**Experiment number =3**

1. **Write a program to simulate CPU Scheduling Algorithm : FCFS**

#include <iostream>

using namespace std;

// Function to find the waiting time for all processes

void findWaitingTime(int processes[], int n, int bt[], int wt[]) {

// Waiting time for the first process is 0

wt[0] = 0;

// Calculating waiting time for each process

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

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

}

}

// Function to calculate turn around time

void findTurnAroundTime(int processes[], int n, int bt[], int wt[], int tat[]) {

// Calculating turnaround time by adding bt[i] + wt[i]

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

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

}

}

// Function to calculate average time

void findavgTime(int processes[], int n, int bt[]) {

int wt[n], tat[n], total\_wt = 0, total\_tat = 0;

// Function to find waiting time of all processes

findWaitingTime(processes, n, bt, wt);

// Function to find turn around time for all processes

findTurnAroundTime(processes, n, bt, wt, tat);

// Display processes along with all details

cout << "Processes " << "Burst Time "

<< "Waiting Time " << "Turn Around Time\n";

// Calculate total waiting time and total turnaround time

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

total\_wt += wt[i];

total\_tat += tat[i];

cout << " " << processes[i] << "\t\t" << bt[i] << "\t\t"

<< wt[i] << "\t\t " << tat[i] << endl;

}

cout << "Average waiting time = " << (float)total\_wt / n << endl;

cout << "Average turn around time = " << (float)total\_tat / n << endl;

}

// Driver code

int main() {

// Process IDs

int processes[] = {4, 5, 6};

int n = sizeof processes / sizeof processes[0];

// Burst time of all processes

int burst\_time[] = {10, 11, 12};

findavgTime(processes, n, burst\_time);

return 0;

}

1. **Write a program to simulate CPU Scheduling Algorithm : SJF**

#include <iostream>

using namespace std;

int main() {

// Matrix for storing Process Id, Burst Time, Waiting Time, and Turn Around Time.

int A[100][4];

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

float avg\_wt, avg\_tat;

cout << "Enter the number of processes: ";

cin >> n;

// Input Burst Time and assign Process IDs.

cout << "Enter Burst Time:" << endl;

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

cout << "P" << i + 1 << ": ";

cin >> A[i][1]; // Burst Time

A[i][0] = i + 1; // Process ID

}

// Sort processes based on Burst Time (Ascending Order).

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

index = i;

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

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

index = j;

}

}

// Swap Burst Time

temp = A[i][1];

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

A[index][1] = temp;

// Swap Process ID

temp = A[i][0];

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

A[index][0] = temp;

}

// Waiting Time for the first process is 0.

A[0][2] = 0;

// Calculate Waiting Time for each process.

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

A[i][2] = 0;

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

A[i][2] += A[j][1]; // Accumulate previous Burst Times

}

total += A[i][2];

}

avg\_wt = (float)total / n; // Average Waiting Time

total = 0;

// Print header

cout << "P\tBT\tWT\tTAT" << endl;

// Calculate Turn Around Time and print the details.

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

A[i][3] = A[i][1] + A[i][2]; // Turn Around Time = Burst Time + Waiting Time

total += A[i][3];

cout << "P" << A[i][0] << "\t" << A[i][1] << "\t" << A[i][2] << "\t" << A[i][3] << endl;

}

avg\_tat = (float)total / n; // Average Turn Around Time

// Print Average Times

cout << "Average Waiting Time = " << avg\_wt << endl;

cout << "Average Turnaround Time = " << avg\_tat << endl;

return 0;

};

1. **Write a program to simulate CPU Scheduling Algorithm : ROUND Robin**

#include <iostream>

using namespace std;

// Function to find the waiting time for all processes

void findWaitingTime(int processes[], int n, int bt[], int wt[], int quantum) {

int rem\_bt[n];

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

rem\_bt[i] = bt[i]; // Copy burst times into remaining burst times

int t = 0; // Current time

// Keep traversing processes in round-robin manner until all are done

while (true) {

bool done = true;

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

// If burst time of a process is greater than 0, process further

if (rem\_bt[i] > 0) {

done = false; // There is a pending process

if (rem\_bt[i] > quantum) {

// Process for the quantum time

t += quantum;

rem\_bt[i] -= quantum;

} else {

// Process completes within the remaining burst time

t += rem\_bt[i];

wt[i] = t - bt[i]; // Calculate waiting time

rem\_bt[i] = 0; // Process is done

}

}

}

// If all processes are done, exit the loop

if (done == true)

break;

}

}

// Function to calculate turn around time

void findTurnAroundTime(int processes[], int n, int bt[], int wt[], int tat[]) {

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

tat[i] = bt[i] + wt[i]; // Turnaround time = Burst time + Waiting time

}

// Function to calculate average time

void findavgTime(int processes[], int n, int bt[], int quantum) {

int wt[n], tat[n], total\_wt = 0, total\_tat = 0;

// Find waiting time for all processes

findWaitingTime(processes, n, bt, wt, quantum);

// Find turn around time for all processes

findTurnAroundTime(processes, n, bt, wt, tat);

// Display processes along with all details

cout << "PN\tBurst Time\tWaiting Time\tTurn Around Time\n";

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

total\_wt += wt[i];

total\_tat += tat[i];

cout << " " << processes[i] << "\t\t" << bt[i] << "\t\t"

<< wt[i] << "\t\t " << tat[i] << endl;

}

// Calculate and display average waiting and turnaround times

cout << "Average waiting time = " << (float)total\_wt / n << endl;

cout << "Average turn around time = " << (float)total\_tat / n << endl;

}

// Driver code

int main() {

// Process IDs

int processes[] = {1, 2, 3};

int n = sizeof processes / sizeof processes[0];

// Burst time of all processes

int burst\_time[] = {10, 5, 8};

// Time quantum

int quantum = 2;

// Calculate average time

findavgTime(processes, n, burst\_time, quantum);

return 0;

};

1. **Write a program to simulate CPU Scheduling Algorithm : Priority (c language)**

#include <stdio.h>

// Function to swap two integers

void swap(int \*a, int \*b) {

int temp = \*a;

\*a = \*b;

\*b = temp;

}

int main() {

int n;

// Input: Number of processes

printf("Enter Number of Processes: ");

scanf("%d", &n);

int burst[n], priority[n], index[n];

// Input: Burst time and priority for each process

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

printf("Enter Burst Time and Priority Value for Process %d: ", i + 1);

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

index[i] = i + 1; // Store process IDs

}

// Sorting processes based on priority (higher priority first)

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

int max\_priority = priority[i], max\_index = i;

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

if (priority[j] > max\_priority) {

max\_priority = priority[j];

max\_index = j;

}

}

// Swap priority, burst time, and process ID

swap(&priority[i], &priority[max\_index]);

swap(&burst[i], &burst[max\_index]);

swap(&index[i], &index[max\_index]);

}

// Output: Order of process execution

int t = 0;

printf("\nOrder of Process Execution:\n");

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

printf("P%d is executed from %d to %d\n", index[i], t, t + burst[i]);

t += burst[i];

}

// Output: Process details and waiting time

printf("\nProcess ID\tBurst Time\tWait Time\n");

int wait\_time = 0, total\_wait\_time = 0;

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

printf("P%d\t\t%d\t\t%d\n", index[i], burst[i], wait\_time);

total\_wait\_time += wait\_time;

wait\_time += burst[i];

}

// Average waiting time

float avg\_wait\_time = (float)total\_wait\_time / n;

printf("Average Waiting Time = %.2f\n", avg\_wait\_time);

// Average turnaround time

int total\_turnaround\_time = 0;

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

total\_turnaround\_time += burst[i];

}

float avg\_turnaround\_time = (float)total\_turnaround\_time / n;

printf("Average Turnaround Time = %.2f\n", avg\_turnaround\_time);

return 0;

};

**Experiment number =4**

1. **Write a program to simulate Memory placement strategies strategies : best fit**

#include <iostream>

using namespace std;

// Method 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

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

allocation[i] = -1;

// Pick each process and find suitable blocks according to its size and 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 || blockSize[bestIdx] > blockSize[j])

bestIdx = j;

}

}

// If we could find a block for the current process

if (bestIdx != -1) {

// Allocate block `bestIdx` to process `i`

allocation[i] = bestIdx;

// Reduce available memory in this block

blockSize[bestIdx] -= processSize[i];

}

}

// Display the allocation details

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; // Adding 1 for 1-based indexing

else

cout << "Not Allocated";

cout << endl;

}

}

// Driver Method

int main() {

int blockSize[] = {1000, 2000, 3000, 4000, 5000};

int processSize[] = {1014, 4212, 1410, 2501};

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

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

// Call the Best-Fit function

bestFit(blockSize, m, processSize, n);

return 0;

}

1. **Write a program to simulate Memory placement strategies strategies : first fit**

#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 and assign to it

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

// Traverse through all the blocks

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

// If block is large enough, allocate it

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

allocation[i] = j; // Allocate block j to process i

// Reduce available memory in this block

blockSize[j] -= processSize[i];

break;

}

}

}

// Display the allocation details

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; // Add 1 for 1-based index

else

cout << "Not Allocated";

cout << endl;

}

}

// Driver Code

int main() {

// Define the sizes of memory blocks

int blockSize[] = {100, 200, 300, 400, 500};

// Define the sizes of processes to be allocated memory

int processSize[] = {212, 417, 542, 304, 145};

// Number of memory blocks and processes

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

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

// Call the First-Fit function

firstFit(blockSize, m, processSize, n);

return 0;

}

1. **Write a program to simulate Memory placement strategies strategies : Next fit**

#include <bits/stdc++.h>

using namespace std;

// Function to allocate memory to blocks as per Next fit algorithm

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

// Stores block id of the block allocated to a process

int allocation[n];

int j = 0, t = m - 1;

// Initially no block is assigned to any process

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

// Traverse through each process and find suitable blocks for it

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

// Start from the current position (j) and look for a block

bool allocated = false;

while (j < m) {

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

// Allocate block j to process i

allocation[i] = j;

// Reduce the available memory in this block

blockSize[j] -= processSize[i];

allocated = true;

break;

}

// Move to the next block in a circular manner

j = (j + 1) % m;

}

// If no suitable block found after traversing all blocks, stop

if (!allocated) {

break;

}

}

// Display process allocation details

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; // Output block number (1-based index)

else

cout << "Not Allocated"; // If no block is allocated

cout << endl;

}

}

// Driver code

int main() {

// Define block sizes

int blockSize[] = {10, 20, 30, 40, 50};

// Define process sizes

int processSize[] = {2, 11, 22, 34, 14};

// Number of blocks and processes

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

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

// Call the Next-Fit function

NextFit(blockSize, m, processSize, n);

return 0;

}

1. **Write a program to simulate Memory placement strategies strategies : Worst fit**

#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));

// Traverse through each process and find suitable blocks for it

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

// Find the block with the largest available space

int wstIdx = -1;

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

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

// If no block has been found yet or the current block has a larger space

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

wstIdx = j;

}

}

}

// If a suitable block was found

if (wstIdx != -1) {

// Allocate block to process

allocation[i] = wstIdx;

// Reduce available memory in the block

blockSize[wstIdx] -= processSize[i];

}

}

// Display process allocation details

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; // Output block number (1-based index)

else

cout << "Not Allocated"; // If no block is allocated

cout << endl;

}

}

// Driver code

int main() {

// Define block sizes

int blockSize[] = {700, 900, 500, 600, 400};

// Define process sizes

int processSize[] = {412, 510, 512, 626};

// Number of blocks and processes

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

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

// Call the Worst-Fit function

worstFit(blockSize, m, processSize, n);

return 0;

};