**FCFS**

#include <stdio.h>

int main() {

int pid[15], bt[15], n, wt[15];

float twt = 0.0, tat = 0.0;

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

scanf("%d", &n);

printf("Enter process ID of all the processes: ");

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

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

}

printf("Enter burst time of all the processes: ");

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

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

}

wt[0] = 0; // Waiting time for the first process is always 0

// Calculate waiting time for each process

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

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

}

printf("Process ID Burst Time Waiting Time Turnaround Time\n");

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

int tat\_i = bt[i] + wt[i]; // Turnaround time for process i

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

twt += wt[i]; // Total waiting time

tat += tat\_i; // Total turnaround time

}

printf("Avg. waiting time = %.2f\n", twt / n);

printf("Avg. turnaround time = %.2f\n", tat / n);

return 0;

}

**ROUND ROBIN**

#include <stdio.h>

int main()

{

int n, tq, i, time = 0, remain;

int pid[15], bt[15], rt[15]; // pid: process IDs, bt: burst times, rt: remaining times

float awt = 0.0, att = 0.0; // Average waiting time and turnaround time

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

scanf("%d", &n);

remain = n; // Remaining processes to execute

printf("Enter process ID of all processes: ");

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

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

}

printf("Enter burst time of all processes: ");

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

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

rt[i] = bt[i]; // Initialize remaining times

}

printf("Enter the time quantum: ");

scanf("%d", &tq);

printf("Process ID Burst Time Waiting Time TurnAround Time\n");

int wt[15] = {0}; // Initialize waiting times

while (remain > 0) {

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

if (rt[i] > 0) {

if (rt[i] > tq) {

time += tq;

rt[i] -= tq;

} else {

time += rt[i];

wt[i] = time - bt[i]; // Waiting time = Total time - Burst time

rt[i] = 0; // Process is finished

remain--;

}

}

}

}

// Calculate turnaround times and display results

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

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

awt += wt[i]; // Accumulate total waiting time

att += tat; // Accumulate total turnaround time

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

}

// Calculate averages

awt /= n;

att /= n;

printf("Avg. waiting time = %f\n", awt);

printf("Avg. turnaround time = %f\n", att);

return 0;

}

**SJF (non preemptive)**

#include <stdio.h>

int main() {

int n;

// Taking input for number of processes

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

scanf("%d", &n);

int pid[n], bt[n], at[n], wt[n], tat[n], completed[n];

int total\_wt = 0, total\_tat = 0, current\_time = 0;

// Accept Process IDs using a for loop

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

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

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

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

}

// Accept Burst Times using a separate for loop

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

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

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

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

}

// Accept Arrival Times using a separate for loop

printf("Enter Arrival Times: \n");

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

printf("Enter Arrival Time for Process %d: ", i + 1);

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

completed[i] = 0; // Initially, no process is completed

}

// Calculate waiting time and turnaround time for each process

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

int idx = -1;

int min\_bt = 9999;

// Find the process with the smallest burst time that has arrived and not yet completed

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

if (!completed[i] && at[i] <= current\_time && bt[i] < min\_bt) {

min\_bt = bt[i];

idx = i;

}

}

// If no process can be executed (i.e., all are waiting), move the current time forward

if (idx == -1) {

current\_time++;

count--;

continue;

}

// Calculate the waiting time and turnaround time for the selected process

wt[idx] = current\_time - at[idx];

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

// Update total waiting time and total turnaround time

total\_wt += wt[idx];

total\_tat += tat[idx];

// Update the current time

current\_time += bt[idx];

// Mark the process as completed

completed[idx] = 1;

}

// Output the results

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

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

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

}

// Calculating and printing average waiting time and average turnaround time

printf("\nAverage Waiting Time: %.2f\n", (float)total\_wt / n);

printf("Average Turnaround Time: %.2f\n", (float)total\_tat / n);

return 0;

}

**FIFO**

#include <stdio.h>

int main() {

// Hardcoded reference string and number of frames

int pages[] = {7, 0, 1, 2, 0, 3, 0, 4};

int n = 8; // Number of pages

int frames = 3; // Number of memory frames

int memory[3] = {-1, -1, -1}; // Initialize memory to -1 (empty)

int front = 0; // Points to the oldest page

int pageFaults = 0;

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

int found = 0;

// Check if the page is already in memory

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

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

found = 1; // Page hit

break;

}

}

// If page is not in memory, replace the oldest page

if (!found) {

memory[front] = pages[i];

front = (front + 1) % frames; // Move to the next oldest frame

pageFaults++;

}

}

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

return 0;

}

**OPTIMAL PAGE REPLACEMENT**

#include <stdio.h>

int main() {

// Hardcoded reference string and number of frames

int pages[] = {7, 0, 1, 2, 0, 3, 0, 4};

int n = 8; // Number of pages

int frames = 3; // Number of memory frames

int memory[3] = {-1, -1, -1}; // Initialize memory to -1 (empty)

int pageFaults = 0;

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

int found = 0;

// Check if the page is already in memory

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

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

found = 1; // Page hit

break;

}

}

// If page is not in memory, replace a page

if (!found) {

int replaceIndex = -1, farthest = -1;

// If an empty frame is available, use it

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

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

replaceIndex = j;

break;

}

}

// If no empty frame, find the page to replace

if (replaceIndex == -1) {

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

int k;

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

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

break;

}

}

if (k > farthest) {

farthest = k;

replaceIndex = j;

}

}

}

// Replace the page in memory

memory[replaceIndex] = pages[i];

pageFaults++;

}

}

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

return 0;

}

**LEAST RECENTLY USED (LRU)**

#include <stdio.h>

int main() {

// Hardcoded reference string and number of frames

int p[] = {7, 0, 1, 2, 0, 3, 0, 4}; // Page references

int n = 8; // Number of pages

int f = 3; // Number of frames

int mem[3] = {-1, -1, -1}; // Memory frames initialized to -1 (empty)

int last[3] = {0, 0, 0}; // Last used times for frames

int faults = 0; // Page fault counter

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

int hit = 0;

// Check if the page is already in memory

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

if (mem[j] == p[i]) {

hit = 1; // Page hit

last[j] = i; // Update last used time

break;

}

}

// If page is not in memory, replace the least recently used page

if (!hit) {

int lru = 0;

// Find the least recently used frame

for (int j = 1; j < f; j++) {

if (last[j] < last[lru]) {

lru = j;

}

}

// Replace the page in the LRU frame

mem[lru] = p[i];

last[lru] = i; // Update last used time

faults++;

}

}

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

return 0;

}

**NEED MATRIX**

#include <stdio.h>

#define P 5 // Processes

#define R 3 // Resources

int main() {

// Example: Maximum matrix (Max demand for each process)

int max[P][R] = {

{7, 5, 3},

{3, 2, 2},

{9, 0, 2},

{2, 2, 2},

{4, 3, 3}

};

// Example: Allocated matrix (Allocated resources to each process)

int alloc[P][R] = {

{0, 1, 0},

{2, 1, 1},

{3, 2, 2},

{2, 1, 1},

{0, 0, 2}

};

// Need matrix to store the result

int need[P][R];

// Calculate Need matrix and print it

printf("Need Matrix:\n");

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

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

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

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

}

printf("\n");

}

return 0;

}

**SAFE SEQUENCE**

#include <stdio.h>

#include <stdbool.h>

#define P 5 // Processes

#define R 3 // Resources

int main() {

// Maximum resources needed by each process

int max[P][R] = {

{7, 5, 3},

{3, 2, 2},

{9, 0, 2},

{2, 2, 2},

{4, 3, 3}

};

// Resources allocated to each process

int alloc[P][R] = {

{0, 1, 0},

{2, 1, 1},

{3, 2, 2},

{2, 1, 1},

{0, 0, 2}

};

// Available resources

int avail[R] = {3, 3, 2};

// Need matrix: Max - Allocated

int need[P][R];

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

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

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

// Work vector and Finish array

int work[R];

bool finish[P] = {0}; // All processes initially unfinished

for (int i = 0; i < R; i++) work[i] = avail[i];

// Safe sequence array

int safeSeq[P], idx = 0;

// Banker's algorithm to find safe sequence

for (int count = 0; count < P; count++) {

bool found = false;

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

if (!finish[i]) {

bool canFinish = true;

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

if (need[i][j] > work[j]) {

canFinish = false;

break;

}

if (canFinish) {

// Update work and mark process as finished

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

work[j] += alloc[i][j];

finish[i] = true;

safeSeq[idx++] = i;

found = true;

break;

}

}

}

if (!found) { printf("System is in an unsafe state.\n"); return 0; }

}

// If all processes finished, print the safe sequence

printf("System is in a safe state.\nSafe sequence: ");

for (int i = 0; i < P; i++) printf("P%d ", safeSeq[i]);

printf("\n");

return 0;

}

/\*

Explanation:

Input Matrices:

max[P][R]: Maximum resources needed by each process.

alloc[P][R]: Resources allocated to each process.

avail[R]: Available resources in the system.

Need Matrix Calculation:

need[i][j] = max[i][j] - alloc[i][j] computes the remaining resources each process needs.

Work Vector:

work[]: Holds the available resources, initialized with the system's available resources (avail[]).

Safe Sequence Calculation:

Finish Array: Tracks which processes have been completed.

In each iteration, we check if a process can finish (i.e., its Need is less than or equal to the Work vector). If so, we simulate the process finishing and update the Work vector (by adding the allocated resources of that process back to the available pool).

The process is marked as finished and added to the safeSeq[] array.

Termination:

If no process can finish in any iteration, the system is in an unsafe state.

If all processes can finish, the system is in a safe state, and the safe sequence is printed \*/

/\*For the input:

Maximum Matrix

0 1 0

2 1 1

3 2 2

2 1 1

0 0 2

Allocated Matrix:

0 1 0

2 1 1

3 2 2

2 1 1

Available Resources:

3 3 2

System is in a safe state.

Safe sequence: P1 P3 P4 P0 P2\*/

**FIRST FIT**

#include <stdio.h>

#define MAX\_BLOCKS 10

int main() {

int block\_sizes[MAX\_BLOCKS], process\_allocations[MAX\_BLOCKS];

int num\_blocks, num\_processes, process\_sizes[MAX\_BLOCKS];

// Input memory block sizes

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

scanf("%d", &num\_blocks);

printf("Enter the sizes of the memory blocks:\n");

for (int i = 0; i < num\_blocks; i++) {

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

scanf("%d", &block\_sizes[i]);

process\_allocations[i] = -1; // Initially, no process is allocated

}

// Input process sizes

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

scanf("%d", &num\_processes);

printf("Enter the sizes of the processes:\n");

for (int i = 0; i < num\_processes; i++) {

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

scanf("%d", &process\_sizes[i]);

}

// First-fit memory allocation

for (int i = 0; i < num\_processes; i++) {

for (int j = 0; j < num\_blocks; j++) {

if (block\_sizes[j] >= process\_sizes[i]) {

process\_allocations[j] = i; // Allocate process i to block j

block\_sizes[j] -= process\_sizes[i]; // Reduce block size

break;

}

}

}

// Output allocation results

printf("\nMemory Allocation:\n");

for (int i = 0; i < num\_blocks; i++) {

if (process\_allocations[i] != -1)

printf("Block %d -> Process %d\n", i + 1, process\_allocations[i] + 1);

else

printf("Block %d -> Free\n", i + 1);

}

return 0;

}