**Experiment 1:**

**Aim:**

Write a program to simulate the following CPU Scheduling algorithms.

1. **FCFS**

**Code:**

#include <stdio.h>

void bubblesort(int at[], int bt[], int p[], int n) {

int i, j, temp;

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

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

if (at[j] > at[j + 1] || (at[j] == at[j + 1] && p[j] > p[j + 1])) {

temp = at[j];

at[j] = at[j + 1];

at[j + 1] = temp;

temp = bt[j];

bt[j] = bt[j + 1];

bt[j + 1] = temp;

temp = p[j];

p[j] = p[j + 1];

p[j + 1] = temp;

}

}

}

}

int main() {

int n, i;

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

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

scanf("%d", &n);

int p[n], at[n], bt[n], wt[n], tat[n], ct[n];

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

p[i] = i + 1;

printf("Enter Arrival Time for Process %d: ", i + 1);

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

printf("Enter Burst Time for Process %d: ", i + 1);

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

}

bubblesort(at, bt, p, n);

ct[0] = at[0] + bt[0];

tat[0] = ct[0] - at[0];

wt[0] = tat[0] - bt[0];

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

if (at[i] > ct[i - 1]) {

ct[i] = at[i] + bt[i];

} else {

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

}

tat[i] = ct[i] - at[i];

wt[i] = tat[i] - bt[i];

}

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

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

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

}

avg\_waiting\_time /= n;

avg\_turnaround\_time /= n;

printf("\nProcess\tArrival Time\tBurst Time\tCompletion Time\tWaiting Time\tTurnaround Time\n");

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

printf("%d\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n", p[i], at[i], bt[i], ct[i], wt[i], tat[i]);

}

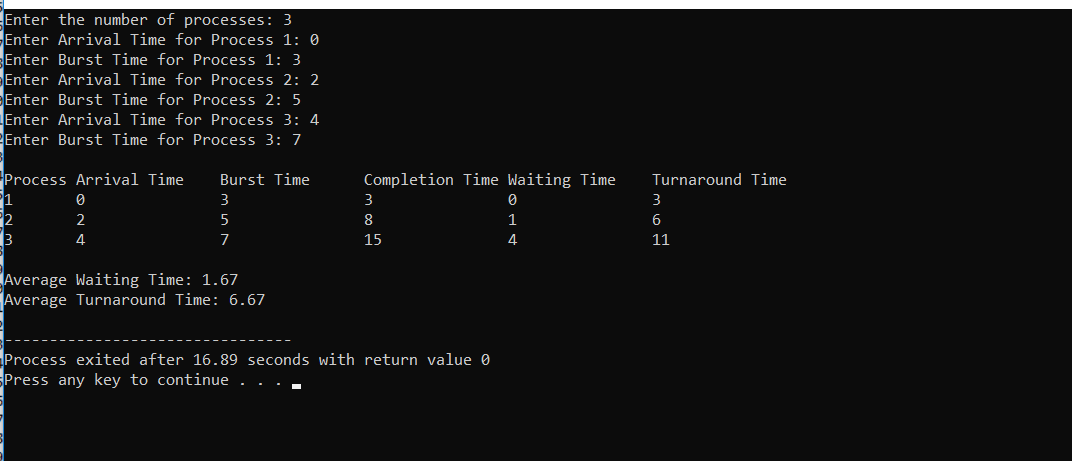
printf("\nAverage Waiting Time: %.2f", avg\_waiting\_time);

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

return 0;

}

**Output:**



**Aim:**

To write a program in C to simulate the **Shortest Job First** (SJF) CPU scheduling algorithm**.**

**Code:**

#include <stdio.h>

void swap(int \*a, int \*b) {

int temp = \*a;

\*a = \*b;

\*b = temp;

}

void sortProcesses(int pid[], int burstTime[], int arrivalTime[], int n) {

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

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

if (burstTime[j] > burstTime[j + 1] ||

(burstTime[j] == burstTime[j + 1] && arrivalTime[j] > arrivalTime[j + 1])) {

swap(&pid[j], &pid[j + 1]);

swap(&burstTime[j], &burstTime[j + 1]);

swap(&arrivalTime[j], &arrivalTime[j + 1]);

}

}

}

}

int main() {

int n;

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

scanf("%d", &n);

int pid[n], burstTime[n], arrivalTime[n], waitingTime[n], turnAroundTime[n];

int totalWaitingTime = 0, totalTurnAroundTime = 0;

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

pid[i] = i + 1;

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

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

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

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

}

sortProcesses(pid, burstTime, arrivalTime, n);

waitingTime[0] = 0;

int completionTime = 0;

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

if (completionTime < arrivalTime[i]) {

completionTime = arrivalTime[i];

}

waitingTime[i] = completionTime - arrivalTime[i];

if (waitingTime[i] < 0) {

waitingTime[i] = 0;

}

turnAroundTime[i] = waitingTime[i] + burstTime[i];

completionTime += burstTime[i];

totalWaitingTime += waitingTime[i];

totalTurnAroundTime += turnAroundTime[i];

}

printf("\nProcesses\tBurst Time\tArrival Time\tWaiting Time\tTurn-Around Time\n");

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

printf("%d\t\t%d\t\t%d\t\t%d\t\t%d\n", pid[i], burstTime[i], arrivalTime[i], waitingTime[i], turnAroundTime[i]);

}

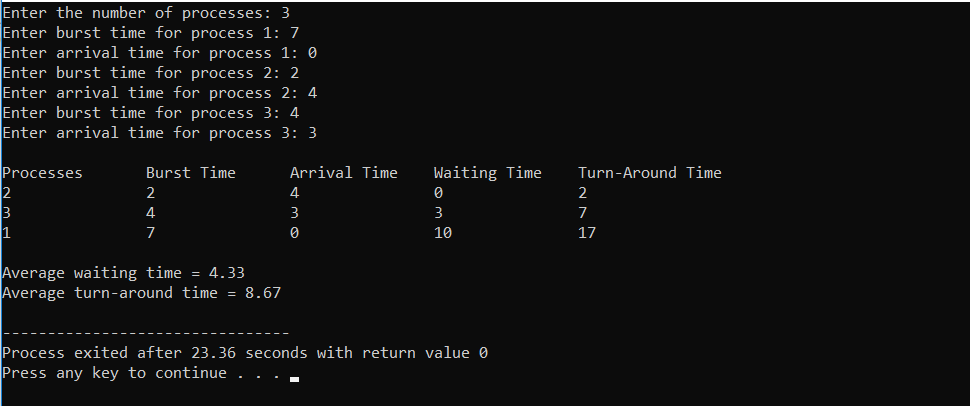
printf("\nAverage waiting time = %.2f\n", (float)totalWaitingTime / n);

printf("Average turn-around time = %.2f\n", (float)totalTurnAroundTime / n);

return 0;

}

**Output:**



**Aim:**

To write a program in C to simulate the **Round Robin** (RR) CPU scheduling algorithm.

**Code:**

#include <stdio.h>

#define MAX 100

int main() {

int n, timeQuantum,i;

int burstTime[MAX], arrivalTime[MAX], waitingTime[MAX], turnAroundTime[MAX], remainingBurstTime[MAX], completionTime[MAX];

int pid[MAX]; // Process IDs

int totalWaitingTime = 0, totalTurnAroundTime = 0, currentTime = 0, processesRemaining;

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

scanf("%d", &n);

printf("Enter the time quantum: ");

scanf("%d", &timeQuantum);

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

pid[i] = i + 1;

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

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

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

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

remainingBurstTime[i] = burstTime[i];

waitingTime[i] = 0;

completionTime[i] = 0;

}

processesRemaining = n;

while (processesRemaining > 0) {

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

if (arrivalTime[i] <= currentTime && remainingBurstTime[i] > 0) {

if (remainingBurstTime[i] > timeQuantum) {

currentTime += timeQuantum;

remainingBurstTime[i] -= timeQuantum;

} else {

currentTime += remainingBurstTime[i];

completionTime[i] = currentTime;

waitingTime[i] = currentTime - burstTime[i] - arrivalTime[i];

if (waitingTime[i] < 0) {

waitingTime[i] = 0;

}

turnAroundTime[i] = waitingTime[i] + burstTime[i];

remainingBurstTime[i] = 0;

processesRemaining--;

}

}

}

}

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

totalWaitingTime += waitingTime[i];

totalTurnAroundTime += turnAroundTime[i];

}

printf("\nProcesses\tBurst Time\tArrival Time\tCompletion Time\tWaiting Time\tTurn-Around Time\n");

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

printf("%d\t\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n", pid[i], burstTime[i], arrivalTime[i], completionTime[i], waitingTime[i], turnAroundTime[i]);

}

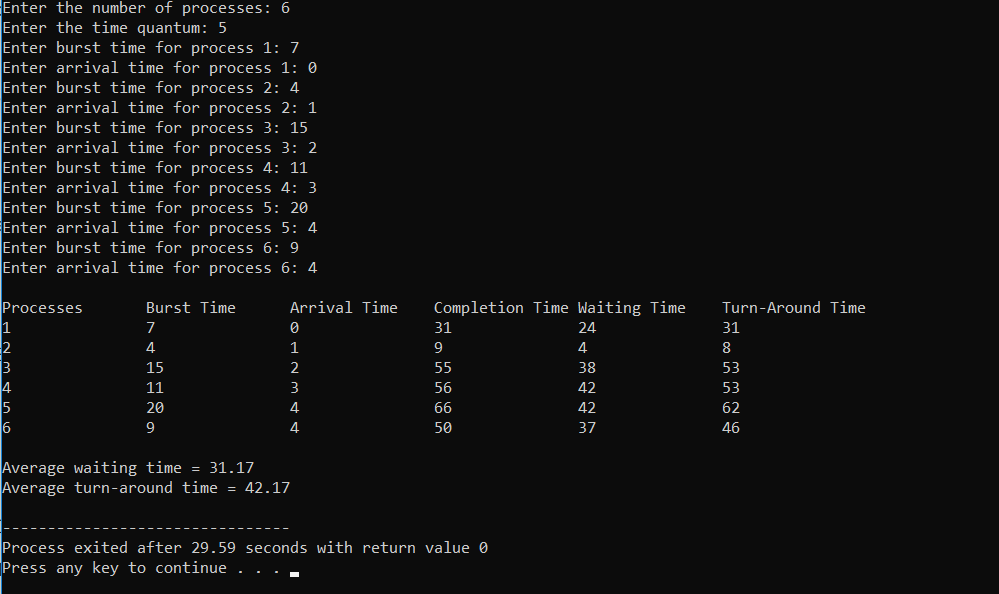
printf("\nAverage waiting time = %.2f\n", (float)totalWaitingTime / n);

printf("Average turn-around time = %.2f\n", (float)totalTurnAroundTime / n);

return 0;

}

**Output:**



**Aim:** To write a program in C to simulate the **Priority** CPU scheduling algorithm.

**Code:**

#include <stdio.h>

void calculateWaitingTime(int n, int arrivalTime[], int burstTime[], int waitingTime[]) {

int currentTime = 0;

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

if (currentTime < arrivalTime[i]) {

currentTime = arrivalTime[i];

}

waitingTime[i] = currentTime - arrivalTime[i];

if (waitingTime[i] < 0) {

waitingTime[i] = 0;

}

currentTime += burstTime[i];

}

}

void calculateTurnaroundTime(int n, int burstTime[], int waitingTime[], int turnaroundTime[]) {

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

turnaroundTime[i] = waitingTime[i] + burstTime[i];

}

}

void sortByPriorityAndArrival(int n, int process[], int arrivalTime[], int burstTime[], int priority[]) {

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

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

if (priority[i] > priority[j] || (priority[i] == priority[j] && arrivalTime[i] > arrivalTime[j])) {

int temp = priority[i];

priority[i] = priority[j];

priority[j] = temp;

temp = arrivalTime[i];

arrivalTime[i] = arrivalTime[j];

arrivalTime[j] = temp;

temp = burstTime[i];

burstTime[i] = burstTime[j];

burstTime[j] = temp;

temp = process[i];

process[i] = process[j];

process[j] = temp;

}

}

}

}

void displayResults(int n, int process[], int arrivalTime[], int burstTime[], int priority[], int waitingTime[], int turnaroundTime[]) {

printf("Process\tArrival Time\tPriority\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\t\t%d\n", process[i], arrivalTime[i], priority[i], burstTime[i], waitingTime[i], turnaroundTime[i]);

}

}

void displayAverages(int n, int waitingTime[], int turnaroundTime[]) {

double totalWaitingTime = 0, totalTurnaroundTime = 0;

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

totalWaitingTime += waitingTime[i];

totalTurnaroundTime += turnaroundTime[i];

}

printf("\nAverage Waiting Time: %.2lf\n", totalWaitingTime / n);

printf("Average Turnaround Time: %.2lf\n", totalTurnaroundTime / n);

}

int main() {

int n;

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

scanf("%d", &n);

int process[n], arrivalTime[n], burstTime[n], priority[n];

int waitingTime[n], turnaroundTime[n];

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

process[i] = i + 1;

printf("Enter arrival time for process P%d: ", process[i]);

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

printf("Enter burst time for process P%d: ", process[i]);

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

printf("Enter priority for process P%d: ", process[i]);

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

}

sortByPriorityAndArrival(n, process, arrivalTime, burstTime, priority);

calculateWaitingTime(n, arrivalTime, burstTime, waitingTime);

calculateTurnaroundTime(n, burstTime, waitingTime, turnaroundTime);

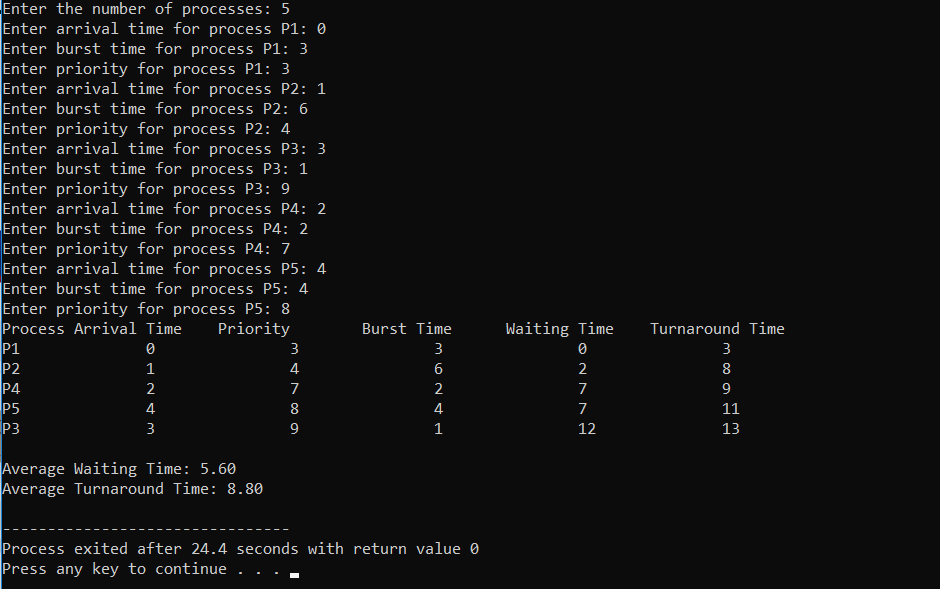
displayResults(n, process, arrivalTime, burstTime, priority, waitingTime, turnaroundTime);

displayAverages(n, waitingTime, turnaroundTime);

return 0;

}

**Output:**



**Experiment 2:**

**Aim:**

2a)Write a program to implement Process management system calls viz., fork, exit, wait, waitpid, exec.

**Code:**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/wait.h>

int main() {

    pid\_t pid = fork();

    if (pid < 0) {

        perror("Fork failed");

        exit(1);

    }

    if (pid == 0) {

        char \*args[] = {"./fc", NULL};

        execvp(args[0], args);

        perror("exec failed");

        exit(1);

    } else {

        printf("Parent Process: PID = %d, Child PID = %d\n", getpid(), pid);

        int status;

        wait(&status);

        printf("Parent Process: Child has finished with status %d\n", status);

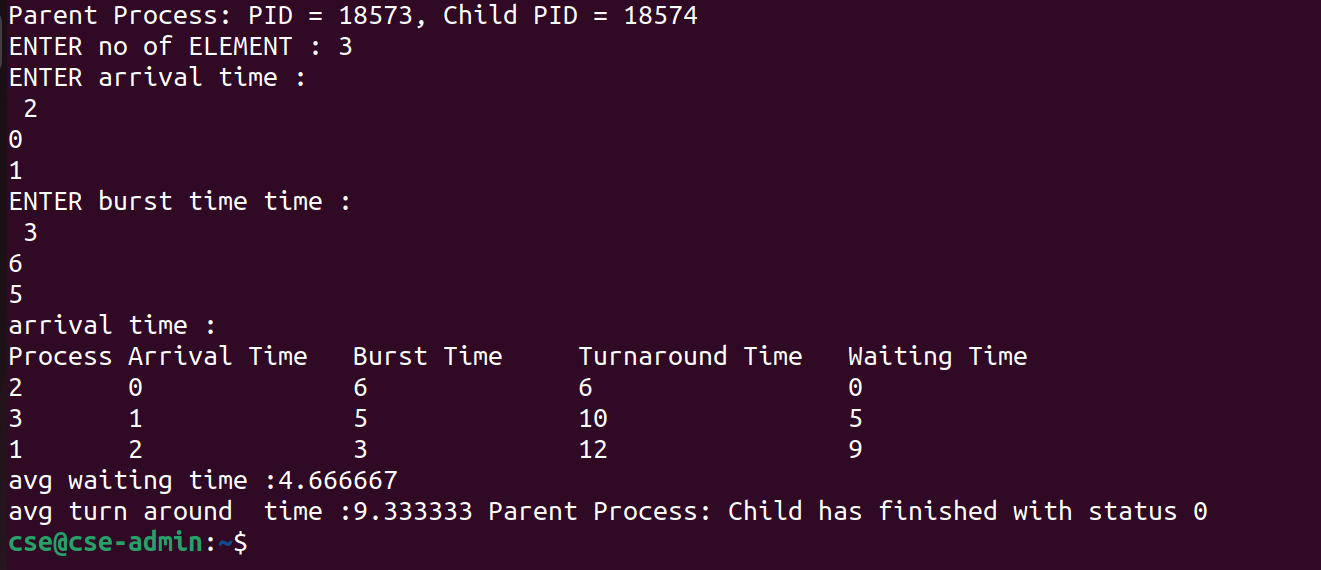
        exit(0);

    }

    return 0;

}

**Output:**

****

**Aim:**

2b) Write a program to implement I/O system calls viz., open, read, write, close, seek, stat,

opendir, readdir.

**Code:**

#include <stdio.h>

#include <fcntl.h>

#include <unistd.h>

#include <sys/stat.h>

#include <dirent.h>

#include <stdlib.h>

int main() {

int fd = open("example.txt", O\_CREAT | O\_RDWR, 0644);

if (fd < 0) {

perror("open");

return 1;

}

printf("File opened successfully.\n");

const char \*data = "Hello, this is a test file!\n";

ssize\_t bytes\_written = write(fd, data, strlen(data));

if (bytes\_written < 0) {

perror("write");

close(fd);

return 1;

}

printf("Data written to file.\n");

if (lseek(fd, 0, SEEK\_SET) < 0) {

perror("lseek");

close(fd);

return 1;

}

char buffer[256];

ssize\_t bytes\_read = read(fd, buffer, sizeof(buffer) - 1);

if (bytes\_read < 0) {

perror("read");

close(fd);

return 1;

}

buffer[bytes\_read] = '\0';

printf("Data read from file:\n%s\n", buffer);

struct stat file\_stat;

if (stat("example.txt", &file\_stat) < 0) {

perror("stat");

close(fd);

return 1;

}

printf("File size: %ld bytes\n", file\_stat.st\_size);

printf("File permissions: %o\n", file\_stat.st\_mode & 0777);

if (close(fd) < 0) {

perror("close");

return 1;

}

printf("File closed successfully.\n");

DIR \*dir = opendir(".");

if (!dir) {

perror("opendir");

return 1;

}

printf("Directory opened successfully.\n");

struct dirent \*entry;

printf("Directory entries:\n");

while ((entry = readdir(dir)) != NULL) {

printf("%s\n", entry->d\_name);

}

// Close the directory

if (closedir(dir) < 0) {

perror("closedir");

return 1;

}

printf("Directory closed successfully.\n");

return 0;

}

**Output:**

