VISVESVARAYA TECHNOLOGICAL UNIVERSITY

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



**LAB REPORT**

**On**

**Operating Systems**

**Submitted by:**

**Ridhima Suhane**

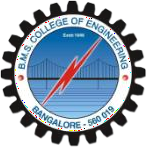
**(1BM23CS266)**

**In partial fulfillment 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)**

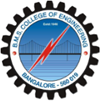
**BENGALURU - 560019**

**September2024-January2025**

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

**Bull Temple Road, Bangalore 560019**

**(Affiliated To Visvesvaraya Technological University, Belgaum) Department of Computer Science and Engineering**

****

This is to certify that the Lab work entitled **“OPERATING SYSTEMS”** carried out by **RIDHIMA SUHANE (1BM23CS266)** who is bonafide student of **B. M. S. College of Engineering**. It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum during the year 2024-25. The Lab report has been approved as it satisfies the academic requirements in respect of Operating System Lab - **(23CS4PCOPS)** work prescribed for the said degree.

**Prof. Sandhya A. Kulkarni Dr.Kavitha Sooda**

Assistant Professor Professor and Head

Department of CSE Department of CSE

BMSCE, Bengaluru BMSCE, Bengaluru

**Index Sheet**

|  |  |  |
| --- | --- | --- |
| **Sl. No.** | **Experiment Title** | **PageNo.** |
| 1 | LAB PROG-1  (first come first serve) | 4-5 |
| 2 | LAB PROG-2  (shortest job first) | 6-8 |
| 3 | LAB PROG-3  (priority & round robin) | 9-13 |
| 4 | LAB PROG-4  (multilevel queue) | 14-16 |
| 5 | LAB PROG-5  (rate monotonic, earliest dedline first, dining & producer consumer) | 17-25 |
| 6 | LAB PROG-6  (deadlock detection and avoidance) | 26-29 |
| 7 | LAB PROG-7  (memory management) | 30-32 |
| 8 | LAB PROG-8  (page replacement algorithm) | 33-36 |

**Lab program 1**

**Write a C program to simulate CPU scheduling algorithm with the following:  
 a) FCFS**

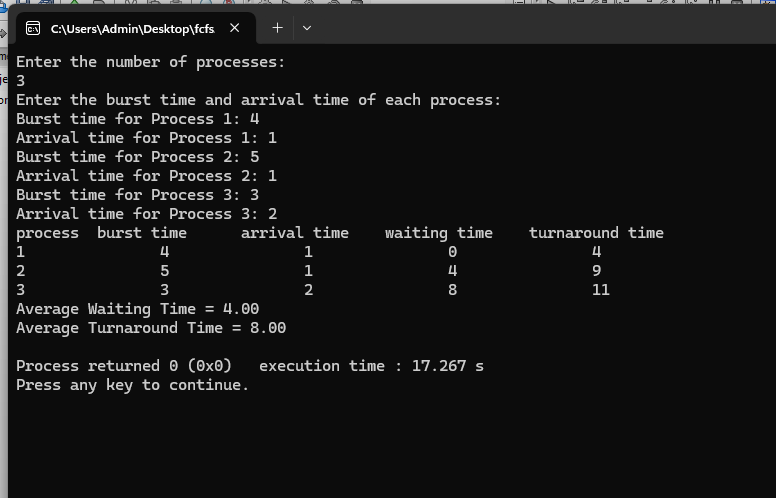
#include <stdio.h>  
void findwaitingtime(int bt[], int at[], int n, int wt[]) {  
    wt[0] = 0;  
    for (int i = 1; i < n; i++) {  
        wt[i] = bt[i - 1] + wt[i - 1] - at[i] + at[i - 1];  
    }  
}

void findturnaroundtime(int bt[], int n, int wt[], int tat[]) {  
    tat[0] = wt[0];  
    for (int i = 1; i < n; i++) {  
        tat[i] = bt[i] + wt[i];  
    }  
}

void findaveragetime(int bt[], int at[], int n) {  
    int wt[n], tat[n];  
    findwaitingtime(bt, at, n, wt);  
    findturnaroundtime(bt, n, wt, tat);  
    int totalwt = 0, totaltat = 0;  
    printf("Process\t burst time\t arrival time\t waiting time\t turnaround time\n");  
    for (int i = 0; i < n; i++) {  
        totalwt += wt[i];  
        totaltat += tat[i];  
        printf("%d\t\t%d\t\t%d\t\t%d\t\t%d\n", i + 1, bt[i], at[i], wt[i], tat[i]);  
    }  
    printf("Average Waiting Time = %.2f\n", (float)totalwt / n);  
    printf("Average Turnaround Time = %.2f\n", (float)totaltat / n);  
}

int main() {  
    int n;  
    printf("Enter the number of processes:\n");  
    scanf("%d", &n);  
    int bt[n], at[n];  
    printf("Enter the burst time and arrival time of each process:\n");  
    for (int i = 0; i < n; i++) {  
        printf("Burst time for Process %d: ", i + 1);  
        scanf("%d", &bt[i]);  
        printf("Arrival time for Process %d: ", i + 1);  
        scanf("%d", &at[i]);  
    }  
    findaveragetime(bt, at, n);  
    return 0;  
}

**Output -**



**Lab program 2**

**Write a C program to simulate CPU scheduling algorithm with the following:  
 a) SJF**

#include <stdio.h>  
#include <limits.h>

struct Process {  
    int pid;  
    int arrival\_time;  
    int burst\_time;  
    int remaining\_time;  
    int waiting\_time;  
    int turnaround\_time;  
    int completed;  
};

void sortByBurstTime(struct Process p[], int n) {  
    for (int i = 0; i < n - 1; i++) {  
        for (int j = 0; j < n - i - 1; j++) {  
            if (p[j].burst\_time > p[j + 1].burst\_time) {  
                struct Process temp = p[j];  
                p[j] = p[j + 1];  
                p[j + 1] = temp;  
            }  
        }  
    }  
}

void sjf\_non\_preemptive(struct Process p[], int n) {  
    sortByBurstTime(p, n);  
    p[0].waiting\_time = 0;  
    for (int i = 1; i < n; i++) {  
        p[i].waiting\_time = p[i - 1].waiting\_time + p[i - 1].burst\_time;  
    }  
    for (int i = 0; i < n; i++) {  
        p[i].turnaround\_time = p[i].waiting\_time + p[i].burst\_time;  
    }  
}

void sjf\_preemptive(struct Process p[], int n) {  
    int completed = 0, time = 0, min\_index;  
    while (completed != n) {  
        min\_index = -1;  
        int min\_time = INT\_MAX;

        for (int i = 0; i < n; i++) {  
            if (p[i].arrival\_time <= time && p[i].remaining\_time > 0 && p[i].remaining\_time < min\_time) {  
                min\_time = p[i].remaining\_time;  
                min\_index = i;  
            }  
        }  
        if (min\_index == -1) {  
            time++;  
            continue;  
        }

        p[min\_index].remaining\_time--;  
        time++;  
        if (p[min\_index].remaining\_time == 0) {  
            p[min\_index].completed = 1;  
            completed++;  
            p[min\_index].turnaround\_time = time - p[min\_index].arrival\_time;  
            p[min\_index].waiting\_time = p[min\_index].turnaround\_time - p[min\_index].burst\_time;  
        }  
    }  
}

void displayProcesses(struct Process p[], int n) {  
    float total\_wt = 0, total\_tat = 0;  
    printf("Process\tArrival Time\tBurst Time\tWaiting Time\tTurnaround Time\n");  
    for (int i = 0; i < n; i++) {  
        printf("P%d\t\t%d\t\t%d\t\t%d\t\t%d\n", p[i].pid, p[i].arrival\_time, p[i].burst\_time, p[i].waiting\_time, p[i].turnaround\_time);  
        total\_wt += p[i].waiting\_time;  
        total\_tat += p[i].turnaround\_time;  
    }  
    printf("\nAverage Waiting Time: %.2f\n", total\_wt / n);  
    printf("Average Turnaround Time: %.2f\n", total\_tat / n);  
}

int main() {  
    int n, choice;  
    printf("Enter number of processes: ");  
    scanf("%d", &n);  
    struct Process p[n];  
    for (int i = 0; i < n; i++) {  
        p[i].pid = i + 1;  
        printf("Enter Arrival Time and Burst Time for Process %d: ", i + 1);  
        scanf("%d %d", &p[i].arrival\_time, &p[i].burst\_time);  
        p[i].remaining\_time = p[i].burst\_time;  
        p[i].completed = 0;  
    }

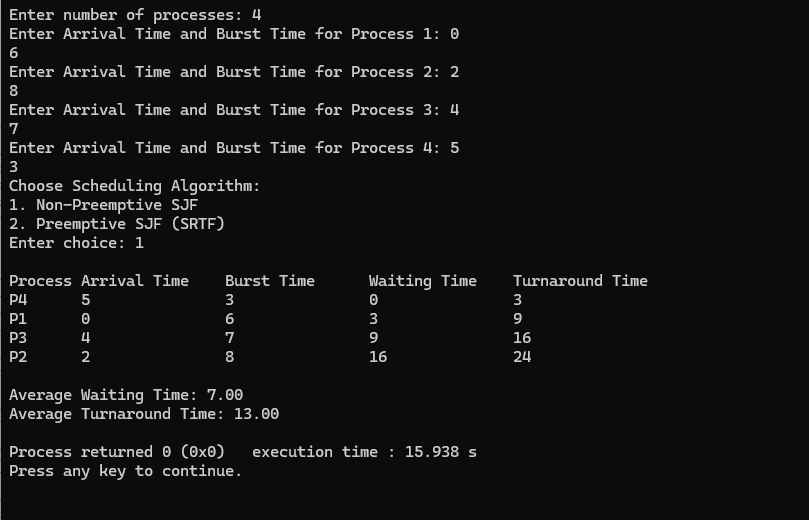
    printf("Choose Scheduling Algorithm:\n1. Non-Preemptive SJF\n2. Preemptive SJF (SRTF)\nEnter choice: ");  
    scanf("%d", &choice);  
    if (choice == 1)

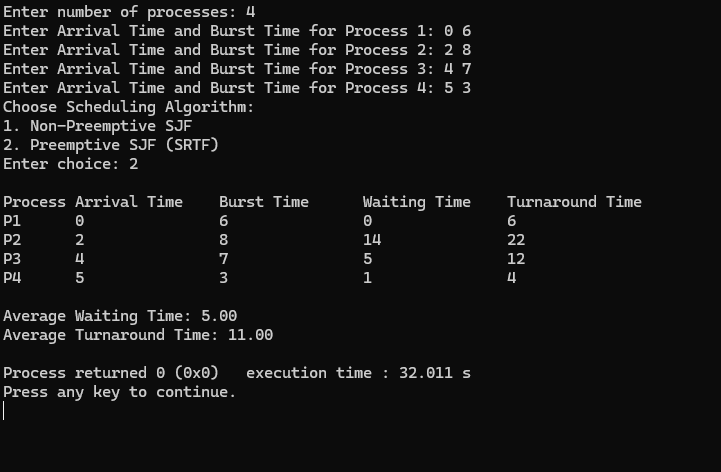
{  
        sjf\_non\_preemptive(p, n);  
    } else if (choice == 2) {  
        sjf\_preemptive(p, n);  
    }

else

{  
        printf("Invalid choice!\n");  
        return 1;  
    }  
    displayProcesses(p, n);  
    return 0;  
}

**Output:**





**Lab program 3**

**Write a C program to simulate CPU scheduling algorithm with the following:**

**a) priority**

**b) round robin**

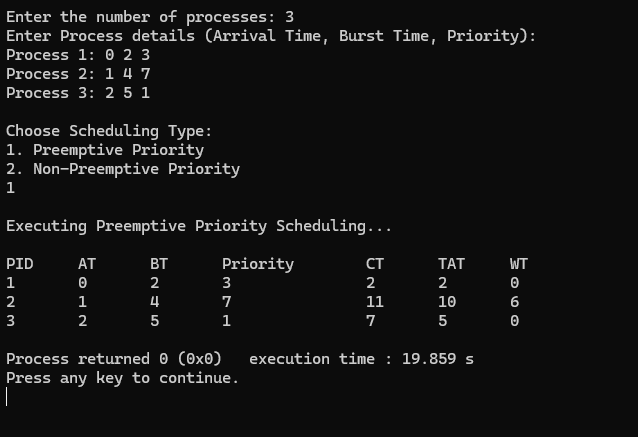
#include <stdio.h>  
#define MAX 100  
typedef struct {  
    int pid;  
    int arrival;  
    int burst;  
    int priority;  
    int remaining;  
    int completion;  
    int waiting;  
    int turnaround;  
} Process;  
  
void calculateTimes(Process processes[], int n, int preemptive) {  
    int time = 0, completed = 0, min\_index;  
    int is\_completed[MAX] = {0};  
  
    while (completed != n) {  
        min\_index = -1;  
  
        for (int i = 0; i < n; i++) {  
            if (processes[i].arrival <= time && !is\_completed[i]) {  
                if (min\_index == -1 ||  
                    processes[i].priority < processes[min\_index].priority ||  
                    (processes[i].priority == processes[min\_index].priority && processes[i].arrival < processes[min\_index].arrival)) {  
                    min\_index = i;  
                }  
            }  
        }  
  
        if (min\_index == -1) {  
            time++;  
        } else {  
            if (preemptive) {  
                processes[min\_index].remaining--;  
                if (processes[min\_index].remaining == 0) {  
                    processes[min\_index].completion = time + 1;  
                    is\_completed[min\_index] = 1;  
                    completed++;  
                }  
            } else {  
                time += processes[min\_index].burst;  
                processes[min\_index].completion = time;  
                is\_completed[min\_index] = 1;  
                completed++;  
            }  
  
            if (!preemptive)  
                processes[min\_index].remaining = 0;  
            else  
                time++;

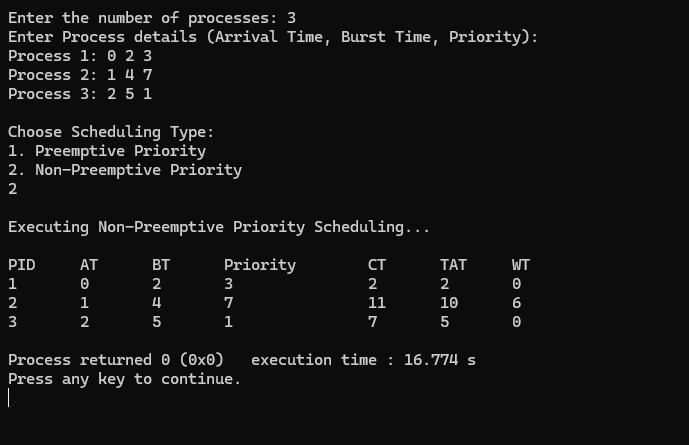
    }  
    }  
  
    for (int i = 0; i < n; i++) {  
        processes[i].turnaround = processes[i].completion - processes[i].arrival;  
        processes[i].waiting = processes[i].turnaround - processes[i].burst;  
    }  
}  
  
void printResults(Process processes[], int n) {  
    printf("\nPID\tAT\tBT\tPriority\tCT\tTAT\tWT\n");

    for (int i = 0; i < n; i++) {  
        printf("%d\t%d\t%d\t%d\t\t%d\t%d\t%d\n",  
               processes[i].pid, processes[i].arrival, processes[i].burst,  
               processes[i].priority, processes[i].completion,  
               processes[i].turnaround, processes[i].waiting);  
    }  
}  
  
int main() {  
    int n, choice;  
    Process processes[MAX];  
  
    printf("Enter the number of processes: ");  
    scanf("%d", &n);printf("Enter Process details (Arrival Time, Burst Time, Priority):\n");  
for (int i = 0; i < n; i++) {  
    processes[i].pid = i + 1;  
    printf("Process %d: ", i + 1);  
    scanf("%d %d %d", &processes[i].arrival, &processes[i].burst, &processes[i].priority);  
    processes[i].remaining = processes[i].burst;  
}  
  
printf("\nChoose Scheduling Type:\n1. Preemptive Priority\n2. Non-Preemptive Priority\n");  
scanf("%d", &choice);  
  
if (choice == 1) {  
    printf("\nExecuting Preemptive Priority Scheduling...\n");  
    calculateTimes(processes, n, 1);  
} else {  
    printf("\nExecuting Non-Preemptive Priority Scheduling...\n");  
    calculateTimes(processes, n, 0);  
}  
  
printResults(processes, n);  
  
return 0;

}

**Output:**





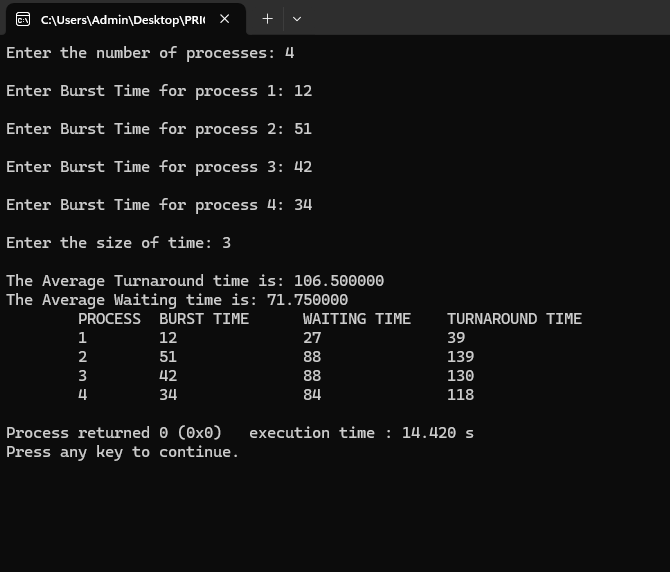
**c) ROUND ROBIN**

#include <stdio.h>  
  
int main() {  
    int i, j, n, bu[10], wa[10], tat[10], t, ct[10], max;  
    float awt = 0, att = 0, temp = 0;  
  
    printf("Enter the number of processes: ");  
    scanf("%d", &n);  
  
    for (i = 0; i < n; i++) {  
        printf("\nEnter Burst Time for process %d: ", i + 1);  
        scanf("%d", &bu[i]);  
        ct[i] = bu[i];  
    }  
  
    printf("\nEnter the size of time: ");  
    scanf("%d", &t);  
  
    max = bu[0];  
    for (i = 1; i < n; i++) {  
        if (max < bu[i])  
            max = bu[i];  
    }  
  
    for (j = 0; j <= (max + t - 1) / t; j++) {  
        for (i = 0; i < n; i++) {  
            if (bu[i] != 0) {  
                if (bu[i] < t) {  
                    tat[i] = temp + bu[i];  
                    temp += bu[i];  
                    bu[i] = 0;  
                } else {  
                    bu[i] -= t;  
                    temp += t;  
                }  
            }  
        }  
    }  
  
    for (i = 0; i < n; i++) {  
        wa[i] = tat[i] - ct[i];  
        att += tat[i];  
        awt += wa[i];  
    }  
  
    printf("\nThe Average Turnaround time is: %f", att / n);  
    printf("\nThe Average Waiting time is: %f", awt / n);  
    printf("\n\nPROCESS\t BURST TIME \t WAITING TIME \t TURNAROUND TIME\n");

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

printf("\t%d \t %d \t\t%d \t\t %d\n", i + 1, ct[i], wa[i], tat[i]);  
    }  
  
    return 0;  
}

**Output:**



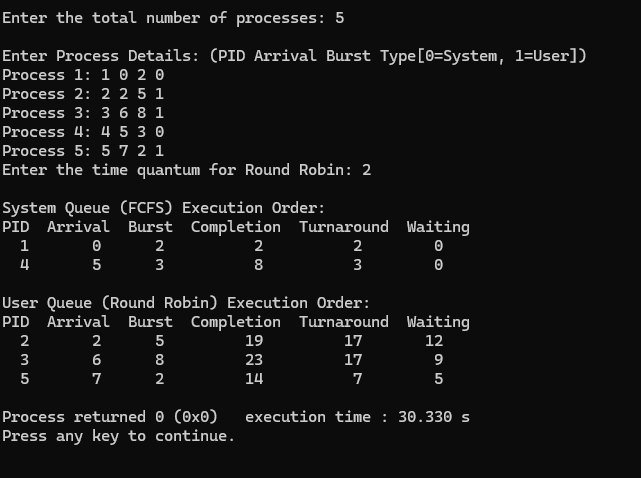
**Lab program 4**

**Write a C program to simulate multilevel queue scheduling**

#include <stdio.h>  
#define MAX 100  
  
typedef struct {  
    int pid, arrival, burst, completion, turnaround, waiting, remaining;  
} Process;  
  
void inputProcesses(Process sys[], int \*sysCount, Process user[], int \*userCount);  
void fcfsScheduling(Process p[], int n, int startTime);  
void roundRobinScheduling(Process p[], int n, int startTime, int timeQuantum);  
void displayProcesses(Process p[], int n, const char \*queueName);  
  
int main() {  
    Process systemQueue[MAX], userQueue[MAX];  
    int sysCount = 0, userCount = 0, timeQuantum;  
  
    inputProcesses(systemQueue, &sysCount, userQueue, &userCount);  
  
    printf("Enter the time quantum for Round Robin: ");  
    scanf("%d", &timeQuantum);  
  
    fcfsScheduling(systemQueue, sysCount, 0);  
    roundRobinScheduling(userQueue, userCount, systemQueue[sysCount - 1].completion, timeQuantum);  
  
    displayProcesses(systemQueue, sysCount, "System Queue (FCFS)");  
    displayProcesses(userQueue, userCount, "User Queue (Round Robin)");  
  
    return 0;  
}  
  
void inputProcesses(Process sys[], int \*sysCount, Process user[], int \*userCount) {  
    int n, type;  
    printf("Enter the total number of processes: ");  
    scanf("%d", &n);  
  
    printf("\nEnter Process Details: (PID Arrival Burst Type[0=System, 1=User])\n");  
    for (int i = 0; i < n; i++) {  
        Process temp;  
        printf("Process %d: ", i + 1);  
        scanf("%d %d %d %d", &temp.pid, &temp.arrival, &temp.burst, &type);  
        temp.remaining = temp.burst;  
  
        if (type == 0) {  
            sys[\*sysCount] = temp;  
            (\*sysCount)++;  
        } else {  
            user[\*userCount] = temp;  
            (\*userCount)++;  
        }  
    }  
}  
  
void fcfsScheduling(Process p[], int n, int startTime) {  
    if (n == 0) return;  
  
    int time = startTime;  
    for (int i = 0; i < n; i++) {  
        if (time < p[i].arrival) {  
            time = p[i].arrival;  
        }  
        p[i].completion = time + p[i].burst;  
        p[i].turnaround = p[i].completion - p[i].arrival;  
        p[i].waiting = p[i].turnaround - p[i].burst;  
        time = p[i].completion;  
    }  
}  
  
void roundRobinScheduling(Process p[], int n, int startTime, int timeQuantum) {  
    if (n == 0) return;  
  
    int time = startTime, remainingProcesses = n;  
    while (remainingProcesses > 0) {  
        int allIdle = 1;  
        for (int i = 0; i < n; i++) {  
            if (p[i].remaining > 0 && p[i].arrival <= time) {  
                allIdle = 0;  
                if (p[i].remaining > timeQuantum) {  
                    time += timeQuantum;  
                    p[i].remaining -= timeQuantum;  
                } else {  
                    time += p[i].remaining;  
                    p[i].completion = time;  
                    p[i].turnaround = p[i].completion - p[i].arrival;  
                    p[i].waiting = p[i].turnaround - p[i].burst;  
                    p[i].remaining = 0;  
                    remainingProcesses--;  
                }  
            }  
        }  
        if (allIdle) time++;

    }  
}  
  
void displayProcesses(Process p[], int n, const char \*queueName) {  
    if (n == 0) {  
        printf("\nNo processes in %s.\n", queueName);  
        return;  
    }  
  
    printf("\n%s Execution Order:\n", queueName);  
    printf("PID  Arrival  Burst  Completion  Turnaround  Waiting\n");  
    for (int i = 0; i < n; i++) {  
        printf("%3d %7d %6d %10d %10d %8d\n",  
               p[i].pid, p[i].arrival, p[i].burst,  
               p[i].completion, p[i].turnaround, p[i].waiting);  
    }  
}

**OUTPUT:**



**Lab program 5**

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

**(a) Rate- Monotonic**

**(b) Earliest-deadline First.**

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#include <stdbool.h>

#define MAX\_PROCESS 10

typedef struct {

int id;

int burst\_time;

float priority;

} Task;

int num\_of\_process;

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

void get\_process\_info(int selected\_algo) {

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

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

if (num\_of\_process < 1) {

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]);

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

if (selected\_algo == 2) {

printf("==> Deadline: ");

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

} else {

printf("==> Period: ");

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

}

}

}

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

int max;

if (a >= b && a >= c)

max = a;

else if (b >= a && b >= c)

max = b;

else

max = c;

return max;

}

int get\_observation\_time(int selected\_algo)

{

if (selected\_algo == 1) {

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

} else if (selected\_algo == 2) {

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

}

return 0;

}

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

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

printf("Time: ");

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

if (i < 10)

printf("| 0%d ", i);

else

printf("| %d ", 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}, min = 999, next\_process = 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;

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

if (utilization > m) {

printf("\nGiven problem is not schedulable under the said scheduling algorithm.\n");

return;

}

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

min = 1000;

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

if (remain\_time[j] > 0) {

if (min > period[j]) {

min = period[j];

next\_process = j;

}

}

}

if (remain\_time[next\_process] > 0) {

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

remain\_time[next\_process] -= 1;

}

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

if ((i + 1) % period[k] == 0) {

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

}

}

}

print\_schedule(process\_list, time);

}

void earliest\_deadline\_first(int time) {

float utilization = 0;

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

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

}

int n = num\_of\_process;

int process[num\_of\_process];

int max\_deadline, current\_process = 0, min\_deadline, process\_list[time];

bool is\_ready[num\_of\_process];

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

is\_ready[i] = true;

process[i] = i + 1;

}

max\_deadline = deadline[0];

for (int i = 1; i < num\_of\_process; i++) {

if (deadline[i] > max\_deadline)

max\_deadline = deadline[i];

}

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

for (int j = i + 1; j < num\_of\_process; j++) {

if (deadline[j] < deadline[i]) {

int temp = execution\_time[j];

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

execution\_time[i] = temp;

temp = deadline[j];

deadline[j] = deadline[i];

deadline[i] = temp;

temp = process[j];

process[j] = process[i];

process[i] = temp;

}

}

}

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

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

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

}

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

if (current\_process != -1) {

--execution\_time[current\_process];

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

} else {

process\_list[t] = 0;

}

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

--deadline[i];

if ((execution\_time[i] == 0) && is\_ready[i]) {

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

is\_ready[i] = false;

}

if ((deadline[i] <= remain\_deadline[i]) && !is\_ready[i]) {

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

is\_ready[i] = true;

}

}

min\_deadline = max\_deadline;

current\_process = -1;

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

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

current\_process = i;

min\_deadline = deadline[i];

}

}

}

print\_schedule(process\_list, time);

}

int main()

{

int option;

int observation\_time;

while (1) {

printf("\n1. Rate Monotonic\n2. Earliest Deadline first\n3. Proportional Scheduling\n\nEnter your choice: ");

scanf("%d", &option);

switch (option) {

case 1:

get\_process\_info(option);

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

rate\_monotonic(observation\_time);

break;

case 2:

get\_process\_info(option);

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

earliest\_deadline\_first(observation\_time);

break;

case 3:

exit(0);

default:

printf("\nInvalid Statement");

}

}

return 0;

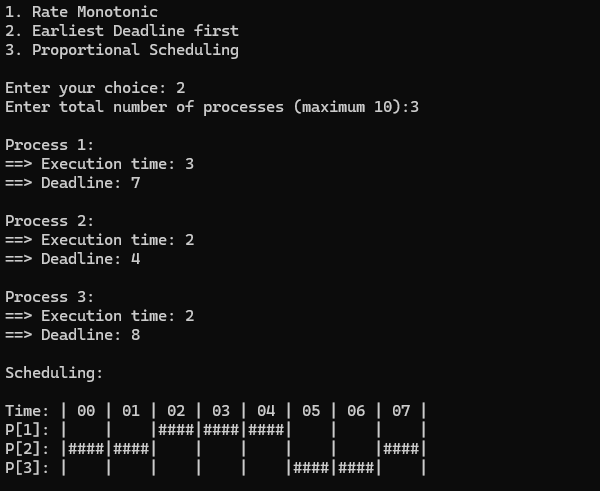
}

**OUTPUT:**

Rate- Monotonic



Earliest-deadline First



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

**problem.**

#include <stdio.h>

#include <pthread.h>

#include <semaphore.h>

#include <unistd.h> // Include for sleep function

#define N 5

#define THINKING 2

#define HUNGRY 1

#define EATING 0

#define LEFT (i + 4) % N

#define RIGHT (i + 1) % N

int state[N];

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

sem\_t mutex;

sem\_t S[N];

void test(int i) {

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

state[i] = EATING;

sleep(2);

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

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

sem\_post(&S[i]);

}

}

void take\_fork(int i) {

sem\_wait(&mutex);

state[i] = HUNGRY;

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

test(i);

sem\_post(&mutex);

sem\_wait(&S[i]);

sleep(1);

}

void put\_fork(int i) {

sem\_wait(&mutex);

state[i] = 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(LEFT);

test(RIGHT);

sem\_post(&mutex);

}

void\* philosopher(void\* num) {

while (1) {

int\* i = num;

sleep(1);

take\_fork(\*i);

sleep(0);

put\_fork(\*i);

}

}

int main() {

int i;

pthread\_t thread\_id[N];

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

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

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

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

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

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

}

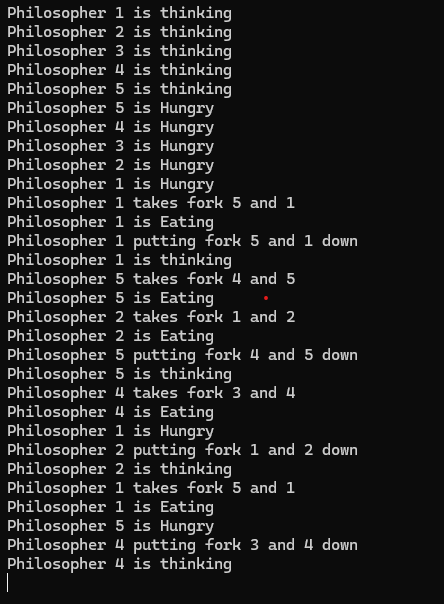
for (i = 0; i < N; i++) {

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

}

}

**OUTPUT:**



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

#include <stdio.h>

#include <stdlib.h>

int mutex = 1, full = 0, empty = 3, x = 0;

void producer();

void consumer();

int wait(int);

int signal(int);

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!!");

}

break;

case 2:

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

consumer();

} else {

printf("Buffer is empty!!");

}

break;

case 3:

exit(0);

break;

default:

printf("Invalid choice! Please enter 1, 2, or 3.");

break;

}

}

return 0;

}

int wait(int s) {

return --s;

}

int signal(int s) {

return ++s;

}

void producer() {

mutex = wait(mutex);

full = signal(full);

empty = wait(empty);

x++;

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

mutex = signal(mutex);

}

void consumer() {

mutex = wait(mutex);

full = wait(full);

empty = signal(empty);

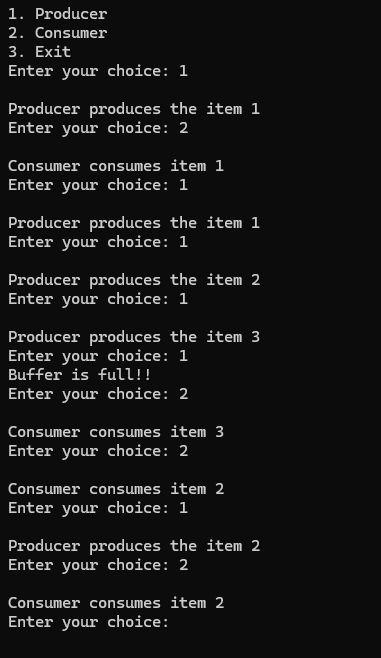
printf("\nConsumer consumes item %d", x);

x--;

mutex = signal(mutex);

}

OUTPUT:



**Lab program 6**

**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];

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

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

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

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

}

)

int max[n][m];

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

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

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

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

}

}

int available[m];

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

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

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

}

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

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

f[k] = 0;

}

int need[n][m];

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

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

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

}

}

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("The following system is not safe\n");

break;

}

}

if (flag == 1) {

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

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

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

}

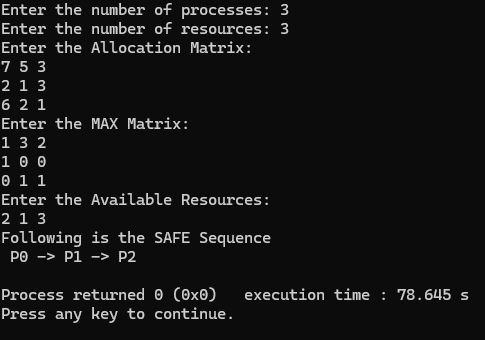
printf(" P%d\n", ans[n - 1]);

}

return 0;

}

**OUTPUT:**

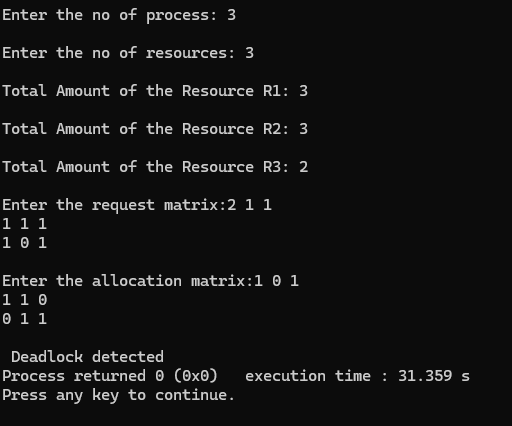


**Write a C program to simulate deadlock detection.**

#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:");  
    for(i = 0; i < np; i++)  
        for(j = 0; j < nr; j++)  
            scanf("%d", &request[i][j]);  
  printf("\nEnter the allocation matrix:");  
    for(i = 0; i < np; i++)  
        for(j = 0; j < nr; j++)  
            scanf("%d", &alloc[i][j]);  
      for(j = 0; j < nr; j++)  
    {  
        avail[j] = r[j];  
        for(i = 0; i < np; i++)  
        {  
            avail[j] -= alloc[i][j];  
        }  
    }   
    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;  
    }  
  for(j = 0; j < nr; j++)  
        w[j] = avail[j];   
    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];  
            }  
        }  
    }   
    int deadlock = 0;  
    for(i = 0; i < np; i++)  
        if(mark[i] != 1)  
            deadlock = 1;  
    if(deadlock)  
        printf("\n Deadlock detected");  
    else  
        printf("\n No Deadlock possible");  
}

**OUTPUT:**



**Lab program 7**

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

**allocation techniques:**

**(a) Worst-fit**

**(b) Best-fit**

**(c) 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++) {

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

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

}

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

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

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}, ff[max], frag[max];

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

ff[i] = -1;

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

if (!bf[j] && b[j] >= f[i]) {

ff[i] = j;

frag[i] = b[j] - f[i];

bf[j] = 1;

break;

}

}

}

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

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

if (ff[i] != -1)

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

}

}

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

int tempBlocks[max];

for (int i = 0; i < nb; i++) tempBlocks[i] = b[i]

int bf[max] = {0}, ff[max], frag[max];

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

int index = -1, maxDiff = -1;

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

if (!bf[j] && tempBlocks[j] >= f[i] && (tempBlocks[j] - f[i] > maxDiff)) {

index = j;

maxDiff = tempBlocks[j] - f[i];

}

}

if (index != -1) {

ff[i] = index;

frag[i] = tempBlocks[index] - f[i];

bf[index] = 1;

} else {

ff[i] = -1;

}

}

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

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

if (ff[i] != -1)

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

}

}

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

int tempBlocks[max];

for (int i = 0; i < nb; i++) tempBlocks[i] = b[i];

int bf[max] = {0}, ff[max], frag[max];

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

int index = -1, minDiff = 100000;

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

if (!bf[j] && tempBlocks[j] >= f[i] && (tempBlocks[j] - f[i] < minDiff)) {

index = j;

minDiff = tempBlocks[j] - f[i];

}

}

if (index != -1) {

ff[i] = index;

frag[i] = tempBlocks[index] - f[i];

bf[index] = 1;

} else {

ff[i] = -1;

}

}

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

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

if (ff[i] != -1)

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]);

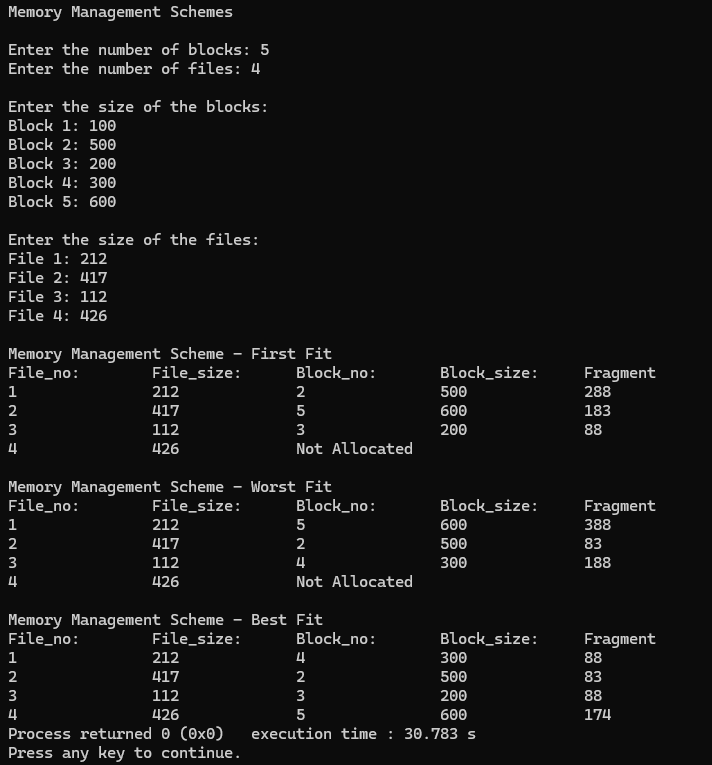
else

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

}

}

**OUTPUT:**



**Lab program 8**

**Write a C program to simulate page replacement algorithms:**

**(a) FIFO**

**(b) LRU**

**(c) Optimal**

#include <stdio.h>

int n, f, i, j, k;

int in[100];

int p[ 50];

int hit = 0;

int pgfaultcnt = 0;

void getData() {

printf("\nEnter length of page reference sequence: ");

scanf("%d", &n);

printf("Enter the page reference sequence: ");

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

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

printf("Enter number of frames: ");

scanf("%d", &f);

}

void initialize() {

pgfaultcnt = 0;

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

p[i] = -1;

}

int isHit(int data) {

hit = 0;

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

if (p[j] == data) {

hit = 1;

break;

}

}

return hit;

}

int getHitIndex(int data) {

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

if (p[k] == data)

return k;

}

return -1;

}

void dispPages() {

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

if (p[k] != -1)

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

}

printf("\n");

}

void dispPgFaultCnt() {

printf("\nTotal number of page faults: %d\n", pgfaultcnt);

}

void fifo() {

initialize();

int index = 0;

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

printf("For %d: ", in[i]);

if (!isHit(in[i])) {

p[index] = in[i];

index = (index + 1) % f;

pgfaultcnt++;

dispPages();

} else {

printf("No page fault\n");

}

}

dispPgFaultCnt();

}

void optimal() {

initialize();

int near[50];

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

printf("For %d: ", in[i]);

if (!isHit(in[i])) {

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

int pg = p[j];

int found = 0;

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

if (pg == in[k]) {

near[j] = k;

found = 1;

break;

}

}

if (!found)

near[j] = 9999;

}

int max = -1, repindex = -1;

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

if (near[j] > max) {

max = near[j];

repindex = j;

}

}

p[repindex] = in[i];

pgfaultcnt++;

dispPages();

} else {

printf("No page fault\n");

}

}

dispPgFaultCnt();

}

void lru() {

initialize();

int least[50];

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

printf("For %d: ", in[i]);

if (!isHit(in[i])) {

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

int pg = p[j];

int found = 0;

for (k = i - 1; k >= 0; k--) {

if (pg == in[k]) {

least[j] = k;

found = 1;

break;

}

}

if (!found)

least[j] = -9999;

}

int min = 9999, repindex = -1;

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

if (least[j] < min) {

min = least[j];

repindex = j;

}

}

p[repindex] = in[i];

pgfaultcnt++;

dispPages();

} else {

printf("No page fault!\n");

}

}

dispPgFaultCnt();

}

int main() {

int choice;

while (1) {

printf("\nPage Replacement Algorithms\n");

printf("1. Enter data\n2. FIFO\n3. Optimal\n4. LRU\n5. Exit\nEnter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1: getData(); break;

case 2: fifo(); break;

case 3: optimal(); break;

case 4: lru(); break;

case 5: return 0;

default: printf("Invalid choice!\n"); break;

}

}

}

**OUTPUT:**

