VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.



LAB REPORT

on

OPERATING SYSTEMS

Submitted by

VINAY G M(1WA23CS044)

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

Feb-2025 to June-2025

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 VINAY G M1WA23CS044), 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 Feb 2025- June 2025. 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.

|  |  |
| --- | --- |
|  |  |

DR.SEEMA PATIL Dr. Kavitha Sooda

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) | 1-10 |
| 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) | 11-19 |
| 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. | 20-24 |
| 4. | Write a C program to simulate Real-Time CPU Scheduling algorithms:   1. Rate- Monotonic 2. Earliest-deadline First | 24-34 |
| 5. | Write a C program to simulate producer-consumer problem using semaphores | 35-38 |
| 6. | Write a C program to simulate the concept of Dining Philosophers problem. | 39-43 |
| 7. | Write a C program to simulate Bankers algorithm for the purpose of deadlock avoidance. | 44-47 |
| 8. | Write a C program to simulate deadlock detection | 48-51 |
| 9. | Write a C program to simulate the following contiguous memory allocation techniques a) Worst-fit   1. Best-fit 2. First-fit | 52-56 |
| 10. | Write a C program to simulate page replacement algorithms a) FIFO   1. LRU 2. Optimal | 57-63 |

Course Outcomes

|  |  |
| --- | --- |
| C01 | Apply the different concepts and functionalities of Operating System |
| C02 | Analyse various Operating system strategies and techniques |
| C03 | Demonstrate the different functionalities of Operating System. |
| C04 | Conduct practical experiments to implement the functionalities of Operating system. |

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)

CODE:

#include <stdio.h>

#include <stdlib.h>

#define MAX 10

#define INF 9999

struct process {

int id, AT, BT, CT, TAT, WT, RT, remaining\_BT;

int completed;

};

void sort\_by\_AT(struct process p[], int n) {

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

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

if (p[i].AT > p[j].AT) {

struct process temp = p[i];

p[i] = p[j];

p[j] = temp;

}

}

}

}

void FCFS(struct process p[], int n) {

sort\_by\_AT(p, n);

int currentTime = 0;

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

if (currentTime < p[i].AT)

currentTime = p[i].AT;

p[i].RT = currentTime - p[i].AT;

p[i].CT = currentTime + p[i].BT;

currentTime = p[i].CT;

p[i].TAT = p[i].CT - p[i].AT;

p[i].WT = p[i].TAT - p[i].BT;

}

}

void SJF\_NonPreemptive(struct process p[], int n) {

int completed = 0, currentTime = 0;

while (completed < n) {

int shortest = -1, minBT = INF;

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

if (!p[i].completed && p[i].AT <= currentTime && p[i].BT < minBT) {

minBT = p[i].BT;

shortest = i;

}

}

if (shortest == -1) {

currentTime++;

} else {

p[shortest].RT = currentTime - p[shortest].AT;

p[shortest].CT = currentTime + p[shortest].BT;

currentTime = p[shortest].CT;

p[shortest].TAT = p[shortest].CT - p[shortest].AT;

p[shortest].WT = p[shortest].TAT - p[shortest].BT;

p[shortest].completed = 1;

completed++;

}

}

}

void SJF\_Preemptive(struct process p[], int n) {

int completed = 0, currentTime = 0;

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

p[i].remaining\_BT = p[i].BT;

}

while (completed < n) {

int shortest = -1, minBT = INF;

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

if (!p[i].completed && p[i].AT <= currentTime && p[i].remaining\_BT < minBT) {

minBT = p[i].remaining\_BT;

shortest = i;

}

}

if (shortest == -1) {

currentTime++;

} else {

if (p[shortest].remaining\_BT == p[shortest].BT)

p[shortest].RT = currentTime - p[shortest].AT;

p[shortest].remaining\_BT--;

currentTime++;

if (p[shortest].remaining\_BT == 0) {

p[shortest].CT = currentTime;

p[shortest].TAT = p[shortest].CT - p[shortest].AT;

p[shortest].WT = p[shortest].TAT - p[shortest].BT;

p[shortest].completed = 1;

completed++;

}

}

}

}

void display(struct process p[], int n) {

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

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

printf("%d\t%d\t%d\t%d\t%d\t%d\t%d\n", p[i].id, p[i].AT, p[i].BT, p[i].CT, p[i].TAT, p[i].WT, p[i].RT);

}

}

int main() {

int n, choice;

struct process p[MAX];

printf("Enter number of processes: ");

scanf("%d", &n);

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

p[i].id = i + 1;

printf("Enter AT for process %d: ", i + 1);

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

printf("Enter BT for process %d: ", i + 1);

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

p[i].completed = 0;

}

while (1) {

printf("\nMenu:\n");

printf("1. First Come First Serve (FCFS)\n");

printf("2. (SJF)- Non Preemptive\n");

printf("3. (SJF)- Preemptive\n");

printf("4. Exit\n");

printf("Enter choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

FCFS(p, n);

display(p, n);

break;

case 2:

SJF\_NonPreemptive(p, n);

display(p, n);

break;

case 3:

SJF\_Preemptive(p, n);

display(p, n);

break;

case 4:

exit(0);

default:

printf("Invalid choice. Try again.\n");

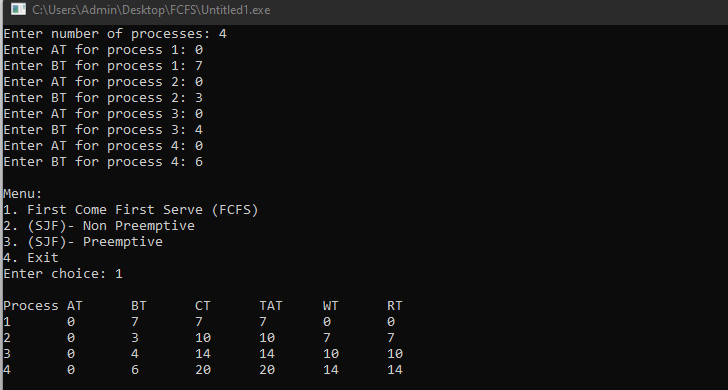
}

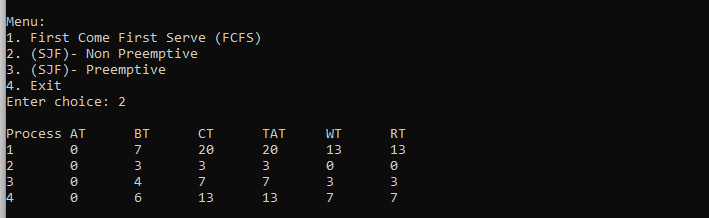
}

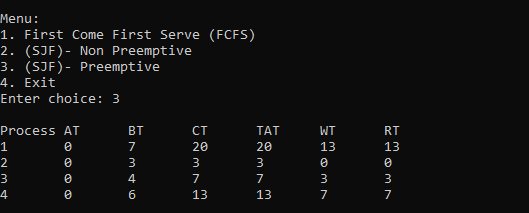
return 0;

}

OUTPUT:







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)

CODE:

#include <stdio.h>

#define MAX 10

typedef struct {

int pid, at, bt, pt, remaining\_bt, ct, tat, wt, rt, is\_completed, st;

} Process;

void nonPreemptivePriority(Process p[], int n) {

int time = 0, completed = 0;

while (completed < n) {

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

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

if (p[i].at <= time && !p[i].is\_completed && p[i].pt < highest\_priority) {

highest\_priority = p[i].pt;

selected = i;

}

}

if (selected == -1) {

time++;

continue;

}

if (p[selected].rt == -1) {

p[selected].st = time;

p[selected].rt = time - p[selected].at;

}

time += p[selected].bt;

p[selected].ct = time;

p[selected].tat = p[selected].ct - p[selected].at;

p[selected].wt = p[selected].tat - p[selected].bt;

p[selected].is\_completed = 1;

completed++;

}

}

void preemptivePriority(Process p[], int n) {

int time = 0, completed = 0;

while (completed < n) {

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

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

if (p[i].at <= time && p[i].remaining\_bt > 0 && p[i].pt < highest\_priority) {

highest\_priority = p[i].pt;

selected = i;

}

}

if (selected == -1) {

time++;

continue;

}

if (p[selected].rt == -1) {

p[selected].st = time;

p[selected].rt = time - p[selected].at;

}

p[selected].remaining\_bt--;

time++;

if (p[selected].remaining\_bt == 0) {

p[selected].ct = time;

p[selected].tat = p[selected].ct - p[selected].at;

p[selected].wt = p[selected].tat - p[selected].bt;

completed++;

}

}

}

void displayProcesses(Process p[], int n) {

float avg\_tat = 0, avg\_wt = 0, avg\_rt = 0;

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

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

printf("%d\t%d\t%d\t%d\t\t%d\t%d\t%d\t%d\n",

p[i].pid, p[i].at, p[i].bt, p[i].pt, p[i].ct, p[i].tat, p[i].wt, p[i].rt);

avg\_tat += p[i].tat;

avg\_wt += p[i].wt;

avg\_rt += p[i].rt;

}

printf("\nAverage TAT: %.2f", avg\_tat / n);

printf("\nAverage WT: %.2f", avg\_wt / n);

printf("\nAverage RT: %.2f\n", avg\_rt / n);

}

int main() {

Process p[MAX];

int n, choice;

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

scanf("%d", &n);

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

p[i].pid = i + 1;

printf("\nEnter Arrival Time, Burst Time, and Priority for Process %d:\n", p[i].pid);

printf("Arrival Time: ");

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

printf("Burst Time: ");

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

printf("Priority : ");

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

p[i].remaining\_bt = p[i].bt;

p[i].is\_completed = 0;

p[i].rt = -1;

}

while (1) {

printf("\nPriority Scheduling Menu:\n");

printf("1. Non-Preemptive Priority Scheduling\n");

printf("2. Preemptive Priority Scheduling\n");

printf("3. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

nonPreemptivePriority(p, n);

printf("Non-Preemptive Scheduling Completed!\n");

displayProcesses(p, n);

break;

case 2:

preemptivePriority(p, n);

printf("Preemptive Scheduling Completed!\n");

displayProcesses(p, n);

break;

case 3:

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

return 0;

default:

printf("Invalid choice! Try again.\n");

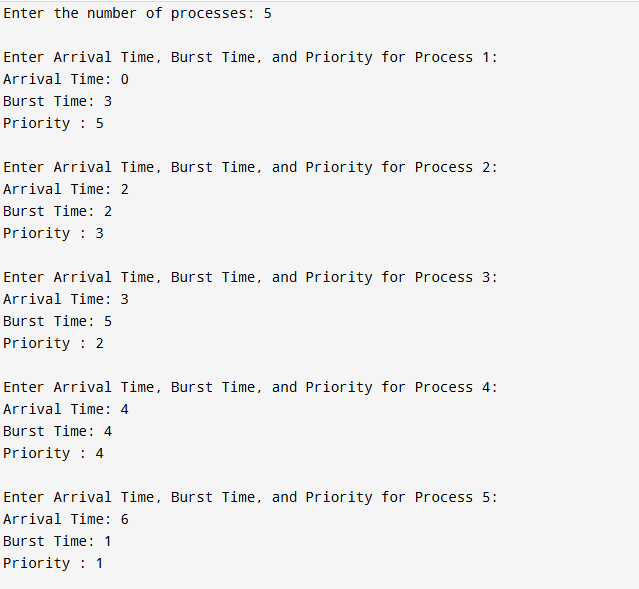
}

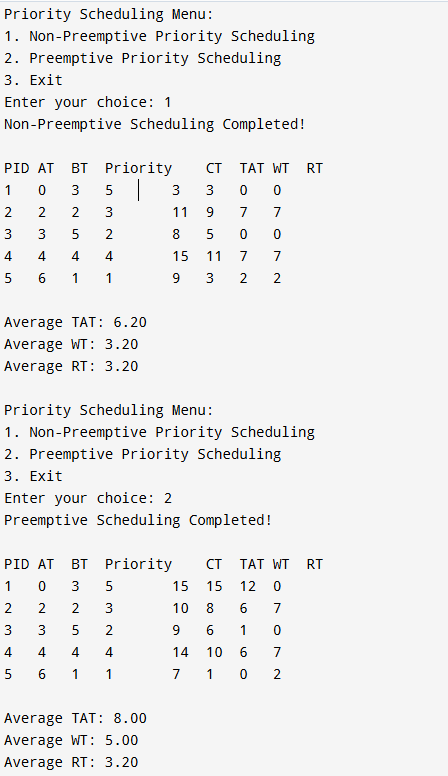
}

return 0;

}

OUTPUT:





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.

CODE:

#include <stdio.h>

struct Process {

int id, burst\_time, arrival\_time, queue;

int waiting\_time, turnaround\_time, response\_time;

};

void round\_robin(struct Process p[], int n, int quantum) {

int remaining\_time[n], completed = 0, time = 0;

for (int i = 0; i < n; i++) remaining\_time[i] = p[i].burst\_time;

while (completed < n) {

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

if (remaining\_time[i] > 0) {

if (remaining\_time[i] > quantum) {

time += quantum;

remaining\_time[i] -= quantum;

} else {

time += remaining\_time[i];

p[i].waiting\_time = time - p[i].arrival\_time - p[i].burst\_time;

p[i].turnaround\_time = time - p[i].arrival\_time;

p[i].response\_time = p[i].waiting\_time;

remaining\_time[i] = 0;

completed++;

}

}

}

}

}

void fcfs(struct Process p[], int n, int start\_time) {

int time = start\_time;

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

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

time = p[i].arrival\_time;

p[i].waiting\_time = time - p[i].arrival\_time;

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

p[i].response\_time = p[i].waiting\_time;

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

}

}

int main() {

int n;

printf("Enter number of processes: ");

scanf("%d", &n);

struct Process processes[n], system\_queue[n], user\_queue[n];

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

printf("Enter Burst Time, Arrival Time and Queue of each process: \n");

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

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

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

processes[i].id = i + 1;

if (processes[i].queue == 1)

system\_queue[sys\_count++] = processes[i];

else if (processes[i].queue == 2)

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

}

int quantum = 2;

round\_robin(system\_queue, sys\_count, quantum);

int last\_exec\_time = (sys\_count > 0) ? system\_queue[sys\_count - 1].turnaround\_time : 0;

fcfs(user\_queue, user\_count, last\_exec\_time);

printf("\nProcess\tWT\tTAT\tRt\n");

for (int i = 0; i < sys\_count; i++)

printf("P%d\t%d\t%d\t%d\n", system\_queue[i].id, system\_queue[i].waiting\_time, system\_queue[i].turnaround\_time, system\_queue[i].response\_time);

for (int i = 0; i < user\_count; i++)

printf("P%d\t%d\t%d\t%d\n", user\_queue[i].id, user\_queue[i].waiting\_time, user\_queue[i].turnaround\_time, user\_queue[i].response\_time);

float avg\_wait = 0, avg\_tat = 0, avg\_resp = 0;

for (int i = 0; i < sys\_count; i++) {

avg\_wait += system\_queue[i].waiting\_time;

avg\_tat += system\_queue[i].turnaround\_time;

avg\_resp += system\_queue[i].response\_time;

}

for (int i = 0; i < user\_count; i++) {

avg\_wait += user\_queue[i].waiting\_time;

avg\_tat += user\_queue[i].turnaround\_time;

avg\_resp += user\_queue[i].response\_time;

}

int total = sys\_count + user\_count;

printf("\nAverage Waiting Time: %.2f", avg\_wait / total);

printf("\nAverage Turn Around Time: %.2f", avg\_tat / total);

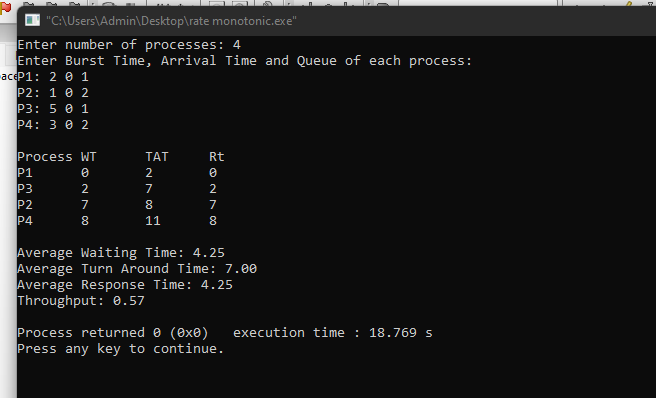
printf("\nAverage Response Time: %.2f", avg\_resp / total);

printf("\nThroughput: %.2f\n", (float)total / avg\_tat \* total);

return 0;

}

Output:



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

1. Rate- Monotonic
2. Earliest-deadline First

CODE:

#include <stdio.h>

#define MAX\_PROCESSES 10

typedef struct {

int id;

int burst\_time;

int period;

int remaining\_time;

int next\_deadline;

} Process;

void sort\_by\_period(Process processes[], int n) {

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

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

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

Process temp = processes[j];

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

processes[j + 1] = temp;

}

}

}

}

int gcd(int a, int b) {

return b == 0 ? a : gcd(b, a % b);

}

int lcm(int a, int b) {

return (a \* b) / gcd(a, b);

}

int calculate\_lcm(Process processes[], int n) {

int result = processes[0].period;

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

result = lcm(result, processes[i].period);

}

return result;

}

double utilization\_factor(Process processes[], int n) {

double sum = 0;

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

sum += (double)processes[i].burst\_time / processes[i].period;

}

return sum;

}

double rms\_threshold(int n) {

return n \* (pow(2.0, 1.0 / n) - 1);

}

void rate\_monotonic\_scheduling(Process processes[], int n) {

int lcm\_period = calculate\_lcm(processes, n);

printf("LCM=%d\n\n", lcm\_period);

printf("Rate Monotone Scheduling:\n");

printf("PID Burst Period\n");

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

printf("%d %d %d\n", processes[i].id, processes[i].burst\_time, processes[i].period);

}

double utilization = utilization\_factor(processes, n);

double threshold = rms\_threshold(n);

printf("\n%.6f <= %.6f => %s\n", utilization, threshold, (utilization <= threshold) ? "true" : "false");

if (utilization > threshold) {

printf("\nSystem may not be schedulable!\n");

return;

}

int timeline = 0, executed = 0;

while (timeline < lcm\_period) {

int selected = -1;

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

if (timeline % processes[i].period == 0) {

processes[i].remaining\_time = processes[i].burst\_time;

}

if (processes[i].remaining\_time > 0) {

selected = i;

break;

}

}

if (selected != -1) {

printf("Time %d: Process %d is running\n", timeline, processes[selected].id);

processes[selected].remaining\_time--;

executed++;

} else {

printf("Time %d: CPU is idle\n", timeline);

}

timeline++;

}

}

int main() {

int n;

Process processes[MAX\_PROCESSES];

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

scanf("%d", &n);

printf("Enter the CPU burst times:\n");

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

processes[i].id = i + 1;

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

processes[i].remaining\_time = processes[i].burst\_time;

}

printf("Enter the time periods:\n");

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

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

}

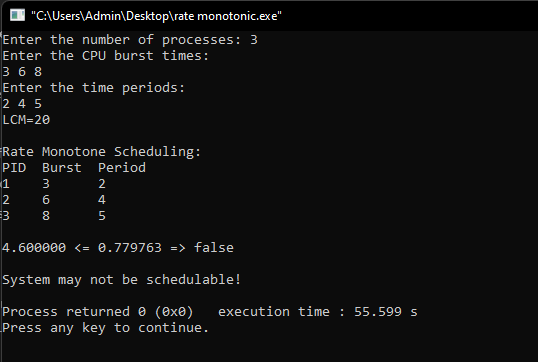
sort\_by\_period(processes, n);

rate\_monotonic\_scheduling(processes, n);

return 0;

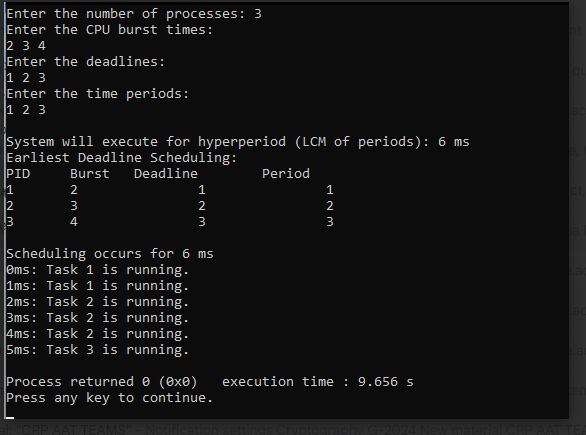
}

Output:



#include <stdio.h>  
  
int gcd(int a, int b) {  
    while (b != 0) {  
        int temp = b;  
        b = a % b;  
        a = temp;  
    }  
    return a;  
}  
  
int lcm(int a, int b) {  
    return (a \* b) / gcd(a, b);  
}  
  
struct Process {  
    int id, burst\_time, deadline, period;  
};  
  
void earliest\_deadline\_first(struct Process p[], int n, int time\_limit) {  
    int time = 0;  
    printf("Earliest Deadline Scheduling:\n");  
    printf("PID\tBurst\tDeadline\tPeriod\n");  
    for (int i = 0; i < n; i++) {  
        printf("%d\t%d\t\t%d\t\t%d\n", p[i].id, p[i].burst\_time, p[i].deadline, p[i].period);  
    }  
  
    printf("\nScheduling occurs for %d ms\n", time\_limit);  
    while (time < time\_limit) {  
        int earliest = -1;  
        for (int i = 0; i < n; i++) {  
            if (p[i].burst\_time > 0) {  
                if (earliest == -1 || p[i].deadline < p[earliest].deadline) {  
                    earliest = i;  
                }  
            }  
        }  
  
        if (earliest == -1) break;  
  
        printf("%dms: Task %d is running.\n", time, p[earliest].id);  
        p[earliest].burst\_time--;  
        time++;  
    }  
}  
  
int main() {  
    int n;  
    printf("Enter the number of processes: ");  
    scanf("%d", &n);  
  
    struct Process processes[n];  
    printf("Enter the CPU burst times:\n");  
    for (int i = 0; i < n; i++) {  
        scanf("%d", &processes[i].burst\_time);  
        processes[i].id = i + 1;  
    }  
  
    printf("Enter the deadlines:\n");  
    for (int i = 0; i < n; i++) {  
        scanf("%d", &processes[i].deadline);  
    }  
  
    printf("Enter the time periods:\n");  
    for (int i = 0; i < n; i++) {  
        scanf("%d", &processes[i].period);  
    }  
  
    int hyperperiod = processes[0].period;  
    for (int i = 1; i < n; i++) {  
        hyperperiod = lcm(hyperperiod, processes[i].period);  
    }  
  
    printf("\nSystem will execute for hyperperiod (LCM of periods): %d ms\n", hyperperiod);  
  
    earliest\_deadline\_first(processes, n, hyperperiod);  
  
    return 0;  
}

Output:



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

CODE:

#include <stdio.h>

#include <stdlib.h>

int mutex = 1;

int full = 0;

int empty = 2;

int x = 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("Producer produces item %d\n", x);

mutex = signal(mutex);

}

void consumer() {

mutex = wait(mutex);

full = wait(full);

empty = signal(empty);

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

x--;

mutex = signal(mutex);

}

int main() {

int choice;

while (1) {

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

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

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!\n");

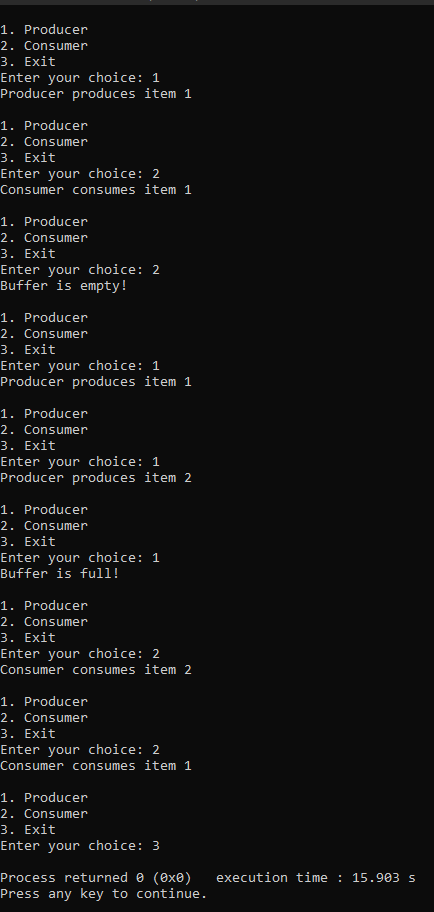
}

}

return 0;

}

Output:



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

CODE:

#include <stdio.h>

#include <stdlib.h>

#define MAX 10

int totalPhilosophers;

int hungry[MAX];

int areNeighbors(int a, int b) {

return (abs(a - b) == 1 || abs(a - b) == totalPhilosophers - 1);

}

void option1(int count) {

printf("\nAllow one philosopher to eat at any time\n");

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

printf("P %d is granted to eat\n", hungry[i]);

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

if (j != i) {

printf("P %d is waiting\n", hungry[j]);

}

}

}

}

void option2(int count) {

printf("\nAllow two philosophers to eat at same time\n");

int combination = 1;

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

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

if (!areNeighbors(hungry[i], hungry[j])) {

printf("combination %d\n", combination++);

printf("P %d and P %d are granted to eat\n", hungry[i], hungry[j]);

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

if (k != i && k != j) {

printf("P %d is waiting\n", hungry[k]);

}

}

printf("\n");

}

}

}

if (combination == 1) {

printf("No combinations found where two non-neighbor philosophers can eat.\n");

}

}

int main() {

int hungryCount;

printf("DINING PHILOSOPHER PROBLEM\n");

printf("Enter the total no. of philosophers: ");

scanf("%d", &totalPhilosophers);

printf("How many are hungry: ");

scanf("%d", &hungryCount);

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

printf("Enter philosopher %d position: ", i + 1);

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

}

int choice;

do {

printf("\n1. One can eat at a time 2. Two can eat at a time 3. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

option1(hungryCount);

break;

case 2:

option2(hungryCount);

break;

case 3:

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

break;

default:

printf("Invalid choice!\n");

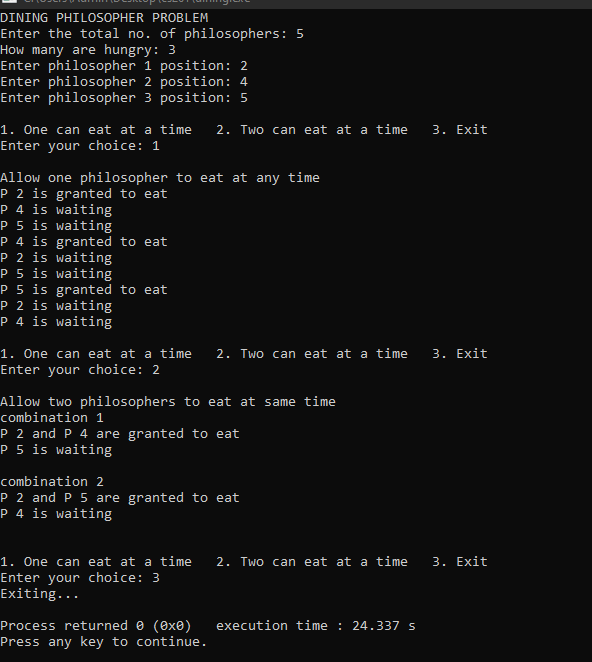
}

} while (choice != 3);

return 0;

}

Output:



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

CODE:

#include <stdio.h>

#include <stdbool.h>

int main() {

int n, m;

printf("Enter number of processes and resources:\n");

scanf("%d %d", &n, &m);

int alloc[n][m], max[n][m], avail[m];

printf("Enter allocation matrix:\n");

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

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

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

printf("Enter max matrix:\n");

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

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

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

printf("Enter available matrix:\n");

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

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

int need[n][m];

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

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

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

bool finish[n];

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

finish[i] = false;

int safeSeq[n];

int work[m];

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

work[i] = avail[i];

int count = 0;

while (count < n) {

bool found = false;

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

if (!finish[p]) {

int j;

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

if (need[p][j] > work[j])

break;

if (j == m) {

for (int k = 0; k < m; k++)

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

safeSeq[count++] = p;

finish[p] = true;

found = true;

}

}

}

if (!found) {

printf("System is not in a safe state.\n");

return 0;

}

}

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

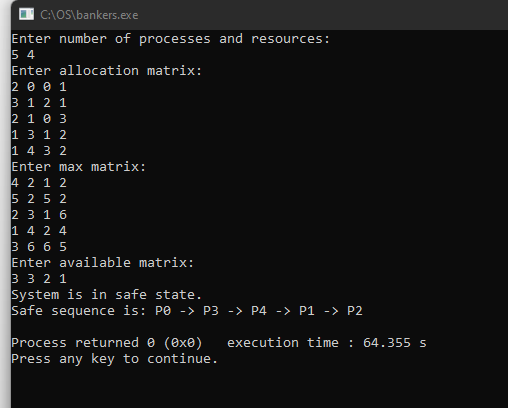
for (int i = 0; i < n; i++)

printf("P%d%s", safeSeq[i], (i == n - 1) ? "\n" : " -> ");

return 0;

}

Output:



8. Write a C program to simulate deadlock detection

CODE:

#include <stdio.h>

#include <stdbool.h>

int main() {

int n, m;

printf("Enter number of processes and number of resources:\n");

scanf("%d %d", &n, &m);

int alloc[n][m], request[n][m], avail[m];

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

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

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

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

printf("Enter Request Matrix:\n");

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

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

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

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

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

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

int work[m];

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

work[i] = avail[i];

bool finish[n];

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

bool hasAllocation = false;

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

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

hasAllocation = true;

break;

}

}

finish[i] = hasAllocation ? false : true;

}

while (true) {

bool progress = false;

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

if (!finish[i]) {

bool canGrant = true;

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

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

canGrant = false;

break;

}

}

if (canGrant) {

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

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

finish[i] = true;

progress = true;

}

}

}

if (!progress)

break;

}

printf("\nDeadlock Detection Result:\n");

bool deadlock = false;

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

if (!finish[i]) {

printf("Process P%d is deadlocked\n", i);

deadlock = true;

} else {

printf("Process P%d is not deadlocked\n", i);

}

}

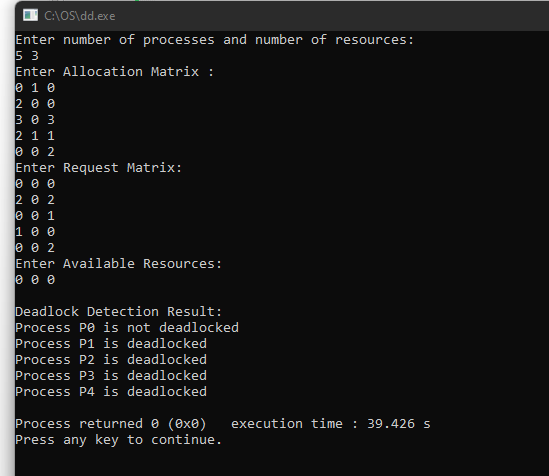
if (!deadlock)

printf("\nNo deadlock detected in the system.\n");

return 0;

}

Output:



9. Write a C program to simulate the following contiguous memory allocation techniques a) Worst-fit

b)Best-fit

c)First-fit

CODE:

#include <stdio.h>

struct Block {

int size;

int allocated;

};

struct File {

int size;

int block\_no;

};

void resetBlocks(struct Block blocks[], int n) {

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

blocks[i].allocated = 0;

}

}

void firstFit(struct Block blocks[], int n\_blocks, struct File files[], int n\_files) {

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

printf("File\_no:\tFile\_size\tBlock\_no:\tBlock\_size:\n");

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

files[i].block\_no = -1;

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

if (!blocks[j].allocated && blocks[j].size >= files[i].size) {

files[i].block\_no = j + 1;

blocks[j].allocated = 1;

printf("%d\t\t%d\t\t%d\t\t%d\n", i + 1, files[i].size, j + 1, blocks[j].size);

break;

}

}

if (files[i].block\_no == -1) {

printf("%d\t\t%d\t\t\_\t\t\_\n", i + 1, files[i].size);

}

}

}

void bestFit(struct Block blocks[], int n\_blocks, struct File files[], int n\_files) {

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

printf("File\_no:\tFile\_size\tBlock\_no:\tBlock\_size:\n");

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

int bestIdx = -1;

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

if (!blocks[j].allocated && blocks[j].size >= files[i].size) {

if (bestIdx == -1 || blocks[j].size < blocks[bestIdx].size) {

bestIdx = j;

}

}

}

if (bestIdx != -1) {

blocks[bestIdx].allocated = 1;

files[i].block\_no = bestIdx + 1;

printf("%d\t\t%d\t\t%d\t\t%d\n", i + 1, files[i].size, bestIdx + 1, blocks[bestIdx].size);

} else {

printf("%d\t\t%d\t\t\_\t\t\_\n", i + 1, files[i].size);

}

}

}

void worstFit(struct Block blocks[], int n\_blocks, struct File files[], int n\_files) {

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

printf("File\_no:\tFile\_size\tBlock\_no:\tBlock\_size:\n");

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

int worstIdx = -1;

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

if (!blocks[j].allocated && blocks[j].size >= files[i].size) {

if (worstIdx == -1 || blocks[j].size > blocks[worstIdx].size) {

worstIdx = j;

}

}

}

if (worstIdx != -1) {

blocks[worstIdx].allocated = 1;

files[i].block\_no = worstIdx + 1;

printf("%d\t\t%d\t\t%d\t\t%d\n", i + 1, files[i].size, worstIdx + 1, blocks[worstIdx].size);

} else {

printf("%d\t\t%d\t\t\_\t\t\_\n", i + 1, files[i].size);

}

}

}

int main() {

int n\_blocks, n\_files, choice;

printf("Memory Management Scheme\n");

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

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

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

scanf("%d", &n\_files);

struct Block blocks[n\_blocks];

struct File files[n\_files];

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

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

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

scanf("%d", &blocks[i].size);

blocks[i].allocated = 0;

}

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

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

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

scanf("%d", &files[i].size);

}

do {

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

printf("Enter your choice: ");

scanf("%d", &choice);

resetBlocks(blocks, n\_blocks); // Reset block allocation before each strategy

switch (choice) {

case 1:

firstFit(blocks, n\_blocks, files, n\_files);

break;

case 2:

bestFit(blocks, n\_blocks, files, n\_files);

break;

case 3:

worstFit(blocks, n\_blocks, files, n\_files);

break;

case 4:

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

break;

default:

printf("Invalid choice.\n");

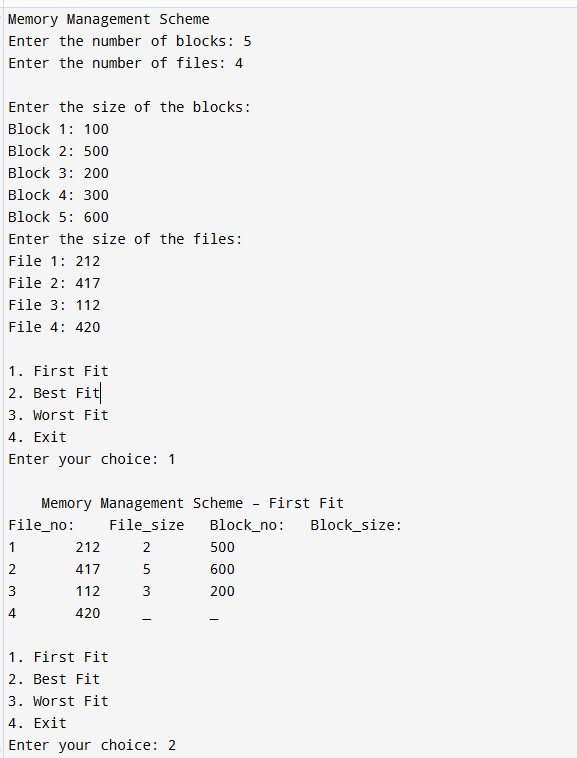
}

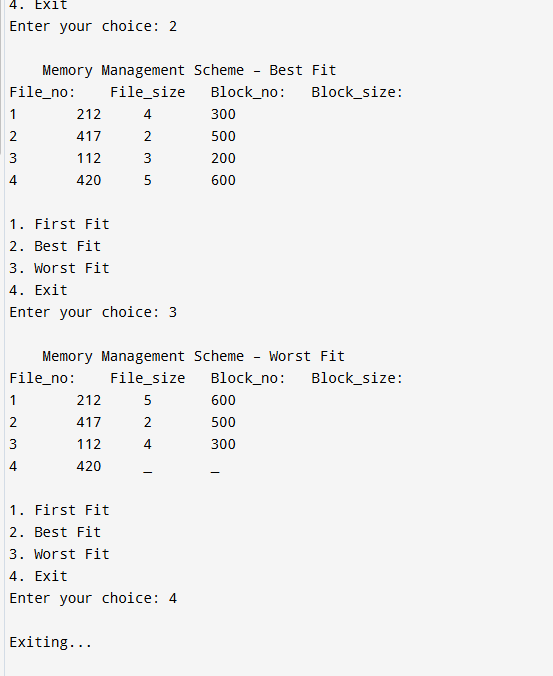
} while (choice != 4);

return 0;

}

Output





10. Write a C program to simulate page replacement algorithms a) FIFO

1. LRU
2. Optimal

CODE:

1. **FIFO**

#include <stdio.h>

int main() {

int frames, pages[50], n, frame[10], i, j, k, avail, count = 0;

printf("Enter number of pages: ");

scanf("%d", &n);

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

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

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

printf("Enter number of frames: ");

scanf("%d", &frames);

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

frame[i] = -1;

printf("\nPage\tFrames\t\tPage Fault\n");

j = 0;

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

avail = 0;

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

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

avail = 1;

break;

}

}

if(avail == 0) {

frame[j] = pages[i];

j = (j + 1) % frames;

count++;

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

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

if(frame[k] != -1)

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

else

printf("- ");

}

printf("\tYes\n");

} else {

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

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

if(frame[k] != -1)

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

else

printf("- ");

}

printf("\tNo\n");

}

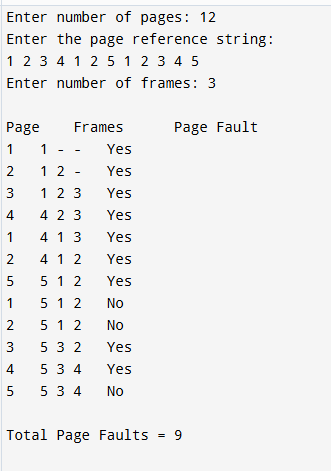
}

printf("\nTotal Page Faults = %d\n", count);

return 0;

}

Output:



**b)recently used**

#include <stdio.h>

int main() {

int n, frames, i, j, k, faults = 0;

printf("Enter number of pages: ");

scanf("%d", &n);

int pages[n];

printf("Enter the reference string: ");

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

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

printf("Enter number of frames: ");

scanf("%d", &frames);

int frame\_arr[frames];

int time[frames];

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

frame\_arr[i] = -1;

time[i] = 0;

}

int counter = 0;

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

int flag = 0;

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

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

flag = 1;

counter++;

time[j] = counter;

break;

}

}

if(flag == 0) {

faults++;

int min\_time = time[0], min\_pos = 0;

for(k = 1; k < frames; k++) {

if(time[k] < min\_time) {

min\_time = time[k];

min\_pos = k;

}

}

frame\_arr[min\_pos] = pages[i];

counter++;

time[min\_pos] = counter;

}

printf("Frames after accessing %d: ", pages[i]);

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

if(frame\_arr[j] == -1)

printf("- ");

else

printf("%d ", frame\_arr[j]);

}

printf("\n");

}

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

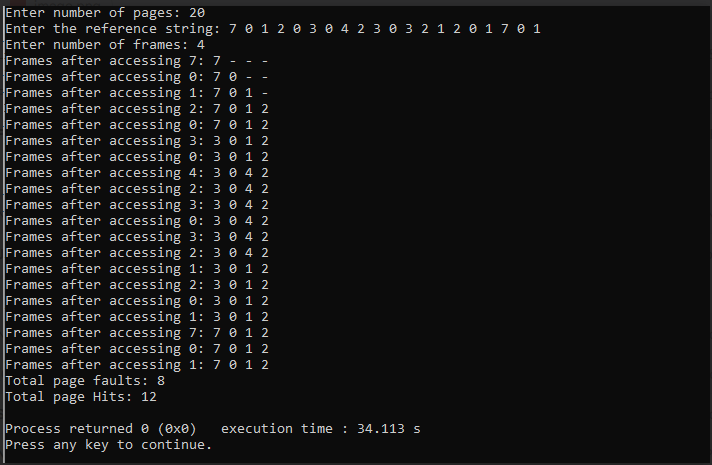
int Hits = n-faults;

printf("Total page Hits: %d\n",Hits);

return 0;

}

Output:



**c)optimal**

#include <stdio.h>

int main() {

int n, frames, i, j, k, faults = 0;

printf("Enter number of pages: ");

scanf("%d", &n);

int pages[n];

printf("Enter the reference string: ");

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

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

printf("Enter number of frames: ");

scanf("%d", &frames);

int frame\_arr[frames];

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

frame\_arr[i] = -1;

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

int flag = 0;

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

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

flag = 1;

break;

}

}

if(flag == 0) {

faults++;

int pos = -1;

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

if(frame\_arr[j] == -1) {

pos = j;

break;

}

}

if(pos == -1) {

int farthest = i, replace\_index = 0;

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

int found = 0;

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

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

if(k > farthest) {

farthest = k;

replace\_index = j;

}

found = 1;

break;

}

}

if(!found) {

replace\_index = j;

break;

}

}

pos = replace\_index;

}

frame\_arr[pos] = pages[i];

}

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

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

if(frame\_arr[j] == -1)

printf("\_ ");

else

printf("%d ", frame\_arr[j]);

}

printf("\n");

}

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

int Hits = n-faults;

printf("Total page Hits: %d\n",Hits);

return 0;

}

Output:

