NAME : K.OMKAR

REG.NO :192111134

SUBJECT CODE:CSA0413

SUBJECT NAME:OPERATING SYSTEMS

1.Consider a system with 4 processes and 3 resources with the given resource matrices.

Claim matrix Allocation matrix

3 2 2 1 0 0

6 1 3 6 1 2

3 1 4 2 1 1

4 2 2 0 0 2

The resource vector is [9,3,6]. Write a C program to determine if the system is in safe or unsafe state.

PROGRAM:

#include <stdio.h>

#include <stdbool.h>

#define NUM\_PROCESSES 4

#define NUM\_RESOURCES 3

int main() {

// initialize the resource matrices

int claim[NUM\_PROCESSES][NUM\_RESOURCES] = {

{3, 2, 2},

{6, 1, 3},

{3, 1, 4},

{4, 2, 2}

};

int allocation[NUM\_PROCESSES][NUM\_RESOURCES] = {

{1, 0, 0},

{6, 1, 2},

{2, 1, 1},

{0, 0, 2}

};

int available[NUM\_RESOURCES] = {9, 3, 6};

// initialize the work and finish arrays

int work[NUM\_RESOURCES];

bool finish[NUM\_PROCESSES];

for (int i = 0; i < NUM\_RESOURCES; i++) {

work[i] = available[i];

}

for (int i = 0; i < NUM\_PROCESSES; i++) {

finish[i] = false;

}

// find a process that can run

bool found = true;

while (found) {

found = false;

for (int i = 0; i < NUM\_PROCESSES; i++) {

if (!finish[i]) {

bool can\_run = true;

for (int j = 0; j < NUM\_RESOURCES; j++) {

if (claim[i][j] - allocation[i][j] > work[j]) {

can\_run = false;

break;

}

}

if (can\_run) {

found = true;

finish[i] = true;

for (int j = 0; j < NUM\_RESOURCES; j++) {

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

}

}

}

}

}

// check if all processes have finished

bool safe = true;

for (int i = 0; i < NUM\_PROCESSES; i++) {

if (!finish[i]) {

safe = false;

break;

}

}

// print the result

if (safe) {

printf("The system is in a safe state.\n");

} else {

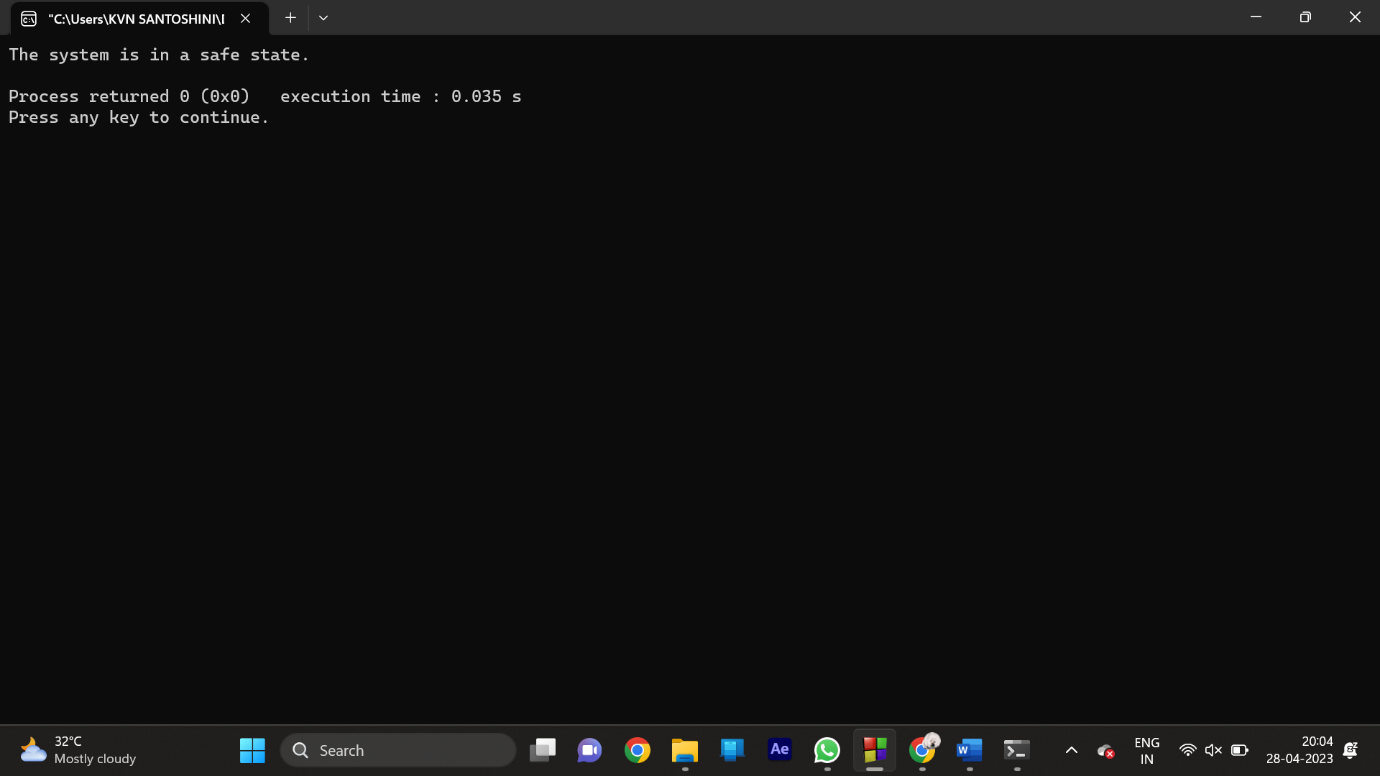
printf("The system is in an unsafe state.\n");

}

return 0;

}

OUTPUT:



2.Write a C program to illustrate the FIFO method of page replacement and determine the number of page faults for the following test case:

No of page frames: 3; Page reference sequence: 4, 1, 2, 4, 3, 2, 1 and 5.

PROGRAM:

#include <stdio.h>

#define MAX\_FRAMES 10

int main() {

int pages[MAX\_FRAMES]; // array to store the pages in frames

int page\_count = 0; // number of pages currently in frames

int page\_faults = 0; // number of page faults

int i, j, k;

// initialize all pages to -1 to indicate an empty frame

for (i = 0; i < MAX\_FRAMES; i++) {

pages[i] = -1;

}

// read the page sequence from input

int n;

printf("Enter number of pages: ");

scanf("%d", &n);

int sequence[n];

printf("Enter page sequence: ");

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

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

}

// perform FIFO page replacement

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

int page = sequence[i];

int page\_found = 0;

// check if page is already in frames

for (j = 0; j < page\_count; j++) {

if (pages[j] == page) {

page\_found = 1;

break;

}

}

// if page is not in frames, replace the oldest page with the new page

if (!page\_found) {

if (page\_count < MAX\_FRAMES) {

// add new page to an empty frame

pages[page\_count] = page;

page\_count++;

} else {

// replace the oldest page with the new page

for (j = 0; j < MAX\_FRAMES - 1; j++) {

pages[j] = pages[j+1];

}

pages[MAX\_FRAMES - 1] = page;

}

page\_faults++;

}

// print the current state of the frames after each page is processed

printf("Page %d: ", page);

for (j = 0; j < page\_count; j++) {

printf("%d ", pages[j]);

}

for (j = page\_count; j < MAX\_FRAMES; j++) {

printf("- ");

}

printf("\n");

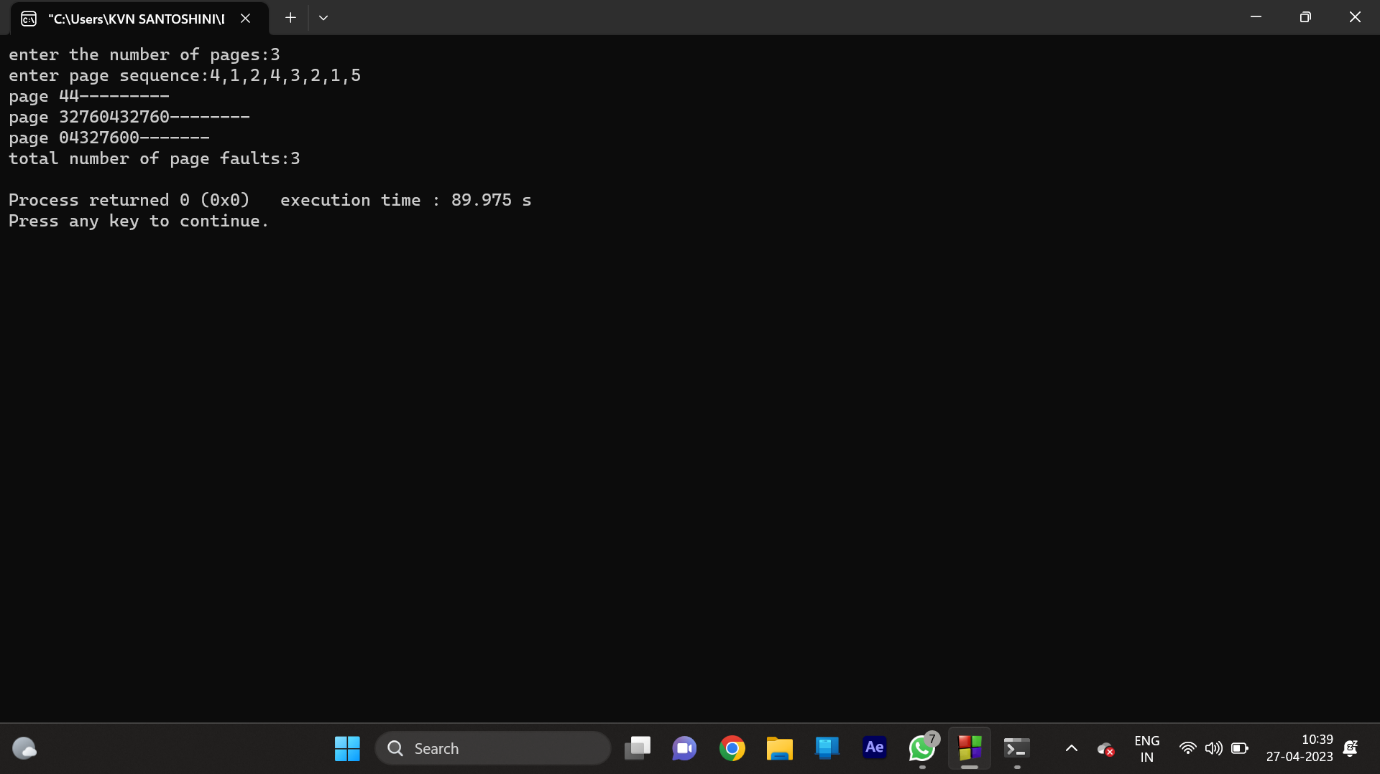
}

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

return 0;

}

OUTPUT:



3.Write a program to compute the average waiting time and average turnaround time based on Non Preemptive Shortest-Job-First Scheduling for the following process with the given CPU burst times, ( and the assumption that all jobs arrive at the same time.)

Process Burst Time

P1 6

P2 8

P3 7

P4 3

PROGRAM:

#include <stdio.h>

#define MAX\_PROCESSES 10

int main() {

int n; // number of processes

int burst\_times[MAX\_PROCESSES]; // array to store burst times

int waiting\_times[MAX\_PROCESSES]; // array to store waiting times

int turnaround\_times[MAX\_PROCESSES]; // array to store turnaround times

int i, j, temp;

// read the number of processes and their burst times

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

scanf("%d", &n);

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

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

scanf("%d", &burst\_times[i]);

}

// sort the burst times in ascending order using selection sort

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

int min\_idx = i;

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

if (burst\_times[j] < burst\_times[min\_idx]) {

min\_idx = j;

}

}

// swap burst times

temp = burst\_times[i];

burst\_times[i] = burst\_times[min\_idx];

burst\_times[min\_idx] = temp;

}

// compute waiting times and turnaround times

waiting\_times[0] = 0;

turnaround\_times[0] = burst\_times[0];

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

waiting\_times[i] = waiting\_times[i-1] + burst\_times[i-1];

turnaround\_times[i] = turnaround\_times[i-1] + burst\_times[i];

}

// compute average waiting time and average turnaround time

float avg\_waiting\_time = 0.0;

float avg\_turnaround\_time = 0.0;

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

avg\_waiting\_time += waiting\_times[i];

avg\_turnaround\_time += turnaround\_times[i];

}

avg\_waiting\_time /= n;

avg\_turnaround\_time /= n;

// print the results

printf("Process\tBurst Time\tWaiting Time\tTurnaround Time\n");

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

printf("%d\t%d\t\t%d\t\t%d\n", i+1, burst\_times[i], waiting\_times[i], turnaround\_times[i]);

}

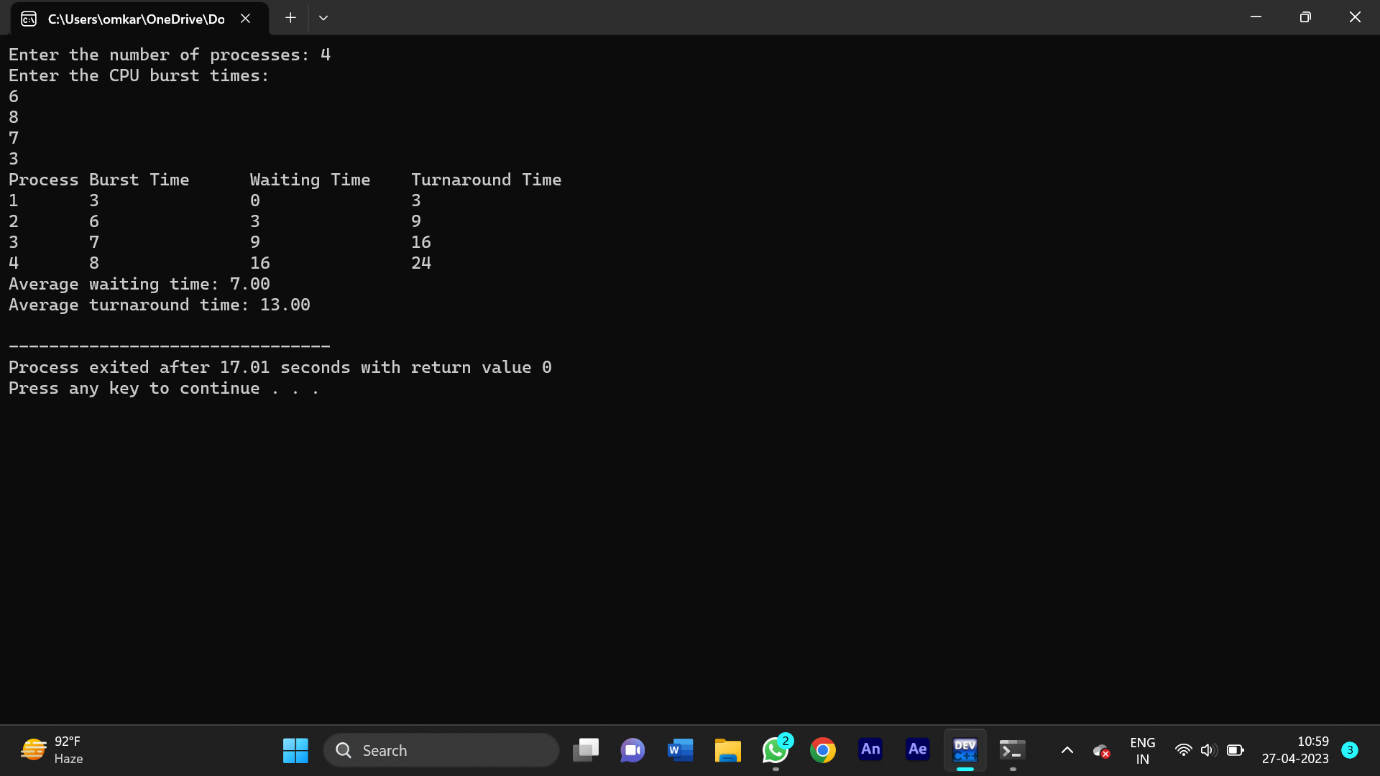
printf("Average waiting time: %.2f\n", avg\_waiting\_time);

printf("Average turnaround time: %.2f\n", avg\_turnaround\_time);

return 0;

}

OUTPUT:



1. Write a C program to implement the first-fit algorithm for memory management.

Test Case:

Memory partitions: 300 KB, 600 KB, 350 KB, 200 KB, 750 KB, and 125 KB (in order) Show the outcome for the test case with first-fit algorithms to place the processes of size 115 KB, 500 KB, 358 KB, 200 KB, and 375 KB (in order)

PROGRAM:

#include <stdio.h>

#define MAX\_BLOCKS 100

#define MAX\_PROCESSES 100

int main() {

int memory\_size, n\_blocks, n\_processes;

int block\_sizes[MAX\_BLOCKS], block\_flags[MAX\_BLOCKS];

int process\_sizes[MAX\_PROCESSES], process\_flags[MAX\_PROCESSES];

int i, j;

// read the memory size and number of memory blocks

printf("Enter the total memory size: ");

scanf("%d", &memory\_size);

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

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

// read the sizes of each memory block

printf("Enter the size of each memory block:\n");

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

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

block\_flags[i] = 0; // initialize all blocks to be free

}

// read the number of processes and their sizes

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

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

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

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

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

process\_flags[i] = 0; // initialize all processes to be not allocated

}

// allocate memory to processes using first-fit algorithm

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

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

if (process\_flags[i] == 0 && block\_flags[j] == 0 && process\_sizes[i] <= block\_sizes[j]) {

// allocate the memory block to the process

block\_flags[j] = 1;

process\_flags[i] = 1;

printf("Process %d allocated to memory block %d\n", i+1, j+1);

}

}

if (process\_flags[i] == 0) {

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

}

}

return 0;

}

OUTPUT:

