 1);**

**printf("Enter number of blocks required: ");**

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

**printf("Enter starting block: ");**

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

**// Check if blocks are available**

**bool conflict = false;**

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

**int blockNum = start[i] + j;**

**if(blockNum >= MAX\_BLOCKS || allocated[blockNum]) {**

**conflict = true;**

**break;**

**}**

**}**

**if(conflict) {**

**printf("Error: Blocks from %d to %d are not available. Try again.\n", start[i], start[i] + length[i] - 1);**

**i--; // Repeat this file's input**

**continue;**

**}**

**// Allocate blocks**

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

**blocks[i][j] = start[i] + j;**

**allocated[blocks[i][j]] = true;**

**}**

**}**

**// Display allocation table**

**printf("\n%-10s %-15s %-10s %-20s\n", "File#", "Start Block", "Length", "Blocks Allocated");**

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

**printf("%-10d %-15d %-10d ", i + 1, start[i], length[i]);**

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

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

**}**

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

**}**

**// Query file by number**

**printf("\nEnter file number to view its block allocation: ");**

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

**if(fileNo < 1 || fileNo > n) {**

**printf("Invalid file number.\n");**

**return 1;**

**}**

**printf("\nFile %d information:\n", fileNo);**

**printf("Start Block: %d\n", start[fileNo - 1]);**

**printf("Length: %d\n", length[fileNo - 1]);**

**printf("Blocks: ");**

**for(i = 0; i < length[fileNo - 1]; i++) {**

**printf("%d ", blocks[fileNo - 1][i]);**

**}**

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

**return 0;**

**}**

**12B).**

**#include <stdio.h>**

**#define MAX\_FILES 10**

**#define MAX\_BLOCKS 10**

**struct File {**

**char fname[20];**

**int start;**

**int size;**

**int blocks[MAX\_BLOCKS];**

**};**

**int main() {**

**struct File f[MAX\_FILES];**

**int n, i, j;**

**// Input: number of files**

**printf("Enter number of files (max %d): ", MAX\_FILES);**

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

**if (n <= 0 || n > MAX\_FILES) {**

**printf("Invalid number of files.\n");**

**return 1;**

**}**

**// Input file details**

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

**printf("\nEnter details for File %d:\n", i + 1);**

**printf("Enter file name: ");**

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

**printf("Enter starting block: ");**

**scanf("%d", &f[i].start);**

**f[i].blocks[0] = f[i].start;**

**printf("Enter number of blocks: ");**

**scanf("%d", &f[i].size);**

**if (f[i].size <= 0 || f[i].size > MAX\_BLOCKS) {**

**printf("Invalid size. Try again.\n");**

**i--;**

**continue;**

**}**

**printf("Enter remaining %d block numbers (excluding start):\n", f[i].size - 1);**

**for (j = 1; j < f[i].size; j++) {**

**scanf("%d", &f[i].blocks[j]);**

**}**

**}**

**// Output file details**

**printf("\n%-10s %-10s %-10s %-20s\n", "File", "Start", "Size", "Blocks (Linked)");**

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

**printf("%-10s %-10d %-10d ", f[i].fname, f[i].start, f[i].size);**

**for (j = 0; j < f[i].size - 1; j++) {**

**printf("%d --> ", f[i].blocks[j]);**

**}**

**printf("%d\n", f[i].blocks[j]); // Last block without arrow**

**}**

**return 0;**

**}**

**12C).**

**#include <stdio.h>**

**#define MAX\_FILES 20**

**#define MAX\_BLOCKS 20**

**int main() {**

**int n, i, j;**

**int indexBlock[MAX\_FILES], length[MAX\_FILES];**

**int blocks[MAX\_FILES][MAX\_BLOCKS];**

**int fileChoice;**

**// Input: Number of files**

**printf("Enter number of files (max %d): ", MAX\_FILES);**

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

**if (n <= 0 || n > MAX\_FILES) {**

**printf("Invalid number of files.\n");**

**return 1;**

**}**

**// Input for each file**

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

**printf("\nEnter details for File %d:\n", i + 1);**

**printf("Enter index block: ");**

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

**printf("Enter number of blocks used by the file: ");**

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

**if (length[i] <= 0 || length[i] > MAX\_BLOCKS) {**

**printf("Invalid length. Try again.\n");**

**i--;**

**continue;**

**}**

**printf("Enter the %d block numbers:\n", length[i]);**

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

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

**}**

**}**

**// Display all files**

**printf("\n%-10s %-12s %-10s %-20s\n", "File#", "Index Block", "Length", "Blocks");**

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

**printf("%-10d %-12d %-10d ", i + 1, indexBlock[i], length[i]);**

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

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

**}**

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

**}**

**// Query a file**

**printf("\nEnter file number to view its block allocation: ");**

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

**if (fileChoice < 1 || fileChoice > n) {**

**printf("Invalid file number.\n");**

**return 1;**

**}**

**// Display block details for the selected file**

**int idx = fileChoice - 1;**

**printf("\nDetails of File %d:\n", fileChoice);**

**printf("Index Block: %d\n", indexBlock[idx]);**

**printf("Length: %d\n", length[idx]);**

**printf("Blocks: ");**

**for (j = 0; j < length[idx]; j++) {**

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

**}**

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

**return 0;**

**}**

**VIVA:-**

**1. List File access methods.**

There are three ways to access a file into a computer system: Sequential-Access, Direct

Access, Index sequential Method.

· Sequential Access – It is the simplest access method. ...

· Direct Access – Another method is direct access method also known as relative access method.

· Index sequential method

**2. Explain Sequential file allocation method.**

In this allocation strategy, each file occupies a set of contiguously blocks on the disk.

This strategy is best suited. For sequential files, the file allocation table consists of a single

entry for each file. It shows the filenames, starting block of the file and size of the file.

**3. Explain Linked file allocation method.**

Each file is a linked list of disk blocks which need not be contiguous. The disk blocks

Can be scattered anywhere on the disk.

The directory entry contains a pointer to the starting and the ending file block. Each block

contains a pointer to the next block occupied by the file.

**4. Explain Indexed file allocation method.**

Indexed allocation scheme stores all the disk pointers in one of the blocks called as indexed

block. Indexed block doesn't hold the file data, but it holds the pointers to all the disk blocks

allocated to that particular file. Directory entry will only contain the index block address.

**5. What is file system mounting?**

Mounting a file system attaches that file system to a directory (mount point) and makes it

available to the system. The root ( / ) file system is always mounted. Any other file system

can be connected or disconnected from the root ( / ) file system.

**13a).PAGING**

#include <stdio.h>

#define MAX\_FRAMES 20

#define MAX\_PAGES 20

int main() {

int framearr[MAX\_FRAMES];

int prosize, memsize, pagesize;

int pages, frames;

int i, frame\_no, offset;

int logical\_address, physical\_address;

int base = 0; // Assuming base is 0 for simplicity

// Input sizes

printf("Enter the Process size (in bytes): ");

scanf("%d", &prosize);

printf("Enter the Main Memory size (in bytes): ");

scanf("%d", &memsize);

printf("Enter the Page size (in bytes): ");

scanf("%d", &pagesize);

// Calculate number of pages and frames

pages = prosize / pagesize;

frames = memsize / pagesize;

printf("\nThe process is divided into %d pages.", pages);

printf("\nThe main memory is divided into %d frames.\n", frames);

// Initialize frame array

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

framearr[i] = -1;

// Page table creation

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

while (1) {

printf("Enter the frame number to which page %d is allocated: ", i);

scanf("%d", &frame\_no);

if (frame\_no < 0 || frame\_no >= frames) {

printf("\tInvalid frame number. Please enter between 0 and %d.\n", frames - 1);

} else if (framearr[frame\_no] != -1) {

printf("\tFrame %d is already occupied by page %d.\n", frame\_no, framearr[frame\_no]);

} else {

framearr[frame\_no] = i;

break;

}

}

// Translate a logical address to physical address for this page

printf("Enter offset (displacement) within the page: ");

scanf("%d", &offset);

if (offset < 0 || offset >= pagesize) {

printf("Invalid offset. It must be less than page size (%d).\n", pagesize);

i--; // Retry this page

continue;

}

physical\_address = base + (frame\_no \* pagesize) + offset;

printf("Physical Address for page %d, offset %d: %d\n\n", i, offset, physical\_address);

}

// Final page table output

printf("\nPage Table (Page -> Frame):\n");

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

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

if (framearr[i] != -1)

printf("Page %d -> Frame %d\n", framearr[i], i);

}

return 0;

}

Sample Output:-

Enter the Process size (in bytes): 1600

Enter the Main Memory size (in bytes): 3200

Enter the Page size (in bytes): 400

The process is divided into 4 pages.

The main memory is divided into 8 frames.

Enter the frame number for page 0: 3

Enter the frame number for page 1: 5

Enter the frame number for page 2: 1

Enter the frame number for page 3: 6

Enter a logical address (0 - 1599): 1180

Logical Address: 1180

Page Number: 2

Offset: 380

Mapped to Frame: 1

Physical Address: 1780

**13b). SEGMENTATION:**

**#include <stdio.h>**

**#define MAX\_SEGMENTS 10**

**#define MAX\_SEGMENT\_SIZE 20**

**int main() {**

**int base[MAX\_SEGMENTS], limit[MAX\_SEGMENTS];**

**int val[MAX\_SEGMENTS][MAX\_SEGMENT\_SIZE];**

**int size, i, j, logical\_address, segment\_no, physical\_address;**

**// Input segment table**

**printf("Enter the number of segments: ");**

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

**for (i = 0; i < size; i++) {**

**printf("\nEnter information for Segment %d\n", i);**

**printf("Base Address: ");**

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

**printf("Limit: ");**

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

**// Input values stored in each segment**

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

**printf("Enter value at address %d: ", base[i] + j);**

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

**}**

**}**

**// Display segment table**

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

**printf("Segment No.\tBase\tLimit\n");**

**for (i = 0; i < size; i++) {**

**printf("%d\t\t%d\t%d\n", i, base[i], limit[i]);**

**}**

**// Segment access**

**printf("\nEnter Segment Number: ");**

**scanf("%d", &segment\_no);**

**if (segment\_no < 0 || segment\_no >= size) {**

**printf("Invalid Segment Number!\n");**

**return 1;**

**}**

**printf("Enter Logical Address (Offset): ");**

**scanf("%d", &logical\_address);**

**if (logical\_address < 0 || logical\_address >= limit[segment\_no]) {**

**printf("Invalid Logical Address! It exceeds segment limit.\n");**

**return 1;**

**}**

**physical\_address = base[segment\_no] + logical\_address;**

**printf("\nMapped Physical Address = %d", physical\_address);**

**printf("\nValue at that address = %d\n", val[segment\_no][logical\_address]);**

**return 0;**

**}**

**SAMPLE OUTPUT:-**

**Enter the number of segments: 2**

**Enter information for Segment 0**

**Base Address: 100**

**Limit: 3**

**Enter value at address 100: 5**

**Enter value at address 101: 10**

**Enter value at address 102: 15**

**Enter information for Segment 1**

**Base Address: 200**

**Limit: 2**

**Enter value at address 200: 20**

**Enter value at address 201: 25**

**\*\*\*\* SEGMENT TABLE \*\*\*\***

**Segment No. Base Limit**

**0 100 3**

**1 200 2**

**Enter Segment Number: 1**

**Enter Logical Address (Offset): 1**

**Mapped Physical Address = 201**

**Value at that address = 25**

**VIVA**

**1.What is the basic function of paging?**

**P**aging is a memory management scheme that permits the physical-address space of a process to be non-contiguous. It avoids the considerable problem of having to fit varied sized memory chunks onto the backing store.

**2.What is fragmentation?**

Fragmentation is an unwanted problem in the operating system in which the processes are loaded and unloaded from memory, and free memory space is fragmented. Processes can't be assigned to memory blocks due to their small size, and the memory blocks stay unused.

**3.What is thrashing?**

Thrashing is caused by under allocation of the minimum number of pages required by a process, forcing it to continuously page fault.

**4.Differentiate between logical and physical address.**

Logical Address is generated by CPU while a program is running. The logical address is virtual address as it does not exist physically, therefore, it is also known as Virtual Address. Physical Address identifies a physical location of required data in a memory. The user never directly deals with the physical address but can access by its corresponding logical address.

**5.Explain internal fragmentation and external fragmentation.**

**Internal Fragmentation**

When a process is allocated to a memory block, and if the process is smaller than the amount of memory requested, a free space is created in the given memory block. Due to this, the free space of the memory block is unused, which causes internal fragmentation**.**

**External Fragmentation**

External fragmentation happens when a dynamic memory allocation method allocates some memory but leaves a small amount of memory unusable. The quantity of available memory is substantially reduced if there is too much external fragmentation. There is enough memory space to complete a request, but it is not contiguous. It's known as external fragmentation

**14). NAME OF THE EXPERIMENT: Page Replacement Techniques**

**AIM: Write C programs to simulate the following Page Replacement Techniques:**

**a) FIFO**

**b) LRU**

**c)OPTIMAL**

**a) FIFO**

#include <stdio.h>

int main() {

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

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

scanf("%d", &fno);

printf("Enter the number of reference strings: ");

scanf("%d", &n);

printf("Enter the reference string:\n");

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

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

// Initialize frame slots to -1

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

frame[i] = -1;

printf("\nFIFO Page Replacement Algorithm\n");

printf("Reference string:\n");

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

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

printf("\n\n");

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

printf("Reference %d -> ", a[i]);

avail = 0;

for(k = 0; k < fno; k++) {

if(frame[k] == a[i]) {

avail = 1; // Page already in frame

break;

}

}

if(avail == 0) { // Page fault

frame[j] = a[i];

j = (j + 1) % fno;

pagefault++;

for(k = 0; k < fno; k++) {

if(frame[k] != -1)

printf("%2d ", frame[k]);

}

} else {

printf("No page fault");

}

printf("\n");

}

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

return 0;

}

**b) LRU**

#include <stdio.h>

int main() {

int i, j, k, n, fno, a[50], frame[10], lru[10], pagefault = 0, avail, max, pos;

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

scanf("%d", &fno);

printf("Enter the number of reference strings: ");

scanf("%d", &n);

printf("Enter the reference string:\n");

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

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

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

frame[i] = -1;

lru[i] = 0;

}

printf("\nLRU Page Replacement Algorithm\n");

printf("Reference string:\n");

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

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

printf("\n\n");

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

printf("Reference %d -> ", a[i]);

avail = 0;

// Check if the page is already in frame

for(j = 0; j < fno; j++) {

if(frame[j] == a[i]) {

avail = 1;

lru[j] = 0; // Reset LRU for this frame

break;

}

}

if(avail == 0) { // Page fault

pos = -1;

// Find an empty frame

for(j = 0; j < fno; j++) {

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

pos = j;

break;

}

}

// If no empty frame, find LRU

if(pos == -1) {

max = lru[0];

pos = 0;

for(j = 1; j < fno; j++) {

if(lru[j] > max) {

max = lru[j];

pos = j;

}

}

}

frame[pos] = a[i];

lru[pos] = 0;

pagefault++;

}

// Update LRU counters

for(j = 0; j < fno; j++) {

if(frame[j] != -1 && j != pos && frame[j] != a[i]) {

lru[j]++;

}

}

// Display current frame status

for(j = 0; j < fno; j++) {

if(frame[j] != -1)

printf("%2d ", frame[j]);

}

if(avail) printf("No page fault");

printf("\n");

}

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

return 0;

}

**c)OPTIMAL**

#include<stdio.h>

int main()

{

int i,j,l,min,flag1,n,a[50],temp,frame[10],flag,fno,k,avail,pagefault=0,opt[10];

printf("\nEnter the number of Frames : ");

scanf("%d",&fno);

printf("\nEnter number of reference string :");

scanf("%d",&n);

printf("\n Enter the Reference string :\n");

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

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

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

{

frame[i]= -1;

opt[i]=0;

}

printf("\nLFU Page Replacement Algorithm\n\nThe given reference string is:\n\n");

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

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

printf("\n");

j=0;

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

{

flag=0;

flag1=0;

printf("\nReference No %d-> ",a[i]);

avail=0;

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

if(frame[k]==a[i])

{

avail=1;

break;

}

if(avail==0)

{

temp = frame[j];

frame[j]=a[i];

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

{

if(frame[k]==-1)

{

j = k;

flag = 1;

break;

}

}

if(flag==0)

{

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

{

opt[k]=0;

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

{

if(frame[k]==a[l])

{

flag1 = 1;

break;

}

}

if(flag1==1)

opt[k] = l-i;

else

{

opt[k] = -1;

break;

}

}

min = 0;

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

if(opt[k]<opt[min]&&opt[k]!=-1)

min = k;

else if(opt[k]==-1)

{

min = k;

frame[j] = temp;

frame[k] = a[i];

break;

}

j = min;

}

pagefault++;

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

if(frame[k]!=-1)

printf(" %2d",frame[k]);

}

printf("\n");

}

printf("\nPage Fault Is %d",pagefault);

return 0;

}

**VIVA**

**1.Why do we use page replacement algorithms?**

Page replacement algorithms are an important part of virtual memory management and it helps the OS to decide which memory page can be moved out, making space for the currently needed page.However, the ultimate objective of all page replacement algorithms is to reduce the number of page faults.

**2.Which is best page replacement algorithm and why?**

LRU resulted to be the best algorithm for page replacement to implement. LRU maintains a linked list of all pages in the memory, in which, the most recently used page is placed at the front, and the least recently used page is placed at the rear**.**

**3.Explain LRU algorithm**.

LRU stands for Least Recently Used.LRU replaces the line in the cache that has been in the cache the longest with no reference to it. It works on the idea that the more recently used blocks are more likely to be referenced again.

**4. When does a page fault occur?**

Page fault occurs when a requested page is mapped in virtual address space but not present in memory.

**5.What are page replacement algorithms in OS?**

1. First In First Out (FIFO) 2. Optimal Page replacement 3. Least Recently Used