**VISVESVARAYA TECHNOLOGICAL UNIVERSITY**

**“JnanaSangama”, Belgaum -590014, Karnataka.**

****

**LAB REPORT**

**on**

**OPERATING SYSTEMS**

***Submitted by***

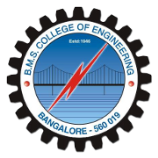
**Aashirvaad Kumar S(1BM23CS400)**

***in partial fulfilment for the award of the degree of***

**BACHELOR OF ENGINEERING**

***in***

**COMPUTER SCIENCE AND ENGINEERING**



**B.M.S. COLLEGE OF ENGINEERING**

**(Autonomous Institution under VTU)**

**BENGALURU-560019**

**Apr-2024 to Aug-2024**

**B. M. S. College of Engineering,**

**Bull Temple Road, Bangalore 560019**

(Affiliated To Visvesvaraya Technological University, Belgaum)

**Department of Computer Science and Engineering**



**CERTIFICATE**

This is to certify that the Lab work entitled “OPERATING SYSTEMS – 23CS4PCOPS” carried out by **Aashirvaad Kumar S(1BM23CS400),** who is bonafide student of **B. M. S. College of Engineering.** It is in partial fulfilment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum during the year 2024. The Lab report has been approved as it satisfies the academic requirements in respect of a **OPERATING SYSTEMS - (23CS4PCOPS)** work prescribed for the said degree.

**Sowmya T**  **Dr. Jyothi S Nayak**

Assistant Professor Professor and Head Department of CSE Department of CSE

BMSCE, Bengaluru BMSCE, Bengaluru

**Index Sheet**

|  |  |  |
| --- | --- | --- |
| **Sl. No.** | **Experiment Title** | **Page No.** |
| 1 | Write a C program to simulate the following non-pre-emptive CPU scheduling algorithm to find turnaround time and waiting time.  →FCFS  → SJF (pre-emptive & Non-preemptive) | 1 |
| 2 | Write a C program to simulate the following CPU scheduling algorithm to find turnaround time and waiting time.  → Priority (pre-emptive & Non-pre-emptive)  →Round Robin (Experiment with different quantum sizes for RR algorithm) | 11 |
| 3 | Write a C program to simulate multi-level queue scheduling algorithm considering the following scenario. All the processes in the system are divided into two categories – system processes and user processes. System processes are to be given higher priority than user processes. Use FCFS scheduling for the  processes in each queue. | 22 |
| 4 | Write a C program to simulate Real-Time CPU Scheduling algorithms:  a) Rate- Monotonic b) Earliest-deadline First c) Proportional scheduling | 27 |
| 5 | Write a C program to simulate producer-consumer problem using semaphores. | 35 |
| 6 | Write a C program to simulate the concept of Dining-Philosophers problem. | 38 |
| 7 | Write a C program to simulate Bankers algorithm for the purpose of deadlock avoidance. | 41 |
| 8 | Write a C program to simulate deadlock detection | 45 |
| 9 | Write a C program to simulate the following contiguous memory allocation techniques  a) Worst-fit b) Best-fit c) First-fit | 51 |
| 10 | Write a C program to simulate page replacement algorithms  a) FIFO b) LRU c) Optimal | 58 |

**Program -1**

**Write a C program to simulate the following non-pre-emptive CPU scheduling algorithm to find turnaround time and waiting time.**

**→ FCFS**

#include <stdio.h>

struct Process {

int pid; // Process ID

int burst\_time; // Burst time

int arrival\_time; // Arrival time

int waiting\_time; // Waiting time

int turnaround\_time; // Turnaround time

};

void findWaitingTime(struct Process proc[], int n) {

int service\_time[n];

service\_time[0] = proc[0].arrival\_time;

proc[0].waiting\_time = 0;

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

service\_time[i] = service\_time[i-1] + proc[i-1].burst\_time;

proc[i].waiting\_time = service\_time[i] - proc[i].arrival\_time;

if (proc[i].waiting\_time < 0)

proc[i].waiting\_time = 0;

}

}

void findTurnaroundTime(struct Process proc[], int n) {

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

proc[i].turnaround\_time = proc[i].burst\_time + proc[i].waiting\_time;

}

void findAverageTime(struct Process proc[], int n) {

int total\_waiting\_time = 0, total\_turnaround\_time = 0;

findWaitingTime(proc, n);

findTurnaroundTime(proc, n);

printf("Processes Burst time Arrival time Waiting time Turnaround time\n");

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

total\_waiting\_time += proc[i].waiting\_time;

total\_turnaround\_time += proc[i].turnaround\_time;

printf(" %d \t\t%d \t\t%d \t\t%d \t\t%d\n", proc[i].pid, proc[i].burst\_time, proc[i].arrival\_time, proc[i].waiting\_time, proc[i].turnaround\_time);

}

printf("Average waiting time = %.2f\n", (float)total\_waiting\_time / (float)n);

printf("Average turnaround time = %.2f\n", (float)total\_turnaround\_time / (float)n);

}

int main() {

struct Process proc[] = {{1, 10, 0}, {2, 5, 1}, {3, 8, 2}};

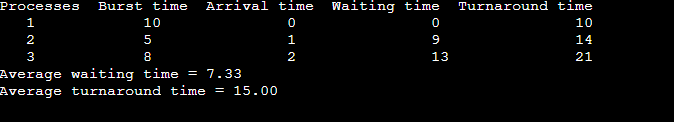
int n = sizeof(proc) / sizeof(proc[0]);

findAverageTime(proc, n);

return 0;

}

**Output**

****

**→ SJF (pre-emptive)**

#include <stdio.h>

struct Process {

int pid;

int burst\_time;

int arrival\_time;

int waiting\_time;

int turnaround\_time;

};

void findWaitingTime(struct Process proc[], int n) {

int complete = 0, t = 0, minm = 10000;

int shortest = 0, finish\_time;

int check = 0;

int rt[n];

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

rt[i] = proc[i].burst\_time;

while (complete != n) {

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

if ((proc[j].arrival\_time <= t) && (rt[j] < minm) && rt[j] > 0) {

minm = rt[j];

shortest = j;

check = 1;

}

}

if (check == 0) {

t++;

continue;

}

rt[shortest]--;

minm = rt[shortest];

if (minm == 0)

minm = 10000;

if (rt[shortest] == 0) {

complete++;

check = 0;

finish\_time = t + 1;

proc[shortest].waiting\_time = finish\_time - proc[shortest].burst\_time - proc[shortest].arrival\_time;

if (proc[shortest].waiting\_time < 0)

proc[shortest].waiting\_time = 0;

}

t++;

}

}

void findTurnaroundTime(struct Process proc[], int n) {

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

proc[i].turnaround\_time = proc[i].burst\_time + proc[i].waiting\_time;

}

void findAverageTime(struct Process proc[], int n) {

int total\_waiting\_time = 0, total\_turnaround\_time = 0;

findWaitingTime(proc, n);

findTurnaroundTime(proc, n);

printf("Processes Burst time Arrival time Waiting time Turnaround time\n");

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

total\_waiting\_time += proc[i].waiting\_time;

total\_turnaround\_time += proc[i].turnaround\_time;

printf(" %d \t\t%d \t\t%d \t\t%d \t\t%d\n", proc[i].pid, proc[i].burst\_time, proc[i].arrival\_time, proc[i].waiting\_time, proc[i].turnaround\_time);

}

printf("Average waiting time = %.2f\n", (float)total\_waiting\_time / (float)n);

printf("Average turnaround time = %.2f\n", (float)total\_turnaround\_time / (float)n);

}

int main() {

struct Process proc[] = {{1, 6, 0}, {2, 8, 1}, {3, 7, 2}, {4, 3, 3}};

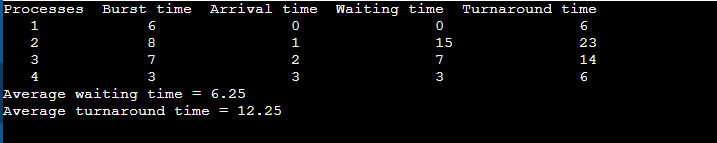
int n = sizeof(proc) / sizeof(proc[0]);

findAverageTime(proc, n);

return 0;

}

**OUTPUT**

****

**→ SJF (Non-preemptive)**

#include <stdio.h>

struct Process {

int pid;

int burst\_time;

int arrival\_time;

int waiting\_time;

int turnaround\_time;

};

void findWaitingTime(struct Process proc[], int n) {

int rt[n];

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

rt[i] = proc[i].burst\_time;

int complete = 0, t = 0, minm = 10000;

int shortest = 0, finish\_time;

int check = 0;

while (complete != n) {

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

if ((proc[j].arrival\_time <= t) && (rt[j] < minm) && rt[j] > 0) {

minm = rt[j];

shortest = j;

check = 1;

}

}

if (check == 0) {

t++;

continue;

}

rt[shortest]--;

minm = rt[shortest];

if (minm == 0)

minm = 10000;

if (rt[shortest] == 0) {

complete++;

check = 0;

finish\_time = t + 1;

proc[shortest].waiting\_time = finish\_time - proc[shortest].burst\_time - proc[shortest].arrival\_time;

if (proc[shortest].waiting\_time < 0)

proc[shortest].waiting\_time = 0;

}

t++;

}

}

void findTurnaroundTime(struct Process proc[], int n) {

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

proc[i].turnaround\_time = proc[i].burst\_time + proc[i].waiting\_time;

}

void findAverageTime(struct Process proc[], int n) {

int total\_waiting\_time = 0, total\_turnaround\_time = 0;

findWaitingTime(proc, n);

findTurnaroundTime(proc, n);

printf("Processes Burst time Arrival time Waiting time Turnaround time\n");

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

total\_waiting\_time += proc[i].waiting\_time;

total\_turnaround\_time += proc[i].turnaround\_time;

printf(" %d \t\t%d \t\t%d \t\t%d \t\t%d\n", proc[i].pid, proc[i].burst\_time, proc[i].arrival\_time, proc[i].waiting\_time, proc[i].turnaround\_time);

}

printf("Average waiting time = %.2f\n", (float)total\_waiting\_time / (float)n);

printf("Average turnaround time = %.2f\n", (float)total\_turnaround\_time / (float)n);

}

int main() {

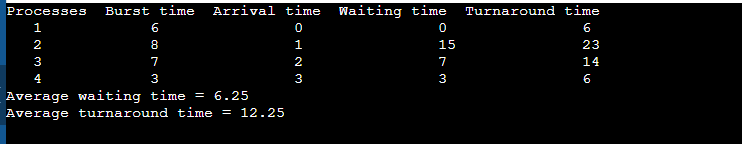
struct Process proc[] = {{1, 6, 0}, {2, 8, 1}, {3, 7, 2}, {4, 3, 3}};

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

findAverageTime(proc, n);

return 0;

}

****

**Program-2**

**Write a C program to simulate the following CPU scheduling to find turnaround time and waiting time.**

**→ Priority (pre-emptive)**

#include <stdio.h>

struct Process {

int pid;

int burst\_time;

int arrival\_time;

int priority;

int waiting\_time;

int turnaround\_time;

};

void findWaitingTime(struct Process proc[], int n) {

int rt[n];

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

rt[i] = proc[i].burst\_time;

int complete = 0, t = 0, minm = 10000;

int shortest = 0, finish\_time;

int check = 0;

while (complete != n) {

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

if ((proc[j].arrival\_time <= t) && (proc[j].priority < minm) && rt[j] > 0) {

minm = proc[j].priority;

shortest = j;

check = 1;

}

}

if (check == 0) {

t++;

continue;

}

rt[shortest]--;

minm = proc[shortest].priority;

if (rt[shortest] == 0) {

complete++;

check = 0;

finish\_time = t + 1;

proc[shortest].waiting\_time = finish\_time - proc[shortest].burst\_time - proc[shortest].arrival\_time;

if (proc[shortest].waiting\_time < 0)

proc[shortest].waiting\_time = 0;

minm = 10000;

}

t++;

}

}

void findTurnaroundTime(struct Process proc[], int n) {

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

proc[i].turnaround\_time = proc[i].burst\_time + proc[i].waiting\_time;

}

void findAverageTime(struct Process proc[], int n) {

int total\_waiting\_time = 0, total\_turnaround\_time = 0;

findWaitingTime(proc, n);

findTurnaroundTime(proc, n);

printf("Processes Burst time Arrival time Priority Waiting time Turnaround time\n");

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

total\_waiting\_time += proc[i].waiting\_time;

total\_turnaround\_time += proc[i].turnaround\_time;

printf(" %d \t\t%d \t\t%d \t\t%d \t\t%d \t\t%d\n", proc[i].pid, proc[i].burst\_time, proc[i].arrival\_time, proc[i].priority, proc[i].waiting\_time, proc[i].turnaround\_time);

}

printf("Average waiting time = %.2f\n", (float)total\_waiting\_time / (float)n);

printf("Average turnaround time = %.2f\n", (float)total\_turnaround\_time / (float)n);

}

int main() {

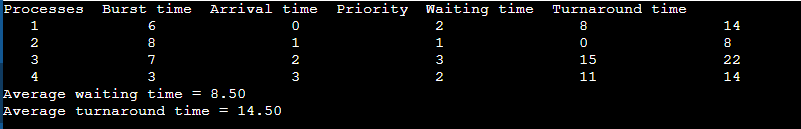
struct Process proc[] = {{1, 6, 0, 2}, {2, 8, 1, 1}, {3, 7, 2, 3}, {4, 3, 3, 2}};

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

findAverageTime(proc, n);

return 0;

}

****

**→ Priority (Non-preemptive)**

#include <stdio.h>

struct Process {

int pid;

int burst\_time;

int arrival\_time;

int priority;

int waiting\_time;

int turnaround\_time;

};

void findWaitingTime(struct Process proc[], int n) {

int completed[n];

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

completed[i] = 0;

int t = 0;

int completed\_count = 0;

while (completed\_count < n) {

int min\_priority = 10000;

int idx = -1;

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

if (proc[i].arrival\_time <= t && !completed[i] && proc[i].priority < min\_priority) {

min\_priority = proc[i].priority;

idx = i;

}

}

if (idx != -1) {

t += proc[idx].burst\_time;

proc[idx].waiting\_time = t - proc[idx].burst\_time - proc[idx].arrival\_time;

if (proc[idx].waiting\_time < 0)

proc[idx].waiting\_time = 0;

completed[idx] = 1;

completed\_count++;

} else {

t++;

}

}

}

void findTurnaroundTime(struct Process proc[], int n) {

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

proc[i].turnaround\_time = proc[i].burst\_time + proc[i].waiting\_time;

}

void findAverageTime(struct Process proc[], int n) {

int total\_waiting\_time = 0, total\_turnaround\_time = 0;

findWaitingTime(proc, n);

findTurnaroundTime(proc, n);

printf("Processes Burst time Arrival time Priority Waiting time Turnaround time\n");

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

total\_waiting\_time += proc[i].waiting\_time;

total\_turnaround\_time += proc[i].turnaround\_time;

printf(" %d \t\t%d \t\t%d \t\t%d \t\t%d \t\t%d\n", proc[i].pid, proc[i].burst\_time, proc[i].arrival\_time, proc[i].priority, proc[i].waiting\_time, proc[i].turnaround\_time);

}

printf("Average waiting time = %.2f\n", (float)total\_waiting\_time / (float)n);

printf("Average turnaround time = %.2f\n", (float)total\_turnaround\_time / (float)n);

}

int main() {

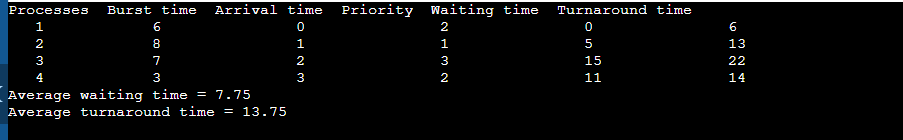
struct Process proc[] = {{1, 6, 0, 2}, {2, 8, 1, 1}, {3, 7, 2, 3}, {4, 3, 3, 2}};

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

findAverageTime(proc, n);

return 0;

}

****

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

#include <stdio.h>

struct Process {

int pid;

int burst\_time;

int arrival\_time;

int priority;

int waiting\_time;

int turnaround\_time;

};

void findWaitingTime(struct Process proc[], int n) {

int completed[n];

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

completed[i] = 0;

int t = 0;

int completed\_count = 0;

while (completed\_count < n) {

int min\_priority = 10000;

int idx = -1;

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

if (proc[i].arrival\_time <= t && !completed[i] && proc[i].priority < min\_priority) {

min\_priority = proc[i].priority;

idx = i;

}

}

if (idx != -1) {

t += proc[idx].burst\_time;

proc[idx].waiting\_time = t - proc[idx].burst\_time - proc[idx].arrival\_time;

if (proc[idx].waiting\_time < 0)

proc[idx].waiting\_time = 0;

completed[idx] = 1;

completed\_count++;

} else {

t++;

}

}

}

void findTurnaroundTime(struct Process proc[], int n) {

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

proc[i].turnaround\_time = proc[i].burst\_time + proc[i].waiting\_time;

}

void findAverageTime(struct Process proc[], int n) {

int total\_waiting\_time = 0, total\_turnaround\_time = 0;

findWaitingTime(proc, n);

findTurnaroundTime(proc, n);

printf("Processes Burst time Arrival time Priority Waiting time Turnaround time\n");

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

total\_waiting\_time += proc[i].waiting\_time;

total\_turnaround\_time += proc[i].turnaround\_time;

printf(" %d \t\t%d \t\t%d \t\t%d \t\t%d \t\t%d\n", proc[i].pid, proc[i].burst\_time, proc[i].arrival\_time, proc[i].priority, proc[i].waiting\_time, proc[i].turnaround\_time);

}

printf("Average waiting time = %.2f\n", (float)total\_waiting\_time / (float)n);

printf("Average turnaround time = %.2f\n", (float)total\_turnaround\_time / (float)n);

}

int main() {

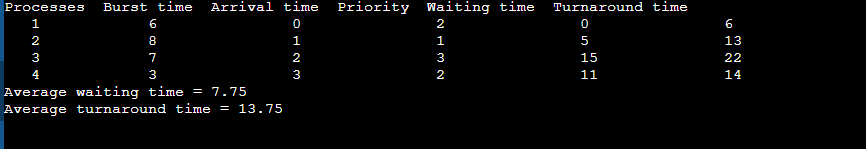
struct Process proc[] = {{1, 6, 0, 2}, {2, 8, 1, 1}, {3, 7, 2, 3}, {4, 3, 3, 2}};

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

findAverageTime(proc, n);

return 0;

}

****

**Program 3**

**Write a C program to simulate multi-level queue scheduling algorithm considering the following scenario. All the processes in the system are divided into two categories – system processes and user processes. System processes are to be given higher priority than user processes. Use FCFS scheduling for the processes in each queue**

#include <stdio.h>

#define MAX\_PROCESSES 100

struct Process {

int pid;

int burst\_time;

int arrival\_time;

int waiting\_time;

int turnaround\_time;

int is\_system\_process; // 1 for system process, 0 for user process

};

void sortProcessesByArrival(struct Process proc[], int n) {

struct Process temp;

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

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

if (proc[i].arrival\_time > proc[j].arrival\_time) {

temp = proc[i];

proc[i] = proc[j];

proc[j] = temp;

}

}

}

}

void calculateWaitingTime(struct Process proc[], int n) {

int current\_time = 0;

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

if (current\_time < proc[i].arrival\_time) {

current\_time = proc[i].arrival\_time;

}

proc[i].waiting\_time = current\_time - proc[i].arrival\_time;

current\_time += proc[i].burst\_time;

}

}

void calculateTurnaroundTime(struct Process proc[], int n) {

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

proc[i].turnaround\_time = proc[i].burst\_time + proc[i].waiting\_time;

}

}

void printProcesses(struct Process proc[], int n) {

int total\_waiting\_time = 0;

int total\_turnaround\_time = 0;

printf("Processes Burst time Arrival time Waiting time Turnaround time Type\n");

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

total\_waiting\_time += proc[i].waiting\_time;

total\_turnaround\_time += proc[i].turnaround\_time;

printf(" %d \t\t%d \t\t%d \t\t%d \t\t%d \t\t%s\n", proc[i].pid, proc[i].burst\_time, proc[i].arrival\_time, proc[i].waiting\_time, proc[i].turnaround\_time, proc[i].is\_system\_process ? "System" : "User");

}

printf("Average waiting time = %.2f\n", (float)total\_waiting\_time / n);

printf("Average turnaround time = %.2f\n", (float)total\_turnaround\_time / n);

}

int main() {

struct Process proc[MAX\_PROCESSES];

int n;

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

scanf("%d", &n);

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

printf("Enter process ID, burst time, arrival time, and type (1 for system, 0 for user) for process %d: ", i + 1);

scanf("%d %d %d %d", &proc[i].pid, &proc[i].burst\_time, &proc[i].arrival\_time, &proc[i].is\_system\_process);

}

struct Process system\_queue[MAX\_PROCESSES];

struct Process user\_queue[MAX\_PROCESSES];

int system\_count = 0, user\_count = 0;

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

if (proc[i].is\_system\_process) {

system\_queue[system\_count++] = proc[i];

} else {

user\_queue[user\_count++] = proc[i];

}

}

sortProcessesByArrival(system\_queue, system\_count);

sortProcessesByArrival(user\_queue, user\_count);

printf("\nSystem Queue:\n");

calculateWaitingTime(system\_queue, system\_count);

calculateTurnaroundTime(system\_queue, system\_count);

printProcesses(system\_queue, system\_count);

printf("\nUser Queue:\n");

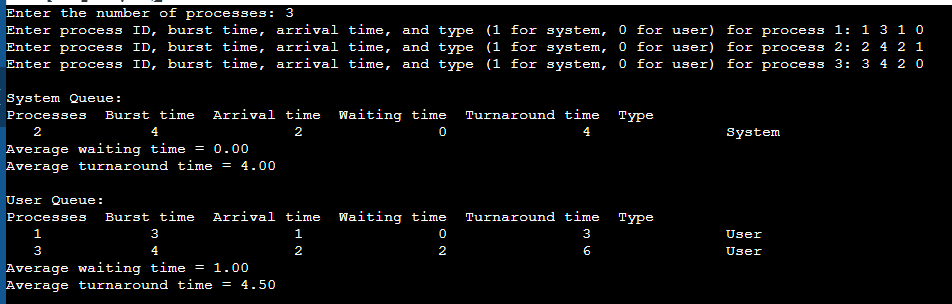
calculateWaitingTime(user\_queue, user\_count);

calculateTurnaroundTime(user\_queue, user\_count);

printProcesses(user\_queue, user\_count);

return 0;

}

****

**Program 4**

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

**→ Rate- Monotonic**

#include <stdio.h>

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

    wt[0] = 0;

    for (int i = 1; i < n; i++) {

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

    }

}

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];

    }

}

void findAvgTime(int processes[], int n, int bt[], int period[]) {

    int wt[n], tat[n];

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

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

    printf("Processes   Burst time   Waiting time   Turnaround time   Period\n");

    for (int i = 0; i < n; i++) {

        printf(" %d ", (i + 1));

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

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

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

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

    }

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

    for (int i = 0; i < n; i++) {

        total\_wt += wt[i];

        total\_tat += tat[i];

    }

    printf("Average waiting time = %.2f\n", (float)total\_wt / (float)n);

    printf("Average turnaround time = %.2f\n", (float)total\_tat / (float)n);

}

void rateMonotonicScheduling(int processes[], int n, int bt[], int period[]) {

    // Sort by period

    for (int i = 0; i < n - 1; i++) {

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

            if (period[j] > period[j + 1]) {

                int temp = period[j];

                period[j] = period[j + 1];

                period[j + 1] = temp;

                temp = bt[j];

                bt[j] = bt[j + 1];

                bt[j + 1] = temp;

                temp = processes[j];

                processes[j] = processes[j + 1];

                processes[j + 1] = temp;

            }

        }

    }

  findAvgTime(processes, n, bt, period);

}

int main() {

    int processes[] = {1, 2, 3};

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

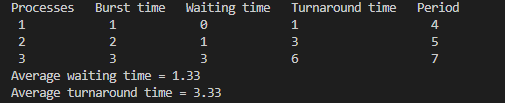
    int burst\_time[] = {3, 1, 2};

    int period[] = {7, 4, 5};

    rateMonotonicScheduling(processes, n, burst\_time, period);

    return 0;

}



**→ Earliest-deadline First**

#include <stdio.h>

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

    wt[0] = 0;

    for (int i = 1; i < n; i++) {

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

    }

}

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];

    }

}

void findAvgTime(int processes[], int n, int bt[], int deadline[]) {

    int wt[n], tat[n];

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

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

    printf("Processes   Burst time   Waiting time   Turnaround time   Deadline\n");

    for (int i = 0; i < n; i++) {

        printf(" %d ", (i + 1));

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

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

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

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

    }

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

    for (int i = 0; i < n; i++) {

        total\_wt += wt[i];

        total\_tat += tat[i];

    }

    printf("Average waiting time = %.2f\n", (float)total\_wt / (float)n);

    printf("Average turnaround time = %.2f\n", (float)total\_tat / (float)n);

}

void earliestDeadlineFirstScheduling(int processes[], int n, int bt[], int deadline[]) {

    // Sort by deadline

    for (int i = 0; i < n - 1; i++) {

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

            if (deadline[j] > deadline[j + 1]) {

                int temp = deadline[j];

                deadline[j] = deadline[j + 1];

                deadline[j + 1] = temp;

                temp = bt[j];

                bt[j] = bt[j + 1];

                bt[j + 1] = temp;

                temp = processes[j];

                processes[j] = processes[j + 1];

                processes[j + 1] = temp;

            }

        }

    }

    findAvgTime(processes, n, bt, deadline);

}

int main() {

    int processes[] = {1, 2, 3};

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

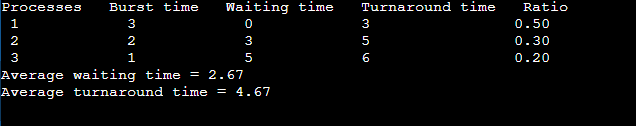
    int burst\_time[] = {3, 1, 2};

    int deadline[] = {7, 4, 5};

    earliestDeadlineFirstScheduling(processes, n, burst\_time, deadline);

    return 0;

}



**→ Proportional scheduling**

#include <stdio.h>

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

    wt[0] = 0;

    for (int i = 1; i < n; i++) {

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

    }

}

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];

    }

}

void findAvgTime(int processes[], int n, int bt[], float ratio[]) {

    int wt[n], tat[n];

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

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

    printf("Processes   Burst time   Waiting time   Turnaround time   Ratio\n");

    for (int i = 0; i < n; i++) {

        printf(" %d ", (i + 1));

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

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

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

        printf("               %.2f\n", ratio[i]);

    }

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

    for (int i = 0; i < n; i++) {

        total\_wt += wt[i];

        total\_tat += tat[i];

    }

    printf("Average waiting time = %.2f\n", (float)total\_wt / (float)n);

    printf("Average turnaround time = %.2f\n", (float)total\_tat / (float)n);

}

void proportionalScheduling(int processes[], int n, int bt[], float ratio[]) {

    for (int i = 0; i < n - 1; i++) {

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

            if (ratio[j] < ratio[j + 1]) {

                float temp = ratio[j];

                ratio[j] = ratio[j + 1];

                ratio[j + 1] = temp;

                int temp\_bt = bt[j];

                bt[j] = bt[j + 1];

                bt[j + 1] = temp\_bt;

                int temp\_proc = processes[j];

                processes[j] = processes[j + 1];

                processes[j + 1] = temp\_proc;

            }

        }

    }

    findAvgTime(processes, n, bt, ratio);

}

int main() {

    int processes[] = {1, 2, 3};

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

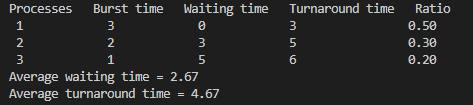
    int burst\_time[] = {3, 1, 2};

    float ratio[] = {0.5, 0.2, 0.3}; // Example ratios

    proportionalScheduling(processes, n, burst\_time, ratio);

    return 0;

}



**Program 5**

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

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>

#define BUFFER\_SIZE 5

int buffer[BUFFER\_SIZE];

int in = 0, out = 0;

sem\_t empty;

sem\_t full;

pthread\_mutex\_t mutex;

void \*producer(void \*param) {

    int item;

    while (1) {

        item = rand() % 100;

        sem\_wait(&empty);

        pthread\_mutex\_lock(&mutex);

        buffer[in] = item;

        printf("Producer produced %d at %d\n", item, in);

        in = (in + 1) % BUFFER\_SIZE;

        pthread\_mutex\_unlock(&mutex);

        sem\_post(&full);

        sleep(1);

    }

}

void \*consumer(void \*param) {

    int item;

    while (1) {

        sem\_wait(&full);

        pthread\_mutex\_lock(&mutex);

        item = buffer[out];

        printf("Consumer consumed %d from %d\n", item, out);

        out = (out + 1) % BUFFER\_SIZE;

        pthread\_mutex\_unlock(&mutex);

        sem\_post(&empty);

        sleep(1);

    }

}

int main() {

    pthread\_t tid1, tid2;

    pthread\_attr\_t attr;

    pthread\_attr\_init(&attr);

    pthread\_mutex\_init(&mutex, NULL);

    sem\_init(&empty, 0, BUFFER\_SIZE);

    sem\_init(&full, 0, 0);

    pthread\_create(&tid1, &attr, producer, NULL);

    pthread\_create(&tid2, &attr, consumer, NULL);

    pthread\_join(tid1, NULL);

    pthread\_join(tid2, NULL);

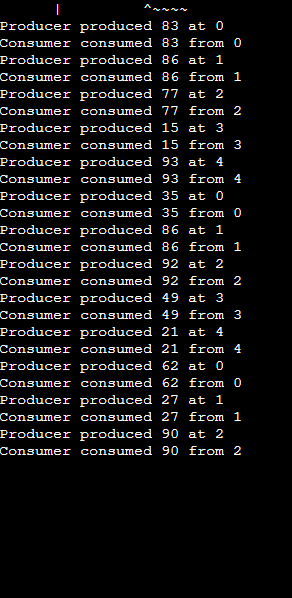
    pthread\_mutex\_destroy(&mutex);

    sem\_destroy(&empty);

    sem\_destroy(&full);

    return 0;

}



**Program 6**

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

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>

#define N

sem\_t forks[N];

sem\_t mutex;

void \*philosopher(void \*num) {

int id = \*(int \*)num;

while (1) {

printf("Philosopher %d is thinking.\n", id);

sleep(1);

sem\_wait(&mutex);

sem\_wait(&forks[id]);

sem\_wait(&forks[(id + 1) % N]);

printf("Philosopher %d is eating.\n", id);

sleep(1);

sem\_post(&forks[id]); // Put down chopsticks

sem\_post(&forks[(id + 1) % N]);

sem\_post(&mutex);

printf("Philosopher %d is done eating and starts thinking again.\n", id);

sleep(1);

}

}

int main() {

pthread\_t tid[N];

int ids[N];

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

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

sem\_init(&forks[i], 0, 1);

ids[i] = i;

}

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

pthread\_create(&tid[i], NULL, philosopher, &ids[i]);

}

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

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

}

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

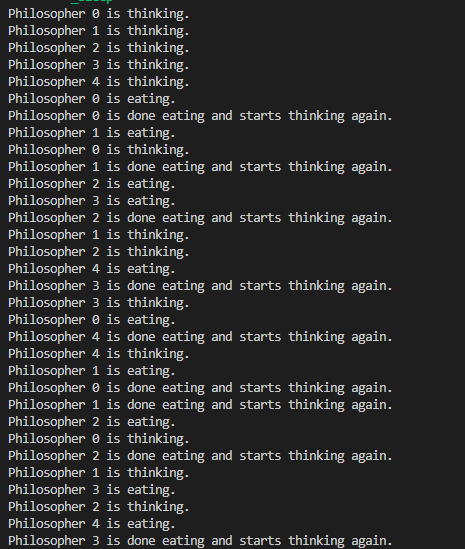
sem\_destroy(&forks[i]);

}

sem\_destroy(&mutex);

return 0;

}



**Program 7**

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

#include <stdio.h>

#include <stdbool.h>

#define MAX\_PROCESSES 5

#define MAX\_RESOURCES 3

int main() {

    int n, m, i, j, k;

    n = 5;

    m = 3;

    int alloc[MAX\_PROCESSES][MAX\_RESOURCES] = { { 0, 1, 0 },

                                                { 2, 0, 0 },

                                                { 3, 0, 2 },

                                                { 2, 1, 1 },

                                                { 0, 0, 2 } };

    int max[MAX\_PROCESSES][MAX\_RESOURCES] = { { 7, 5, 3 },

                                              { 3, 2, 2 },

                                              { 9, 0, 2 },

                                              { 2, 2, 2 },

                                              { 4, 3, 3 } };

    int avail[MAX\_RESOURCES] = { 3, 3, 2 };

    int f[MAX\_PROCESSES], ans[MAX\_PROCESSES], ind = 0;

    for (k = 0; k < n; k++) {

        f[k] = 0;

    }

    int need[MAX\_PROCESSES][MAX\_RESOURCES];

    for (i = 0; i < n; i++) {

        for (j = 0; j < m; j++) {

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

        }

    }

    printf("Need matrix:\n");

    for (i = 0; i < n; 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) {

                bool flag = true;

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

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

                        flag = false;

                        break;

                    } }

                if (flag) {

                    ans[ind++] = i;

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

                        avail[y] += alloc[i][y];

                    }

                    f[i] = 1;

                }

            }

        }

    }

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

    for (i = 0; i < n - 1; i++) {

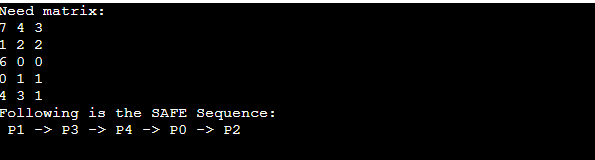
        printf(" P%d ->", ans[i]);

    }

    printf(" P%d\n", ans[n - 1]);

    return 0;

}



**Program 8**

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

#include <stdio.h>

#include <stdbool.h>

#define MAX\_PROCESSES 5

#define MAX\_RESOURCES 3

void printMatrices(int processes, int resources, int alloc[MAX\_PROCESSES][MAX\_RESOURCES], int max[MAX\_PROCESSES][MAX\_RESOURCES], int need[MAX\_PROCESSES][MAX\_RESOURCES], int avail[MAX\_RESOURCES]) {

printf("Allocation Matrix:\n");

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

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

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

}

printf("\n");

}

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

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

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

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

}

printf("\n");

}

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

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

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

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

}

printf("\n");

}

printf("\nAvailable Resources:\n");

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

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

}

printf("\n");

}

void deadlockDetection(int processes, int resources, int alloc[MAX\_PROCESSES][MAX\_RESOURCES], int max[MAX\_PROCESSES][MAX\_RESOURCES], int avail[MAX\_RESOURCES]) {

int need[MAX\_PROCESSES][MAX\_RESOURCES];

int work[MAX\_RESOURCES];

bool finish[MAX\_PROCESSES];

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

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

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

}

}

printMatrices(processes, resources, alloc, max, need, avail);

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

work[i] = avail[i];

}

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

finish[i] = false;

}

bool found;

do {

found = false;

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

if (!finish[i]) {

bool flag = true;

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

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

flag = false;

break;

}

}

if (flag) {

printf("\nProcess %d can be satisfied and is now finishing.\n", i);

for (int k = 0; k < resources; k++) {

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

}

finish[i] = true;

found = true;

printf("New Available Resources:\n");

for (int k = 0; k < resources; k++) {

printf("%d ", work[k]);

}

printf("\n");

}

}

}

} while (found);

bool deadlock = false;

printf("\nDeadlock Check:\n");

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

if (!finish[i]) {

deadlock = true;

printf("Process %d is in a deadlock.\n", i);

}

}

if (!deadlock) {

printf("No deadlock detected.\n");

}

}

int main() {

int processes = 5;

int resources = 3;

int alloc[MAX\_PROCESSES][MAX\_RESOURCES] = {

{ 0, 1, 0 },

{ 2, 0, 0 },

{ 3, 0, 2 },

{ 2, 1, 1 },

{ 0, 0, 2 }

};

int max[MAX\_PROCESSES][MAX\_RESOURCES] = {

{ 7, 5, 3 },

{ 3, 2, 2 },

{ 9, 0, 2 },

{ 2, 2, 2 },

{ 4, 3, 3 }

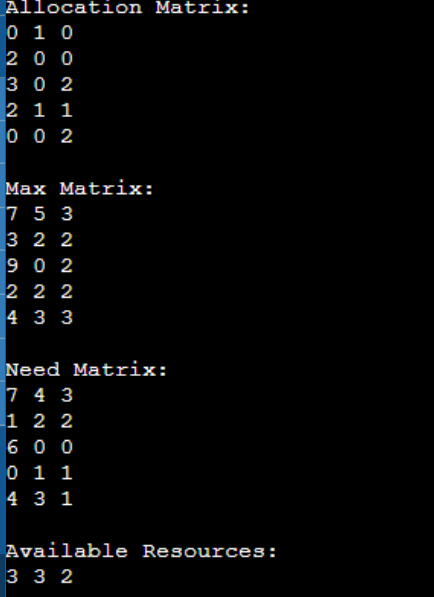
};

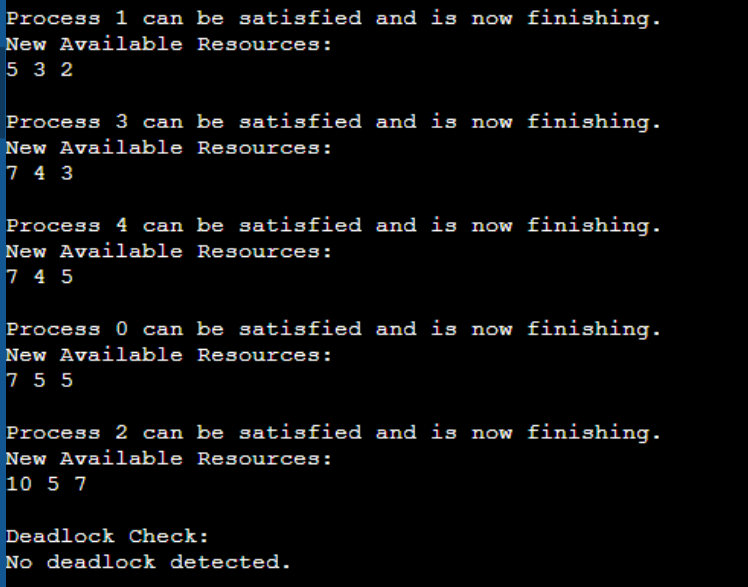
int avail[MAX\_RESOURCES] = { 3, 3, 2 }; // Available resources

deadlockDetection(processes, resources, alloc, max, avail);

return 0;

}





**Program 9**

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

**a) Worst-fit**

**b) Best-fit**

**c) First-fit**

#include <stdio.h>

#include <stdlib.h>

#define MAX 25

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

    int allocation[MAX];

    int allocated[MAX] = {0};

    for (int i = 0; i < nf; i++) {

        allocation[i] = -1;

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

            if (allocated[j] == 0 && b[j] >= f[i]) {

                allocation[i] = j;

                allocated[j] = 1;

                break;

            }

        }

    }

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

    for (int i = 0; i < nf; i++) {

        if (allocation[i] != -1)

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

        else

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

    }

}

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

    int allocation[MAX];

    int allocated[MAX] = {0};

    for (int i = 0; i < nf; i++) {

        int bestIdx = -1;

        allocation[i] = -1;

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

            if (allocated[j] == 0 && b[j] >= f[i]) {

                if (bestIdx == -1 || b[j] < b[bestIdx])

                    bestIdx = j;

            }

        }

        if (bestIdx != -1) {

            allocation[i] = bestIdx;

            allocated[bestIdx] = 1;

        }

    }

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

    for (int i = 0; i < nf; i++) {

        if (allocation[i] != -1)

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

        else

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

    }

}

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

    int allocation[MAX];

    int allocated[MAX] = {0};

    for (int i = 0; i < nf; i++) {

        int worstIdx = -1;

        allocation[i] = -1;

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

            if (allocated[j] == 0 && b[j] >= f[i]) {

                if (worstIdx == -1 || b[j] > b[worstIdx])

                    worstIdx = j;

            }

        }

        if (worstIdx != -1) {

            allocation[i] = worstIdx;

            allocated[worstIdx] = 1;

        }

    }

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

    for (int i = 0; i < nf; i++) {

        if (allocation[i] != -1)

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

        else

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

    }

}

int main() {

    int nb, nf, choice;

    printf("Memory Management Scheme");

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

    scanf("%d", &nb);

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

    scanf("%d", &nf);

    int b[nb], f[nf];

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

    for (int i = 0; i < nb; i++) {

        printf("Block %d: ", i + 1);

        scanf("%d", &b[i]);

    }

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

    for (int i = 0; i < nf; i++) {

        printf("File %d: ", i + 1);

        scanf("%d", &f[i]);

    }

    while (1) {

        printf("\n1. First Fit\n2. Best Fit\n3. Worst Fit\n4. Exit\n");

        printf("Enter your choice: ");

        scanf("%d", &choice);

        switch (choice) {

            case 1:

                printf("\n\tMemory Management Scheme - First Fit\n");

                firstFit(nb, nf, b, f);

                break;

            case 2:

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

                bestFit(nb, nf, b, f);

                break;

            case 3:

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

                worstFit(nb, nf, b, f);

                break;

            case 4:

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

                exit(0);

                break;

            default:

                printf("\nInvalid choice.\n");

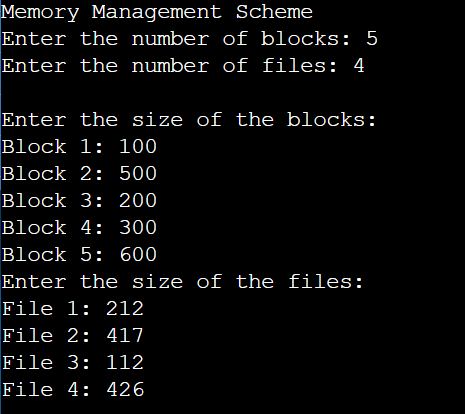
                break;

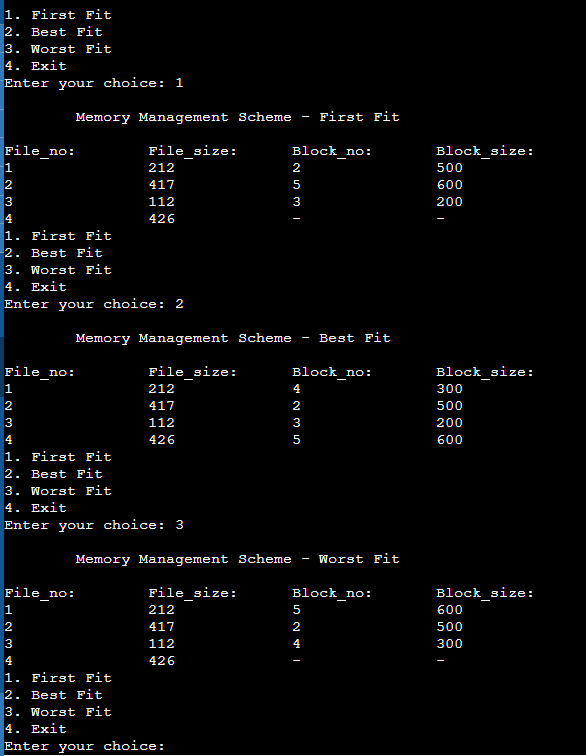
        }

    }

    return 0;

}





**10. Write a C program to simulate page replacement algorithms**

**a) FIFO**

**b) LRU**

**c) Optimal**

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#define MAX\_FRAMES 10

#define MAX\_PAGES 25

void fifo(int pages[], int n, int capacity) {

    int frame[MAX\_FRAMES], frameCount = 0, pageFaults = 0, frameIndex = 0;

    bool isPagePresent = false;

    for (int i = 0; i < n; i++) {

        isPagePresent = false;

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

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

                isPagePresent = true;

                break;

            }

        }

        if (isPagePresent == false) {

            if (frameCount < capacity) {

                frame[frameCount] = pages[i];

                frameCount++;

            } else {

                frame[frameIndex] = pages[i];

                frameIndex++;

                if (frameIndex >= capacity)

                    frameIndex = 0;

            }

            pageFaults++;

        }

    }

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

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

}

void lru(int pages[], int n, int capacity) {

    int frame[MAX\_FRAMES], frameCount = 0, pageFaults = 0, counter[MAX\_FRAMES];

    bool isPagePresent = false;

    for (int i = 0; i < n; i++) {

        isPagePresent = false;

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

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

                isPagePresent = true;

                counter[j] = i;

                break;

            }

        }

        if (isPagePresent == false) {

            if (frameCount < capacity) {

                frame[frameCount] = pages[i];

                counter[frameCount] = i;

                frameCount++;

            } else {

                int lru = 0;

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

                    if (counter[j] < counter[lru])

                        lru = j;

                }

                frame[lru] = pages[i];

                counter[lru] = i;

            }

            pageFaults++;

        }

    }

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

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

}

void optimal(int pages[], int n, int capacity) {

    int frame[MAX\_FRAMES], frameCount = 0, pageFaults = 0;

    bool isPagePresent = false;

    for (int i = 0; i < n; i++) {

        isPagePresent = false;

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

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

                isPagePresent = true;

                break;

            }

        }

        if (isPagePresent == false) {

            if (frameCount < capacity) {

                frame[frameCount] = pages[i];

                frameCount++;

            } else {

                int future[MAX\_FRAMES] = {0};

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

                    bool isFound = false;

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

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

                            future[j] = k;

                            isFound = true;

                            break;

                        }

                    }

                    if (isFound == false)

                        future[j] = n + 1;

                }

                int longest = 0;

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

                    if (future[j] > future[longest])

                        longest = j;

                }

                frame[longest] = pages[i];

            }

            pageFaults++;

        }

    }

    printf("\nOptimal Page Replacement Algorithm:\n");

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

}

int main() {

    int pages[MAX\_PAGES], n, capacity;

    printf("Page Replacement Algorithms\n");

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

    scanf("%d", &n);

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

    for (int i = 0; i < n; i++) {

        printf("Page %d: ", i + 1);

        scanf("%d", &pages[i]);

    }

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

    scanf("%d", &capacity);

    fifo(pages, n, capacity);

    lru(pages, n, capacity);

    optimal(pages, n, capacity);

    return 0;

}

