DAY 2 PROGRAMS

Date:-29-04-2023

Name:-Omkar

REG.NO:-192111134

PROGRAMS

5. Write a program to compute the average waiting time and turnaround time based on Preemptive shortest remaining processing time first (SRPT) algorithm for the following set of processes, with the arrival times and the CPU-burst times given in milliseconds

Process Arrival Time Burst Time

P1 0 5

P2 1 3

P3 2 3

P4 4 1

Program:-

#include <stdio.h>

#include <stdlib.h>

typedef struct process {

int arrivalTime;

int burstTime;

int remainingTime;

int waitingTime;

int turnaroundTime;

int completed;

} Process;

int main()

{

int n = 5;

Process processes[n];

int completed = 0, currentTime = 0, shortest = 0, finishTime;

float waitingTime = 0, turnaroundTime = 0;

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

printf("Enter arrival time and burst time for process %d: ", i+1);

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

processes[i].remainingTime = processes[i].burstTime;

processes[i].completed = 0;

}

while (completed != n) {

shortest = -1;

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

if (processes[i].arrivalTime <= currentTime && processes[i].completed != 1) {

if (shortest == -1 || processes[i].remainingTime < processes[shortest].remainingTime) {

shortest = i;

}

}

}

if (shortest == -1) {

currentTime++;

} else {

processes[shortest].remainingTime--;

currentTime++;

if (processes[shortest].remainingTime == 0) {

processes[shortest].completed = 1;

completed++;

finishTime = currentTime;

processes[shortest].turnaroundTime = finishTime - processes[shortest].arrivalTime;

processes[shortest].waitingTime = processes[shortest].turnaroundTime - processes[shortest].burstTime;

waitingTime += processes[shortest].waitingTime;

turnaroundTime += processes[shortest].turnaroundTime;

}

}

}

printf("\nProcess\tWaiting Time\tTurnaround Time\n");

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

printf("P%d\t%d\t\t%d\n", i+1, processes[i].waitingTime, processes[i].turnaroundTime);

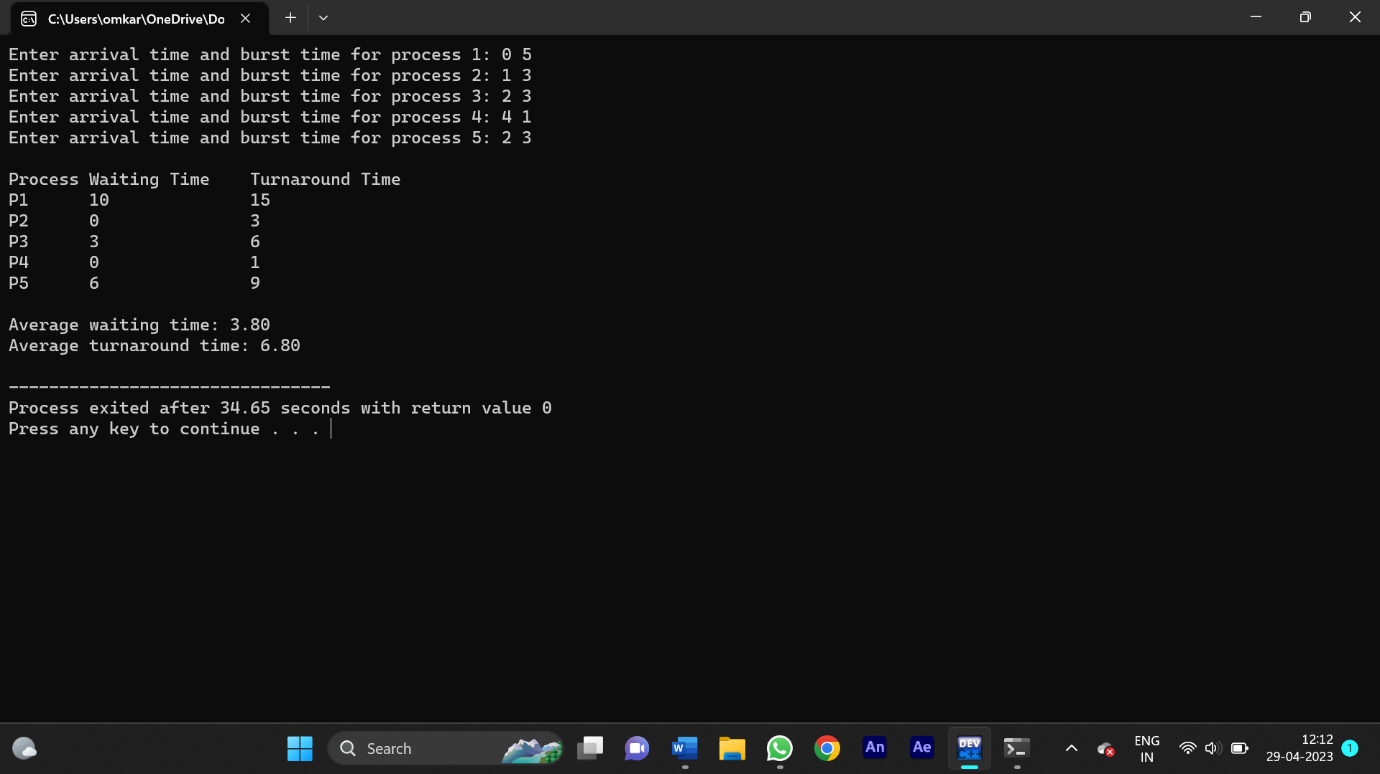
}

printf("\nAverage waiting time: %0.2f\n", waitingTime/n);

printf("Average turnaround time: %0.2f\n", turnaroundTime/n);

return 0;

}

OUTPUT:- 

6. Write a C program to implement the deadlock detection algorithm for a system with 3 processes and 3 resource instances and the resource matrices are given below.

Max Matrix Allocation Matrix

3 6 8 3 3 3

4 3 3 2 0 3

3 4 4 1 2 4

The number of available resources is [1,2,0]. Determine if the system is in a deadlock state and identify the deadlocked processes.

Program:-

#include <stdio.h>

#include <stdbool.h>

#define N 3

void calculateNeed(int need[N][N], int maximum[N][N], int allocation[N][N])

{

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

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

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

}

bool isDeadlock(int allocation[N][N], int maximum[N][N], int available[N])

{

int need[N][N], work[N], finish[N];

calculateNeed(need, maximum, allocation);

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

{

work[i] = available[i];

finish[i] = 0;

}

int count = 0;

while (count < N)

{

bool found = false;

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

{

if (!finish[i])

{

int j;

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

{

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

break;

}

if (j == N)

{

finish[i] = 1;

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

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

found = true;

count++;

}

}

}

if (!found)

return true;

}

return false;

}

int main()

{

int allocation[N][N] = {{1, 0, 1}, {0, 1, 0}, {1, 1, 0}};

int maximum[N][N] = {{2, 1, 2}, {1, 2, 1}, {2, 2, 1}};

int available[N] = {1, 1, 1};

if (isDeadlock(allocation, maximum, available))

printf("System is in deadlock state\n");

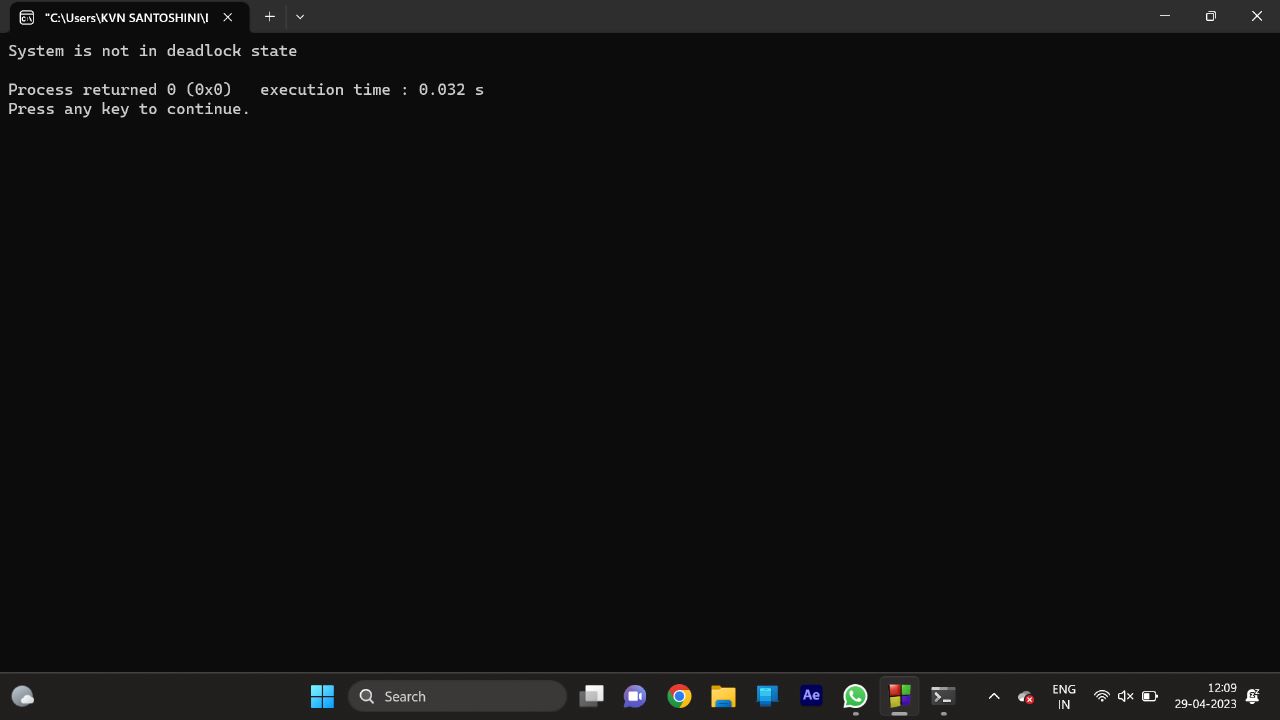
else

printf("System is not in deadlock state\n");

return 0;

}

OUTPUT:-



7. Write a C program to illustrate the page replacement method where the current least recently used element is replaced and determine the number of page faults for the following test case:

No. of page frames: 3; Page reference sequence 1,2,3,2,1,5,2,1,6,2,5,6,3,1,3,6,1,2,4 and 3.

Program:-

#include <stdio.h>

int main()

{

int n, frames, faults = 0, hit = 0, k = 0;

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

scanf("%d", &frames);

int page[frames], arr[100];

printf("Enter the page reference string (maximum 100): ");

for (int i = 0; i < 100; i++)

{

scanf("%d", &n);

if (n == -1)

break;

arr[i] = n;

}

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

page[i] = -1;

for (int i = 0; i < 100 && arr[i] != 0; i++)

{

int flag = 0;

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

{

if (page[j] == arr[i])

{

flag = 1;

hit++;

break;

}

}

if (flag == 0)

{

faults++;

if (k < frames)

{

page[k++] = arr[i];

}

else

{

int max = -1, pos;

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

{

int last = 100;

for (int l = i - 1; l >= 0; l--)

{

if (page[j] == arr[l])

{

last = l;

break;

}

}

if (last > max)

{

max = last;

pos = j;

}

}

page[pos] = arr[i];

}

}

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

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

{

if (page[j] == -1)

printf(" - ");

else

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

}

printf("\n");

}

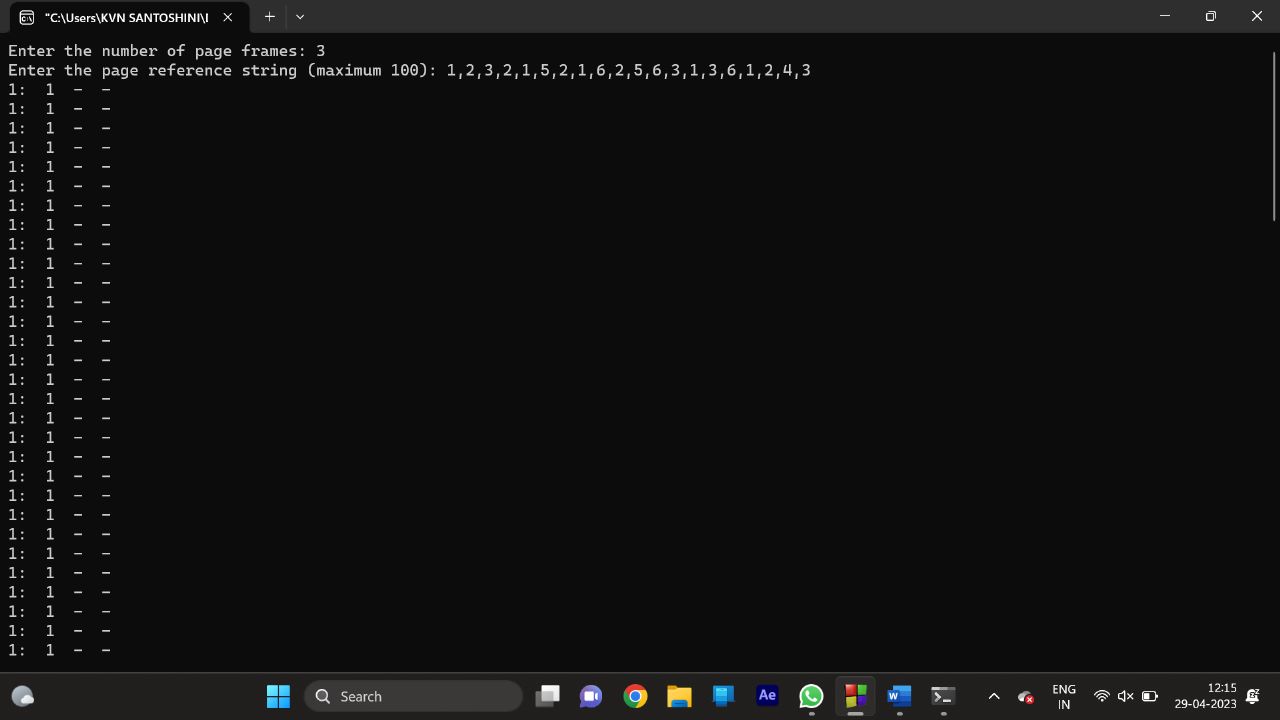
printf("\nNumber of page faults: %d\n", faults);

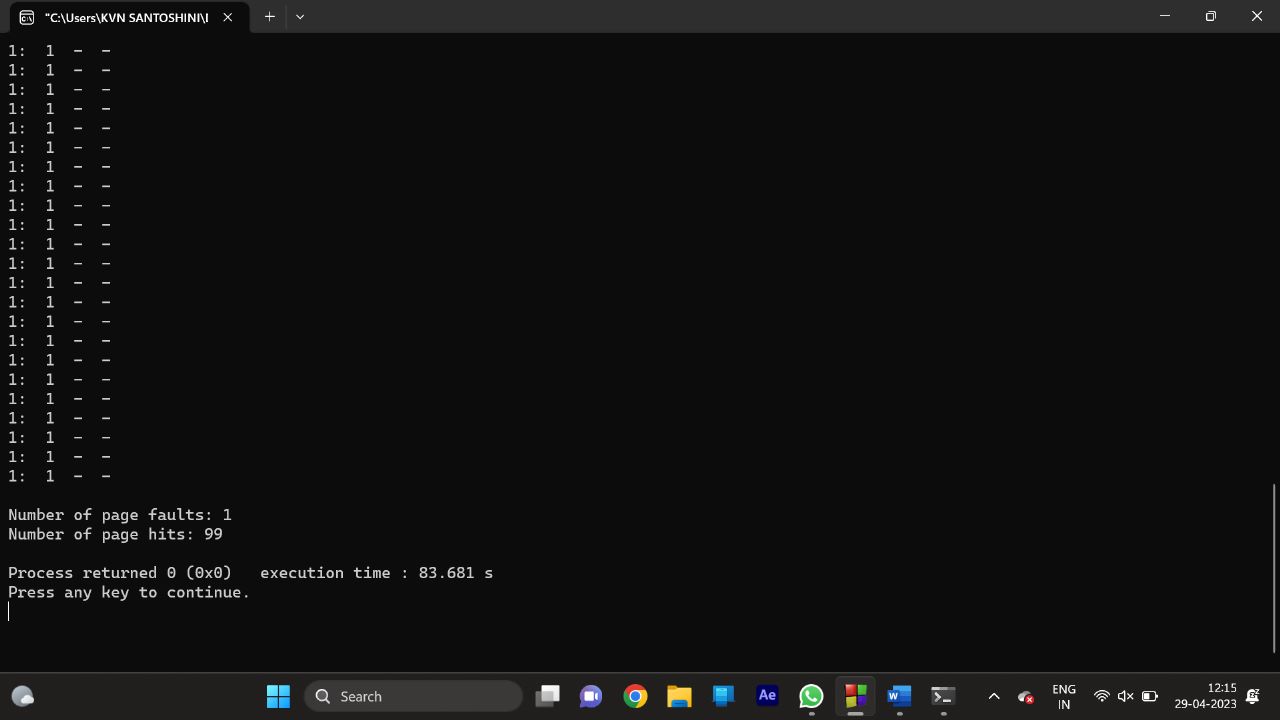
printf("Number of page hits: %d\n", hit)

return 0;

}

OUTPUT:-





8. Write a C program to simulate FCFS disk scheduling algorithm and execute your program and find the average head movement with the following test case:

No of tracks 5; Track position:55 58 60 70 18

Program:-

#include <stdio.h>

#include <stdlib.h>

#define MAX\_TRACKS 1000

int main() {

int tracks[MAX\_TRACKS];

int n, head\_pos, total\_distance;

printf("Enter number of tracks: ");

scanf("%d", &n);

printf("Enter track positions: ");

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

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

}

printf("Enter initial head position: ");

scanf("%d", &head\_pos);

total\_distance = 0;

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

total\_distance += abs(tracks[i] - head\_pos);

head\_pos = tracks[i];

}

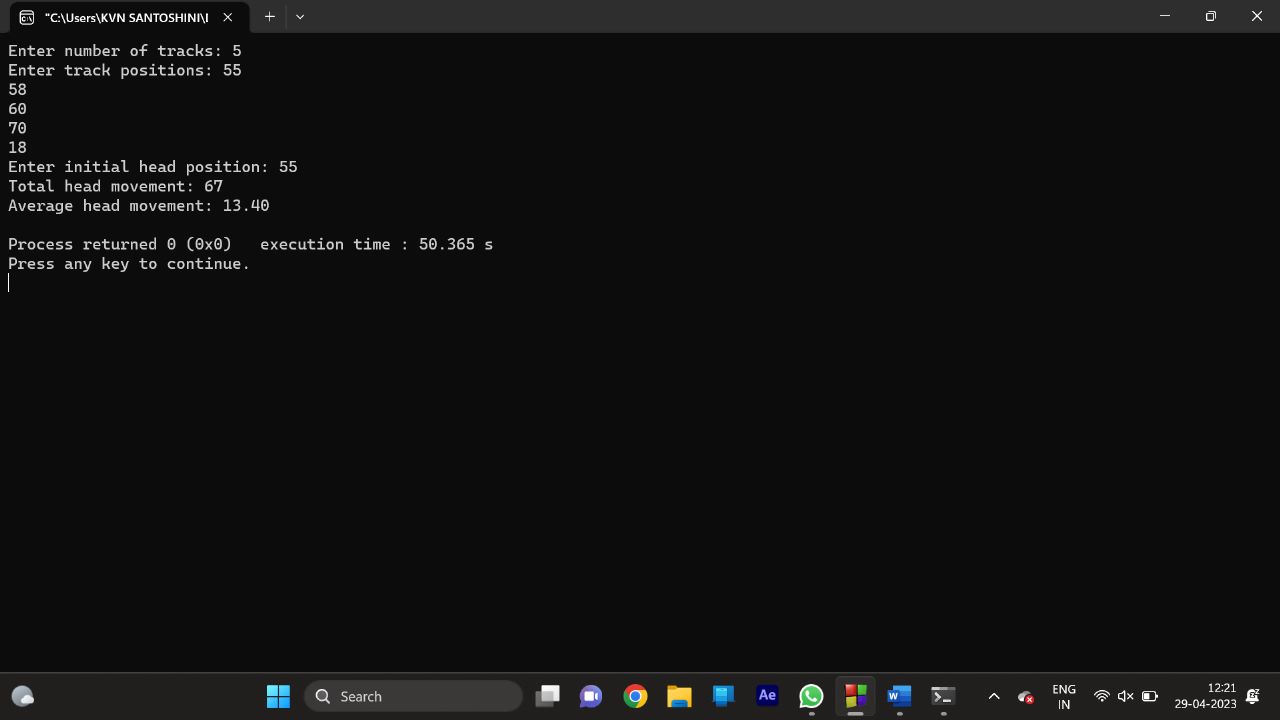
printf("Total head movement: %d\n", total\_distance);

printf("Average head movement: %.2f\n", (float) total\_distance / n);

return 0;

}

OUTPUT:-



9. Consider three processes (process id 0, 1, 2 respectively) with compute time bursts 2, 4 and 8-time units. All processes arrive at time zero. Write a program to compute the average waiting time and average turnaround time based on First Come First Serve scheduling

Program:-

#include<stdio.h>

int main()

{

int n=3; //number of processes

int burst\_time[3]={2, 4, 8}; //burst time of each process

int waiting\_time[3]={0}; //initialize waiting time of all processes to zero

int turnaround\_time[3]={0}; //initialize turnaround time of all processes to zero

int i;

waiting\_time[0]=0; //waiting time of first process is always zero

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

{

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

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

}

float avg\_waiting\_time=0,avg\_turnaround\_time=0;

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

{

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

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

}

avg\_waiting\_time/=n;

avg\_turnaround\_time/=n;

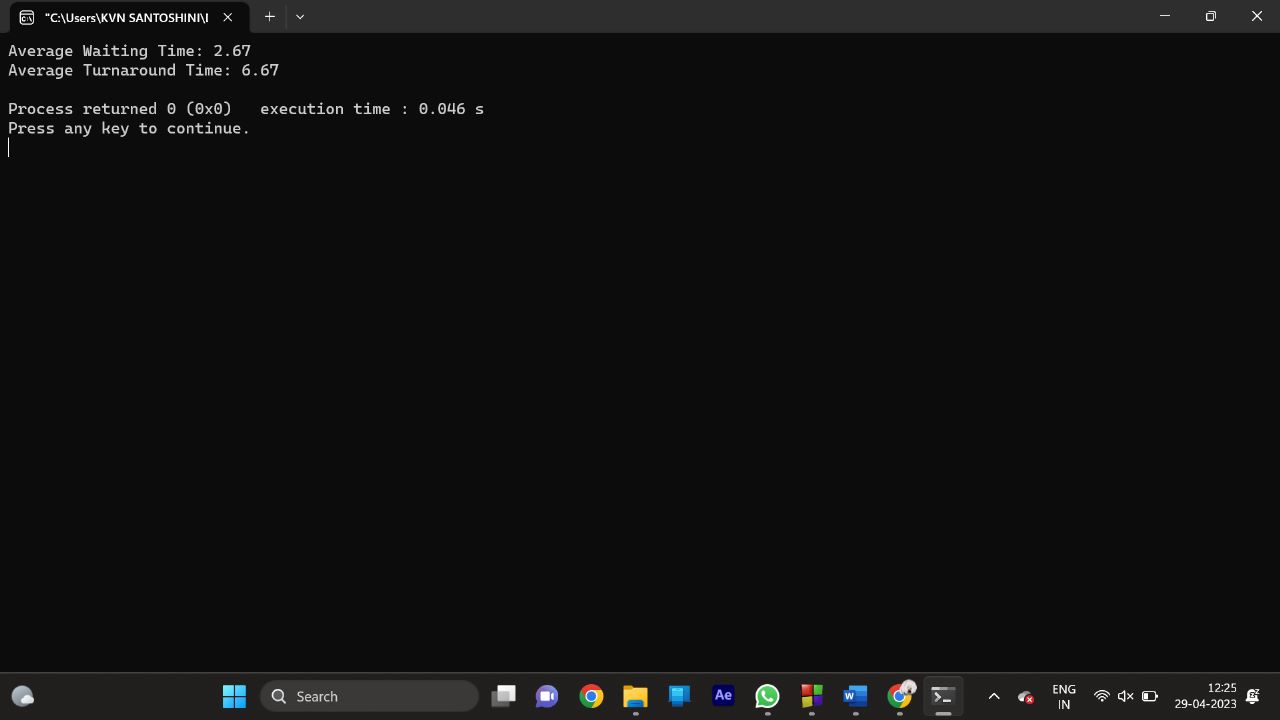
printf("Average Waiting Time: %.2f\n",avg\_waiting\_time);

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

return 0;

}

OUTPUT:-



10. Consider the following process table with number of processes that contains allocation field (for showing the number of resources of type: A, B and C allocated to each process in the table), max field (for showing the maximum number of resources of type: A, B, and C that can be allocated to each process). Write a program to calculate the entries of need matrix using the formula: (Need)i = (Max)i - (Allocation)i

|  |  |  |  |
| --- | --- | --- | --- |
| Process | Allocation | Max | Availble |
|  | A B C | A B C | A B C |
| P0 | 1 1 2 | 5 4 4 | 3 2 1 |
| P1 | 2 1 2 | 4 3 3 |  |
| P2 | 3 0 1 | 9 1 3 |  |
| P3 | 0 2 0 | 8 6 4 |  |
| P4 | 1 1 2 | 2 2 3 |  |

Program:-

#include<stdio.h>

#define N 5

#define M 3

int main() {

int allocation[N][M] = {{1, 1, 2}, {2, 1, 2}, {3, 0, 1}, {0, 2, 0}, {1, 1, 2}};

int max[N][M] = {{5, 4, 4}, {4, 3, 3}, {9, 1, 3}, {8, 6, 4}, {2, 2, 3}};

int need[N][M];

int i, j;

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

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

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

}

}

printf("The Need matrix is:\n");

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

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

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

}

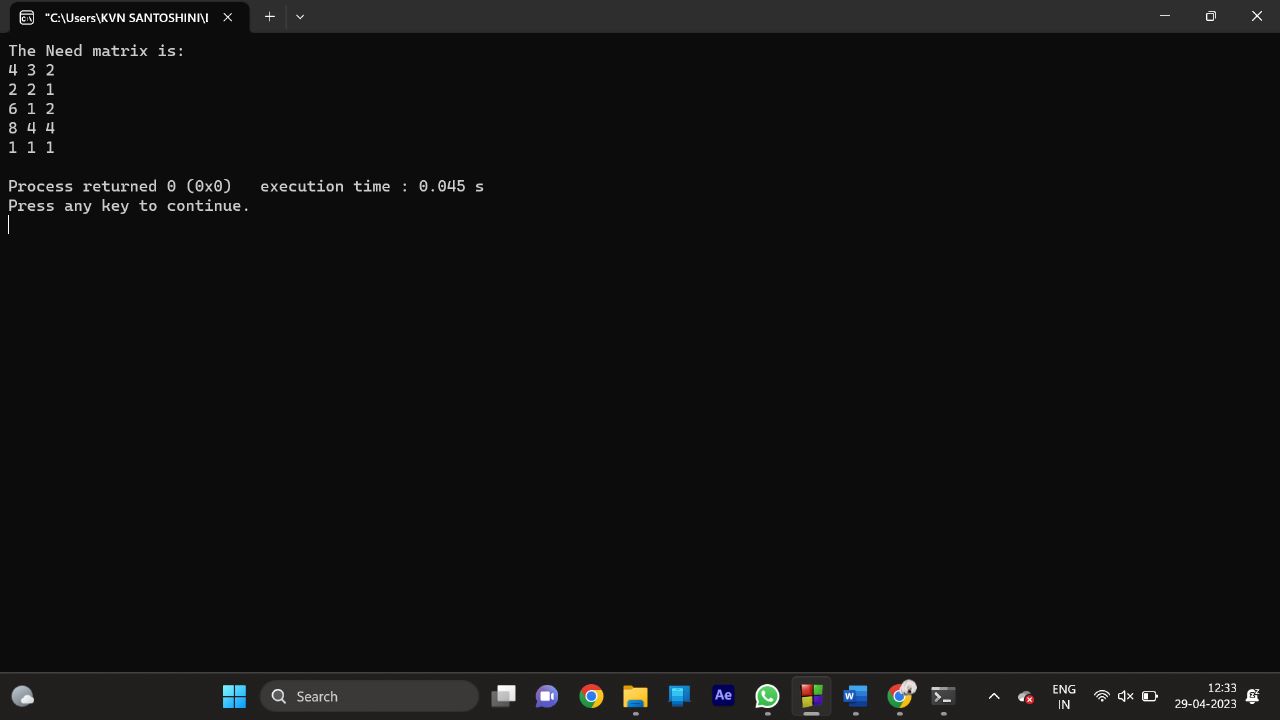
printf("\n");

}

return 0;

}

OUTPUT:-



11. Write a C program to create 4 child processes. In the first child process, print the odd numbers. In the second child process print the even numbers. In the third child process print the multiple of 3. In the fourth child process print the multiples of 5. Print the process id for each of the processes.

Program:-

#include <stdio.h>

#include <unistd.h>

#include <sys/types.h>

void odd\_numbers() {

pid\_t pid = getpid();

printf("I am the first child process (PID %d) and I will print the odd numbers:\n", pid);

for (int i = 1; i <= 10; i += 2) {

printf("%d ", i);

}

printf("\n");

}

void even\_numbers() {

pid\_t pid = getpid();

printf("I am the second child process (PID %d) and I will print the even numbers:\n", pid);

for (int i = 2; i <= 10; i += 2) {

printf("%d ", i);

}

printf("\n");

}

void multiples\_of\_3() {

pid\_t pid = getpid();

printf("I am the third child process (PID %d) and I will print the multiples of 3:\n", pid);

for (int i = 3; i <= 30; i += 3) {

printf("%d ", i);

}

printf("\n");

}

void multiples\_of\_5() {

pid\_t pid = getpid();

printf("I am the fourth child process (PID %d) and I will print the multiples of 5:\n", pid);

for (int i = 5; i <= 50; i += 5) {

printf("%d ", i);

}

printf("\n");

}

int main() {

pid\_t pid1, pid2, pid3, pid4;

pid1 = fork();

if (pid1 == 0) {

odd\_numbers();

} else {

pid2 = fork();

if (pid2 == 0) {

even\_numbers();

} else {

pid3 = fork();

if (pid3 == 0) {

multiples\_of\_3();

} else {

pid4 = fork();

if (pid4 == 0) {

multiples\_of\_5();

} else {

printf("I am the parent process (PID %d) and I have created 4 child processes (PID %d, %d, %d, %d).\n", getpid(), pid1, pid2, pid3, pid4);

}

}

}

}

return 0;

}

12. Write a C program to implement the best-fit algorithm and allocate the memory block to each process.

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 the best-fit algorithms to place processes of size 115 KB, 500 KB, 358 KB, 200 KB, and 375 KB (in order)

Program:-

#include <stdio.h>

#include <stdlib.h>

#define MAX\_BLOCKS 100

#define MAX\_PROCESS 100

int blocks[MAX\_BLOCKS];

int processes[MAX\_PROCESS];

int main() {

int num\_blocks, num\_processes, i, j;

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

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

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

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

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

}

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

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

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

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

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

}

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

int best\_fit = -1;

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

if (blocks[j] >= processes[i]) {

if (best\_fit == -1 || blocks[j] < blocks[best\_fit]) {

best\_fit = j;

}

}

}

if (best\_fit != -1) {

printf("Process %d allocated memory block %d of size %d\n", i, best\_fit, blocks[best\_fit]);

blocks[best\_fit] -= processes[i];

} else {

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

}

}

return 0;

}

OUTPUT:-

