**Write a C program to simulate the following CPU scheduling algorithm to find turnaround time and waiting time. (Any one)**

**a) FCFS**

**b) SJF**

**c) Priority**

**d) Round Robin (Experiment with different quantum sizes for RR algorithm)**

**a) FCFS**

#include<stdio.h>

int n, i, j;

int Burst\_time[20], Arrival\_time[20], Waiting\_time[20], Turn\_around\_time[20], Completion\_time[20], process[20];

float avg\_Turn\_around\_time = 0, avg\_Waiting\_time = 0;

void sortByArrival() {

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

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

if (Arrival\_time[i] > Arrival\_time[j]) {

// Swap Arrival Time

int temp = Arrival\_time[i];

Arrival\_time[i] = Arrival\_time[j];

Arrival\_time[j] = temp;

// Swap Burst Time

temp = Burst\_time[i];

Burst\_time[i] = Burst\_time[j];

Burst\_time[j] = temp;

// Swap Process ID

temp = process[i];

process[i] = process[j];

process[j] = temp;

}

}

}

}

int FCFS() {

sortByArrival(); // Sort based on arrival time

Completion\_time[0] = Arrival\_time[0] + Burst\_time[0];

Turn\_around\_time[0] = Completion\_time[0] - Arrival\_time[0];

Waiting\_time[0] = Turn\_around\_time[0] - Burst\_time[0];

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

if (Arrival\_time[i] > Completion\_time[i - 1])

Completion\_time[i] = Arrival\_time[i] + Burst\_time[i]; // CPU idle

else

Completion\_time[i] = Completion\_time[i - 1] + Burst\_time[i];

Turn\_around\_time[i] = Completion\_time[i] - Arrival\_time[i];

Waiting\_time[i] = Turn\_around\_time[i] - Burst\_time[i];

}

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

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

avg\_Waiting\_time += Waiting\_time[i];

avg\_Turn\_around\_time += Turn\_around\_time[i];

printf("\nP[%d]\t%d\t\t%d\t\t%d\t\t%d", process[i], Arrival\_time[i], Burst\_time[i], Waiting\_time[i], Turn\_around\_time[i]);

}

avg\_Waiting\_time /= n;

avg\_Turn\_around\_time /= n;

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

printf("\nAverage Turnaround Time: %.2f\n", avg\_Turn\_around\_time);

return 0;

}

int main() {

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

scanf("%d", &n);

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

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

scanf("%d", &Arrival\_time[i]);

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

scanf("%d", &Burst\_time[i]);

process[i] = i + 1;

}

FCFS();

return 0;

}

**b) SJF**

#include<stdio.h>

int n, i, j, pos, temp, Burst\_time[20], Arrival\_time[20], Waiting\_time[20], Turn\_around\_time[20], Completion\_time[20], process[20], total = 0;

float avg\_Turn\_around\_time = 0, avg\_Waiting\_time = 0;

void sortByArrivalAndBurst() {

// Sorting processes by Arrival Time first, and then by Burst Time

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

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

if (Arrival\_time[i] > Arrival\_time[j] || (Arrival\_time[i] == Arrival\_time[j] && Burst\_time[i] > Burst\_time[j])) {

// Swap Arrival Time

int temp\_arrival = Arrival\_time[i];

Arrival\_time[i] = Arrival\_time[j];

Arrival\_time[j] = temp\_arrival;

// Swap Burst Time

int temp\_burst = Burst\_time[i];

Burst\_time[i] = Burst\_time[j];

Burst\_time[j] = temp\_burst;

// Swap Process ID

int temp\_process = process[i];

process[i] = process[j];

process[j] = temp\_process;

}

}

}

}

int SJF() {

sortByArrivalAndBurst(); // Sort based on Arrival Time and then Burst Time

Completion\_time[0] = Arrival\_time[0] + Burst\_time[0]; // Completion time for the first process

Turn\_around\_time[0] = Completion\_time[0] - Arrival\_time[0]; // Turnaround Time = Completion Time - Arrival Time

Waiting\_time[0] = Turn\_around\_time[0] - Burst\_time[0]; // Waiting Time = Turnaround Time - Burst Time

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

// If the arrival time of the current process is greater than or equal to the completion time of the previous process

if (Arrival\_time[i] >= Completion\_time[i - 1]) {

Completion\_time[i] = Arrival\_time[i] + Burst\_time[i]; // CPU idle, set completion time

} else {

Completion\_time[i] = Completion\_time[i - 1] + Burst\_time[i]; // Add burst time to previous completion time

}

Turn\_around\_time[i] = Completion\_time[i] - Arrival\_time[i]; // Turnaround Time = Completion Time - Arrival Time

Waiting\_time[i] = Turn\_around\_time[i] - Burst\_time[i]; // Waiting Time = Turnaround Time - Burst Time

}

avg\_Waiting\_time = 0;

total = 0;

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

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

total += Turn\_around\_time[i];

avg\_Waiting\_time += Waiting\_time[i];

printf("\nP[%d]\t%d\t\t%d\t\t%d\t\t%d", process[i], Arrival\_time[i], Burst\_time[i], Waiting\_time[i], Turn\_around\_time[i]);

}

avg\_Waiting\_time /= n;

avg\_Turn\_around\_time = (float)total / n;

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

printf("\nAverage Turnaround Time=%.2f\n", avg\_Turn\_around\_time);

return 0;

}

int main() {

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

scanf("%d", &n);

// Input burst time and arrival time for each process

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

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

scanf("%d", &Arrival\_time[i]);

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

scanf("%d", &Burst\_time[i]);

process[i] = i + 1;

}

// Call the SJF function

SJF();

return 0;

}

**d****) Round Robin (Experiment with different quantum sizes for RR algorithm)**

#include <stdio.h>

struct Process {

int id, at, bt, rt, wt, tat, completed;

};

int main() {

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

struct Process p[10];

float awt = 0, atat = 0;

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

scanf("%d", &n);

remain = n;

// Input Arrival and Burst Times

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

p[i].id = i + 1;

printf("Enter Arrival Time and Burst Time for Process %d: ", p[i].id);

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

p[i].rt = p[i].bt; // remaining time = burst time initially

p[i].completed = 0;

}

printf("Enter Time Quantum: ");

scanf("%d", &tq);

int done = 0;

while (remain > 0) {

done = 1;

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

if (p[i].rt > 0 && p[i].at <= time) {

done = 0;

if (p[i].rt > tq) {

time += tq;

p[i].rt -= tq;

} else {

time += p[i].rt;

p[i].wt = time - p[i].bt - p[i].at;

p[i].tat = time - p[i].at;

p[i].rt = 0;

p[i].completed = 1;

remain--;

}

}

}

if (done) time++; // CPU idle, no process has arrived yet

}

// Output

printf("\nPROCESS\tAT\tBT\tWT\tTAT\n");

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

awt += p[i].wt;

atat += p[i].tat;

printf("P%d\t%d\t%d\t%d\t%d\n", p[i].id, p[i].at, p[i].bt, p[i].wt, p[i].tat);

}

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

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

return 0;

}

**c) Priority**

#include <stdio.h>

struct Process {

int id;

int burst\_time;

int arrival\_time;

int priority;

int completion\_time;

int turnaround\_time;

int waiting\_time;

int is\_completed;

};

int main() {

int n, time = 0, completed = 0;

float total\_TAT = 0, total\_WT = 0;

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

scanf("%d", &n);

struct Process p[n];

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

p[i].id = i + 1;

printf("Enter Arrival Time, Burst Time & Priority for Process %d: ", p[i].id);

scanf("%d %d %d", &p[i].arrival\_time, &p[i].burst\_time, &p[i].priority);

p[i].is\_completed = 0;

}

while (completed < n) {

int idx = -1, highest\_priority = 9999;

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

if (p[i].arrival\_time <= time && !p[i].is\_completed) {

if (p[i].priority < highest\_priority) {

highest\_priority = p[i].priority;

idx = i;

} else if (p[i].priority == highest\_priority) {

if (p[i].arrival\_time < p[idx].arrival\_time) {

idx = i;

}

}

}

}

if (idx != -1) {

time += p[idx].burst\_time;

p[idx].completion\_time = time;

p[idx].turnaround\_time = p[idx].completion\_time - p[idx].arrival\_time;

p[idx].waiting\_time = p[idx].turnaround\_time - p[idx].burst\_time;

total\_TAT += p[idx].turnaround\_time;

total\_WT += p[idx].waiting\_time;

p[idx].is\_completed = 1;

completed++;

} else {

time++;

}

}

printf("\nProcess\tAT\tBT\tPriority\tCT\tTAT\tWT\n");

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

printf("P%d\t%d\t%d\t%d\t\t%d\t%d\t%d\n", p[i].id, p[i].arrival\_time, p[i].burst\_time,

p[i].priority, p[i].completion\_time, p[i].turnaround\_time, p[i].waiting\_time);

}

printf("\nAverage Turnaround Time = %.2f", total\_TAT / n);

printf("\nAverage Waiting Time = %.2f\n", total\_WT / n);

return 0;

}

**Write a C program to simulate Real-Time CPU Scheduling algorithms:**

**5.            Rate- Monotonic**

**6.            Earliest-deadline First**

**5.            Rate- Monotonic**

#include <stdio.h>

#include <math.h>

#include <stdbool.h>

#define MAX\_PROCESS 10

int num\_of\_process;

int execution\_time[MAX\_PROCESS], period[MAX\_PROCESS], remain\_time[MAX\_PROCESS];

void get\_process\_info() {

printf("Enter total number of processes (maximum %d): ", MAX\_PROCESS);

scanf("%d", &num\_of\_process);

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

printf("\nProcess %d:\n", i + 1);

printf("==> Execution time: ");

scanf("%d", &execution\_time[i]);

printf("==> Period: ");

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

remain\_time[i] = execution\_time[i];

}

}

int max(int a, int b, int c) {

return (a > b ? (a > c ? a : c) : (b > c ? b : c));

}

int get\_observation\_time() {

return max(period[0], period[1], period[2]);

}

void print\_schedule(int process\_list[], int cycles) {

printf("\nScheduling:\n\n");

printf("Time: ");

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

printf("| %02d ", i);

}

printf("|\n");

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

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

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

if (process\_list[j] == i + 1)

printf("|####");

else

printf("| ");

}

printf("|\n");

}

}

void rate\_monotonic(int time) {

int process\_list[100] = {0};

float utilization = 0;

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

utilization += (1.0 \* execution\_time[i]) / period[i];

}

int n = num\_of\_process;

float m = n \* (pow(2, 1.0 / n) - 1);

if (utilization > m) {

printf("\nThe given processes are not schedulable under Rate Monotonic.\n");

return;

}

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

int min = 1000, next\_process = -1;

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

if (remain\_time[j] > 0 && period[j] < min) {

min = period[j];

next\_process = j;

}

}

if (next\_process != -1) {

process\_list[i] = next\_process + 1;

remain\_time[next\_process]--;

if ((i + 1) % period[next\_process] == 0) {

remain\_time[next\_process] = execution\_time[next\_process];

}

}

}

print\_schedule(process\_list, time);

}

int main() {

get\_process\_info();

int observation\_time = get\_observation\_time();

rate\_monotonic(observation\_time);

return 0;

}

**6.            Earliest-deadline First**

#include <stdio.h>

#include <stdbool.h>

#include <stdlib.h>

#define MAX\_PROCESS 10

int num\_of\_process;

int execution\_time[MAX\_PROCESS],deadline[MAX\_PROCESS], remain\_time[MAX\_PROCESS], remain\_deadline[MAX\_PROCESS];

void get\_process\_info\_EDF() {

printf("Enter total number of processes (maximum %d): ", MAX\_PROCESS);

scanf("%d", &num\_of\_process);

if (num\_of\_process < 1) {

printf("Invalid number of processes!\n");

exit(0);

}

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

printf("\nProcess %d:\n", i + 1);

printf("==> Execution time: ");

scanf("%d", &execution\_time[i]);

printf("==> Deadline: ");

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

remain\_time[i] = execution\_time[i];

remain\_deadline[i] = deadline[i];

}

}

int max\_EDF(int a, int b, int c) {

return (a > b ? (a > c ? a : c) : (b > c ? b : c));

}

int get\_observation\_time\_EDF() {

return max\_EDF(deadline[0], deadline[1], deadline[2]);

}

void print\_schedule\_EDF(int process\_list[], int cycles) {

printf("\nScheduling:\n\n");

printf("Time: ");

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

printf("| %02d ", i);

}

printf("|\n");

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

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

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

if (process\_list[j] == i + 1)

printf("|####");

else

printf("| ");

}

printf("|\n");

}

}

void earliest\_deadline\_first(int time) {

int process\_list[100] = {0};

int current\_process = -1, min\_deadline;

for (int t = 0; t < time; t++) {

min\_deadline = 1000;

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

if (remain\_time[i] > 0 && deadline[i] < min\_deadline) {

min\_deadline = deadline[i];

current\_process = i;

}

}

if (current\_process != -1) {

process\_list[t] = current\_process + 1;

remain\_time[current\_process]--;

if (remain\_time[current\_process] == 0) {

deadline[current\_process] += remain\_deadline[current\_process];

remain\_time[current\_process] = execution\_time[current\_process];

}

}

}

print\_schedule\_EDF(process\_list, time);

}

int main() {

get\_process\_info\_EDF();

int observation\_time = get\_observation\_time\_EDF();

earliest\_deadline\_first(observation\_time);

return 0;

}

**Write a C program to simulate producer-consumer problem using semaphores.**

#include <stdio.h>

#include <stdlib.h>

// Semaphore variables

int mutex = 1; // Mutex for mutual exclusion

int full = 0; // Count of full slots

int empty = 3; // Count of empty slots

int x = 0; // Item count

// Function prototypes

void producer();

void consumer();

int wait(int s);

int signal(int s);

int main() {

int n;

printf("\n1. Producer\n2. Consumer\n3. Exit");

while (1) {

printf("\nEnter your choice: ");

scanf("%d", &n);

switch (n) {

case 1:

if ((mutex == 1) && (empty != 0))

producer();

else

printf("Buffer is full!!\n");

break;

case 2:

if ((mutex == 1) && (full != 0))

consumer();

else

printf("Buffer is empty!!\n");

break;

case 3:

exit(0);

default:

printf("Invalid choice! Please choose 1, 2, or 3.\n");

}

}

return 0;

}

// Decrease semaphore value (Wait/Acquire)

int wait(int s) {

return (--s);

}

// Increase semaphore value (Signal/Release)

int signal(int s) {

return (++s);

}

// Producer function

void producer() {

mutex = wait(mutex); // Lock the critical section

full = signal(full); // Increase the number of full slots

empty = wait(empty); // Decrease the number of empty slots

x++; // Produce an item

printf("\nProducer produces the item %d\n", x);

mutex = signal(mutex); // Release the critical section

}

// Consumer function

void consumer() {

mutex = wait(mutex); // Lock the critical section

full = wait(full); // Decrease the number of full slots

empty = signal(empty); // Increase the number of empty slots

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

x--; // Consume the item

mutex = signal(mutex); // Release the critical section

}

**Write a C program to simulate the concept of Dining-Philosophers problem.**

#include <stdio.h>

#include <pthread.h>

#include <semaphore.h>

#include <unistd.h> // For sleep()

// Number of philosophers

#define N 5

// States of the philosopher

#define THINKING 2

#define HUNGRY 1

#define EATING 0

// Get the left and right philosopher indices

#define LEFT (i + 4) % N

#define RIGHT (i + 1) % N

// Global variables

int state[N]; // Array to track the state of each philosopher

int phil[N] = {0, 1, 2, 3, 4}; // Philosopher IDs

// Semaphores

sem\_t mutex; // Semaphore to control access to critical section

sem\_t S[N]; // Semaphore for each philosopher

// Function to test if a philosopher can start eating

void test(int i) {

if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {

// Philosopher `i` can eat if neighbors are not eating

state[i] = EATING;

sleep(2);

printf("Philosopher %d takes forks %d and %d\n", i + 1, LEFT + 1, i + 1);

printf("Philosopher %d is Eating\n", i + 1);

// Signal to philosopher `i` that they can start eating

sem\_post(&S[i]);

}

}

// Function for philosopher to take forks

void take\_fork(int i) {

sem\_wait(&mutex); // Enter critical section

state[i] = HUNGRY; // Philosopher is hungry

printf("Philosopher %d is Hungry\n", i + 1);

test(i); // Try to pick up forks

sem\_post(&mutex); // Leave critical section

sem\_wait(&S[i]); // Wait until forks are available

sleep(1);

}

// Function for philosopher to put forks back

void put\_fork(int i) {

sem\_wait(&mutex); // Enter critical section

state[i] = THINKING; // Philosopher starts thinking

printf("Philosopher %d putting fork %d and %d down\n", i + 1, LEFT + 1, i + 1);

printf("Philosopher %d is thinking\n", i + 1);

// Test neighbors if they can start eating

test(LEFT);

test(RIGHT);

sem\_post(&mutex); // Leave critical section

}

// Philosopher thread function

void\* philosopher(void\* num) {

while (1) {

int\* i = num;

sleep(1); // Think for a while

take\_fork(\*i); // Try to pick up forks

sleep(0); // Eat

put\_fork(\*i); // Put down forks

}

}

int main() {

int i;

pthread\_t thread\_id[N]; // Thread IDs for philosophers

// Initialize the semaphores

sem\_init(&mutex, 0, 1); // Initialize mutex to 1

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

sem\_init(&S[i], 0, 0); // Initialize each philosopher's semaphore to 0

}

// Create philosopher threads

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

pthread\_create(&thread\_id[i], NULL, philosopher, &phil[i]);

printf("Philosopher %d is thinking\n", i + 1);

}

// Wait for philosopher threads to complete (they never will in this infinite loop)

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

pthread\_join(thread\_id[i], NULL);

}

return 0;

}

**9.** **Write a C program to simulate Bankers algorithm for the purpose of deadlock Avoidance**

#include <stdio.h>

int main() {

int n, m, i, j, k;

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

scanf("%d", &n);

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

scanf("%d", &m);

int allocation[n][m];

int max[n][m];

int available[m];

int need[n][m];

int f[n], ans[n], ind = 0;

printf("Enter the Allocation Matrix:\n");

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

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

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

}

}

printf("Enter the MAX Matrix:\n");

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

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

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

}

}

printf("Enter the Available Resources:\n");

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

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

}

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

f[k] = 0;

}

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

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

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

}

}

printf("\nNeed Matrix:\n");

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

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

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

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

}

printf("\n");

}

int y = 0;

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

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

if (f[i] == 0) {

int flag = 0;

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

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

flag = 1;

break;

}

}

if (flag == 0) {

ans[ind++] = i;

for (y = 0; y < m; y++) {

available[y] += allocation[i][y];

}

f[i] = 1;

}

}

}

}

int flag = 1;

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

if (f[i] == 0) {

flag = 0;

printf("\nThe following system is not safe\n");

break;

}

}

if (flag == 1) {

printf("\nFollowing is the SAFE Sequence:\n");

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

printf(" P%d ->", ans[i]);

}

printf(" P%d\n", ans[n - 1]);

}

return 0;

}

**Write a C program to simulate deadlock detection.**

#include <stdio.h>

static int mark[20];

int i, j, np, nr;

int main() {

int alloc[10][10], request[10][10], avail[10], r[10], w[10];

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

scanf("%d", &np);

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

scanf("%d", &nr);

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

printf("\nTotal Amount of the Resource R%d: ", i + 1);

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

}

printf("\nEnter the request matrix:\n");

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

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

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

printf("\nEnter the allocation matrix:\n");

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

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

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

/\* Available Resource calculation \*/

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

avail[j] = r[j];

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

avail[j] -= alloc[i][j];

}

}

// Initialize mark[] array to 0 (not processed)

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

mark[i] = 0;

// Mark processes with zero allocation

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

int count = 0;

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

if(alloc[i][j] == 0)

count++;

else

break;

}

if(count == nr)

mark[i] = 1;

}

// Initialize W with avail

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

w[j] = avail[j];

// Mark processes with request less than or equal to W

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

int canbeprocessed = 0;

if(mark[i] != 1) {

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

if(request[i][j] <= w[j])

canbeprocessed = 1;

else {

canbeprocessed = 0;

break;

}

}

if(canbeprocessed) {

mark[i] = 1;

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

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

}

}

}

// Checking for unmarked processes

int deadlock = 0;

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

if(mark[i] != 1) {

deadlock = 1;

printf("\nProcess P%d is in deadlock state.", i + 1);

}

}

if(deadlock)

printf("\nDeadlock detected");

else

printf("\nNo Deadlock possible");

return 0;

}

**Write a C program to simulate the following contiguous memory allocation**

**techniques**

**11.          Worst-fit**

**12.          Best-fit**

**13.          First-fit**

#include <stdio.h>

#define max 25

void firstFit(int b[], int nb, int f[], int nf);

void worstFit(int b[], int nb, int f[], int nf);

void bestFit(int b[], int nb, int f[], int nf);

int main()

{

int b[max], f[max], nb, nf;

printf("Memory Management Schemes\n");

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

scanf("%d", &nb);

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

scanf("%d", &nf);

printf("\nEnter the size of the blocks:\n");

for (int i = 0; i < nb; i++) // Changed from i = 1 to i = 0 (indexing from 0)

{

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

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

}

printf("\nEnter the size of the files:\n");

for (int i = 0; i < nf; i++) // Changed from i = 1 to i = 0 (indexing from 0)

{

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

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

}

printf("\nMemory Management Scheme - First Fit");

firstFit(b, nb, f, nf);

printf("\n\nMemory Management Scheme - Worst Fit");

worstFit(b, nb, f, nf);

printf("\n\nMemory Management Scheme - Best Fit");

bestFit(b, nb, f, nf);

return 0;

}

void firstFit(int b[], int nb, int f[], int nf)

{

int bf[max] = {0}; // Keeps track of which blocks are allocated

int ff[max] = {0}; // Stores block allocated to each file

int frag[max], i, j;

for (i = 0; i < nf; i++) // Changed from i = 1 to i = 0 (indexing from 0)

{

for (j = 0; j < nb; j++) // Changed from j = 1 to j = 0 (indexing from 0)

{

if (bf[j] != 1 && b[j] >= f[i]) // Block is free and can accommodate the file

{

ff[i] = j;

bf[j] = 1; // Block is now allocated

frag[i] = b[j] - f[i]; // Calculate fragmentation

break;

}

}

}

printf("\nFile\_no:\tFile\_size:\tBlock\_no:\tBlock\_size:\tFragment");

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

{

if (ff[i] != 0) // If file is allocated to a block

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

else

printf("\nFile %d cannot be allocated to any block", i + 1);

}

}

void worstFit(int b[], int nb, int f[], int nf)

{

int bf[max] = {0}; // Keeps track of which blocks are allocated

int ff[max] = {0}; // Stores block allocated to each file

int frag[max], i, j, temp, highest = 0;

for (i = 0; i < nf; i++) // Changed from i = 1 to i = 0 (indexing from 0)

{

for (j = 0; j < nb; j++) // Changed from j = 1 to j = 0 (indexing from 0)

{

if (bf[j] != 1)

{

temp = b[j] - f[i]; // Remaining space in block after allocating file

if (temp >= 0 && highest < temp)

{

ff[i] = j;

highest = temp;

}

}

}

frag[i] = highest;

bf[ff[i]] = 1; // Block is now allocated

highest = 0; // Reset for next iteration

}

printf("\nFile\_no:\tFile\_size:\tBlock\_no:\tBlock\_size:\tFragment");

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

{

if (ff[i] != 0) // If file is allocated to a block

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

else

printf("\nFile %d cannot be allocated to any block", i + 1);

}

}

void bestFit(int b[], int nb, int f[], int nf)

{

int bf[max] = {0}; // Keeps track of which blocks are allocated

int ff[max] = {0}; // Stores block allocated to each file

int frag[max], i, j, temp, lowest = 10000;

for (i = 0; i < nf; i++) // Changed from i = 1 to i = 0 (indexing from 0)

{

for (j = 0; j < nb; j++) // Changed from j = 1 to j = 0 (indexing from 0)

{

if (bf[j] != 1)

{

temp = b[j] - f[i]; // Remaining space in block after allocating file

if (temp >= 0 && lowest > temp)

{

ff[i] = j;

lowest = temp;

}

}

}

frag[i] = lowest;

bf[ff[i]] = 1; // Block is now allocated

lowest = 10000; // Reset for next iteration

}

printf("\nFile\_no:\tFile\_size:\tBlock\_no:\tBlock\_size:\tFragment");

for (i = 0; i < nf && ff[i] != 0; i++)

{

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

}

}