**DAY-1**

1. Create a new process by invoking the appropriate system call. Get the process identifier of the currently running process and its respective parent using system calls and display the same using a C program.

PROGRAM

#include <windows.h>

#include <stdio.h>

void create\_process() {

STARTUPINFO si;

PROCESS\_INFORMATION pi;

char commandLine[] = "C:\\Windows\\System32\\notepad.exe"; // Example executable path

ZeroMemory(&si, sizeof(si));

si.cb = sizeof(si);

ZeroMemory(&pi, sizeof(pi));

// Create a new process

if (!CreateProcess(NULL, // No module name (use command line)

commandLine, // Command line

NULL, // Process handle not inheritable

NULL, // Thread handle not inheritable

FALSE, // Set handle inheritance to FALSE

0, // No creation flags

NULL, // Use parent's environment block

NULL, // Use parent's starting directory

&si, // Pointer to STARTUPINFO structure

&pi) // Pointer to PROCESS\_INFORMATION structure

)

{

printf("CreateProcess failed (%lu).\n", GetLastError());

return;

}

// Print parent and child PIDs

printf("Parent process ID: %lu\n", GetCurrentProcessId());

printf("Child process ID: %lu\n", pi.dwProcessId);

// Wait until child process exits

WaitForSingleObject(pi.hProcess, INFINITE);

// Close process and thread handles

CloseHandle(pi.hProcess);

CloseHandle(pi.hThread);

}

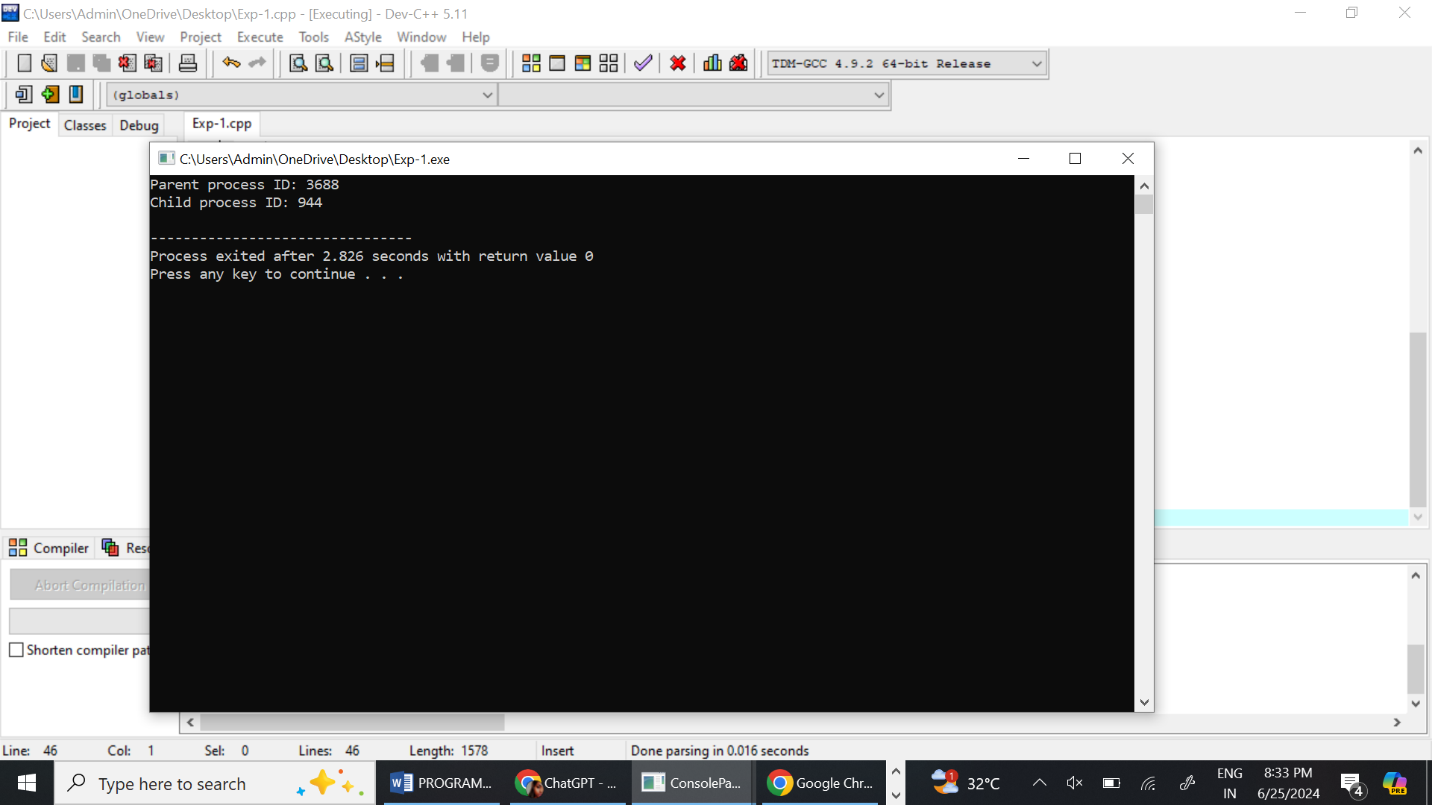
int main() {

create\_process();

return 0;

}

OUTPUT



1. Identify the system calls to copy the content of one file to another and illustrate the same using a C program.

PROGRAM

#include <fcntl.h>

#include <unistd.h>

#include <stdlib.h>

#include <stdio.h>

#define BUFFER\_SIZE 1024

void copy\_file(const char \*source, const char \*destination) {

int source\_fd, dest\_fd; // File descriptors

ssize\_t n\_read; // Number of bytes read

char buffer[BUFFER\_SIZE]; // Buffer to hold file data

// Open the source file

source\_fd = open(source, O\_RDONLY);

if (source\_fd == -1) {

perror("Error opening source file");

exit(EXIT\_FAILURE);

}

// Open the destination file (create it if it doesn't exist, truncate it if it does)

dest\_fd = open(destination, O\_WRONLY | O\_CREAT | O\_TRUNC, 0644);

if (dest\_fd == -1) {

perror("Error opening destination file");

close(source\_fd);

exit(EXIT\_FAILURE);

}

// Read from the source file and write to the destination file

while ((n\_read = read(source\_fd, buffer, BUFFER\_SIZE)) > 0) {

if (write(dest\_fd, buffer, n\_read) != n\_read) {

perror("Error writing to destination file");

close(source\_fd);

close(dest\_fd);

exit(EXIT\_FAILURE);

}

}

if (n\_read == -1) {

perror("Error reading from source file");

}

// Close both files

close(source\_fd);

close(dest\_fd);

}

int main(int argc, char \*argv[]) {

if (argc != 3) {

fprintf(stderr, "Usage: %s <source file> <destination file>\n", argv[0]);

exit(EXIT\_FAILURE);

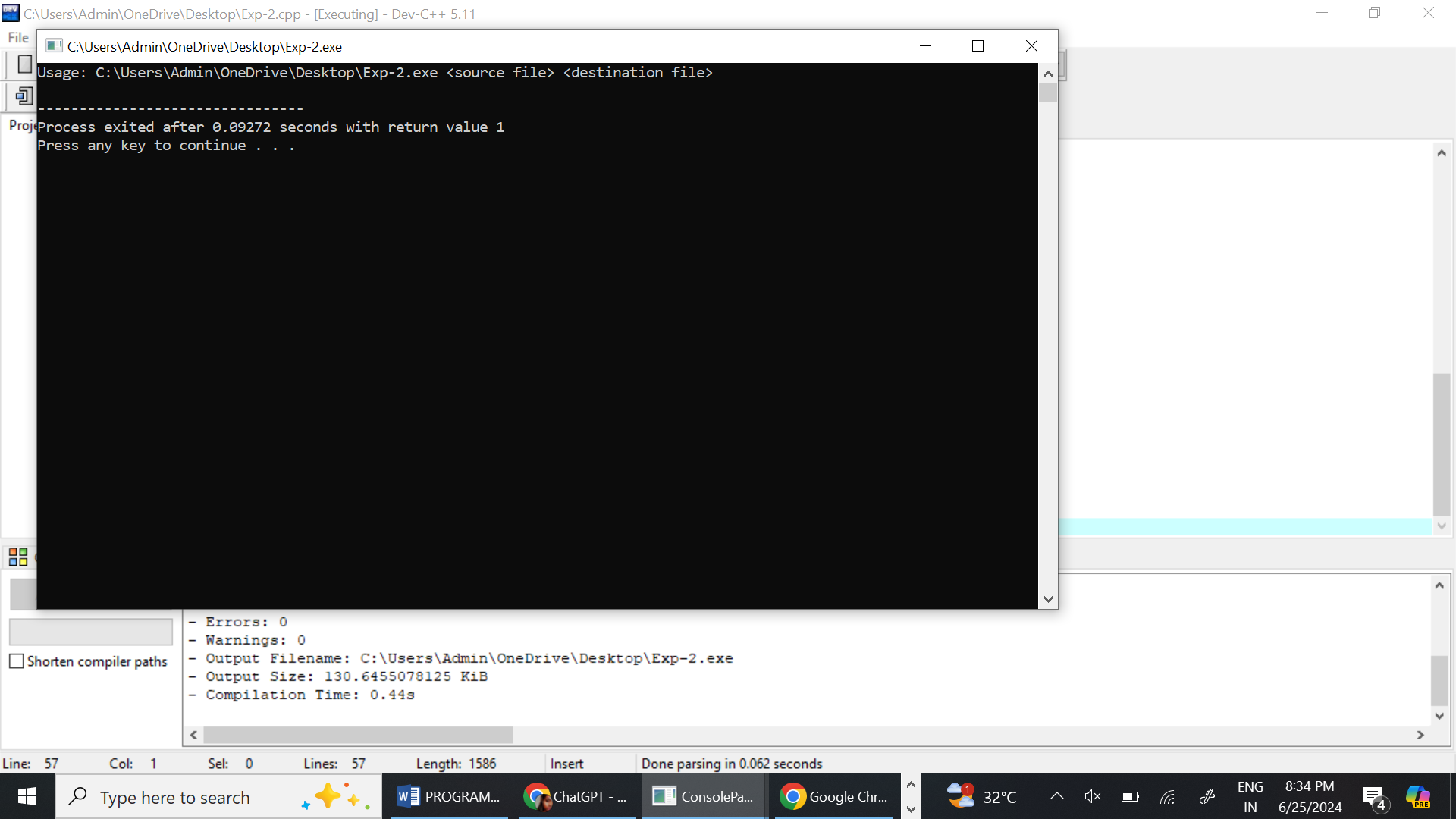
}

copy\_file(argv[1], argv[2]);

return 0;

}

OUTPUT



1. Design a CPU scheduling program with C using First Come First Served technique with the following considerations.
   1. All processes are activated at time 0.
   2. Assume that no process waits on I/O devices.

PROGRAM

#include <stdio.h>

// Define a structure to hold process information

struct Process {

int id; // Process ID

int burstTime; // Burst time of the process

int waitingTime; // Waiting time of the process

int turnaroundTime; // Turnaround time of the process

};

// Function to calculate waiting time for each process

void calculateWaitingTime(struct Process processes[], int n) {

processes[0].waitingTime = 0; // First process has no waiting time

// Calculate waiting time for each process

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

processes[i].waitingTime = processes[i - 1].waitingTime + processes[i - 1].burstTime;

}

}

// Function to calculate turnaround time for each process

void calculateTurnaroundTime(struct Process processes[], int n) {

// Calculate turnaround time for each process

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

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

}

}

// Function to calculate average waiting time and average turnaround time

void calculateAverageTimes(struct Process processes[], int n) {

int totalWaitingTime = 0;

int totalTurnaroundTime = 0;

// Calculate total waiting time and total turnaround time

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

totalWaitingTime += processes[i].waitingTime;

totalTurnaroundTime += processes[i].turnaroundTime;

}

// Calculate and print average waiting time and average turnaround time

float averageWaitingTime = (float) totalWaitingTime / n;

float averageTurnaroundTime = (float) totalTurnaroundTime / n;

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

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

}

// Function to print process information

void printProcessInfo(struct Process processes[], int n) {

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

// Print information for each process

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

printf("%d\t\t%d\t\t%d\t\t%d\n", processes[i].id, processes[i].burstTime,

processes[i].waitingTime, processes[i].turnaroundTime);

}

}

int main() {

int n;

// Ask the user for the number of processes

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

scanf("%d", &n);

struct Process processes[n];

// Get burst time for each process

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

processes[i].id = i + 1;

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

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

}

// Calculate waiting time and turnaround time for each process

calculateWaitingTime(processes, n);

calculateTurnaroundTime(processes, n);

// Print process information

printProcessInfo(processes, n);

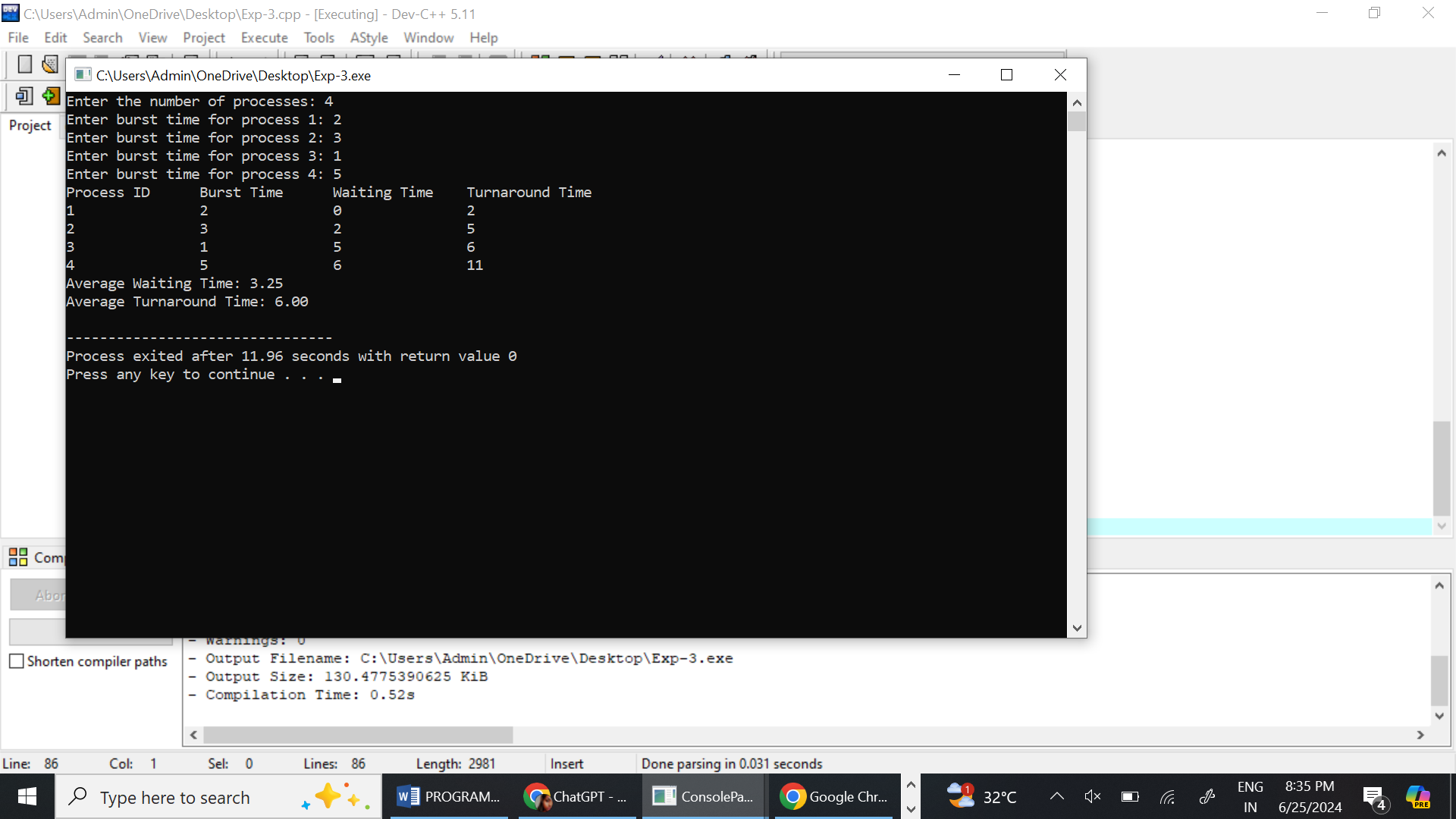
// Calculate and print average waiting time and average turnaround time

calculateAverageTimes(processes, n);

return 0;

}

OUTPUT



1. Construct a scheduling program with C that selects the waiting process with the smallest execution time to execute next.

PROGRAM

#include <stdio.h>

struct Process {

int pid; // Process ID

int burstTime; // Burst Time

int waitingTime; // Waiting Time

int turnaroundTime; // Turnaround Time

};

void calculateWaitingTime(struct Process processes[], int n) {

processes[0].waitingTime = 0; // Waiting time for first process is 0

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

processes[i].waitingTime = processes[i - 1].waitingTime + processes[i - 1].burstTime;

}

}

void calculateTurnaroundTime(struct Process processes[], int n) {

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

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

}

}

void sortProcessesByBurstTime(struct Process processes[], int n) {

struct Process temp;

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

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

if (processes[i].burstTime > processes[j].burstTime) {

temp = processes[i];

processes[i] = processes[j];

processes[j] = temp;

}

}

}

}

void printProcesses(struct Process processes[], int n) {

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

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

printf("%d\t\t%d\t\t%d\t\t%d\n", processes[i].pid, processes[i].burstTime, processes[i].waitingTime, processes[i].turnaroundTime);

}

}

int main() {

int n;

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

scanf("%d", &n);

struct Process processes[n];

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

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

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

processes[i].pid = i + 1;

}

// Sort processes by burst time

sortProcessesByBurstTime(processes, n);

// Calculate waiting time for all processes

calculateWaitingTime(processes, n);

// Calculate turnaround time for all processes

calculateTurnaroundTime(processes, n);

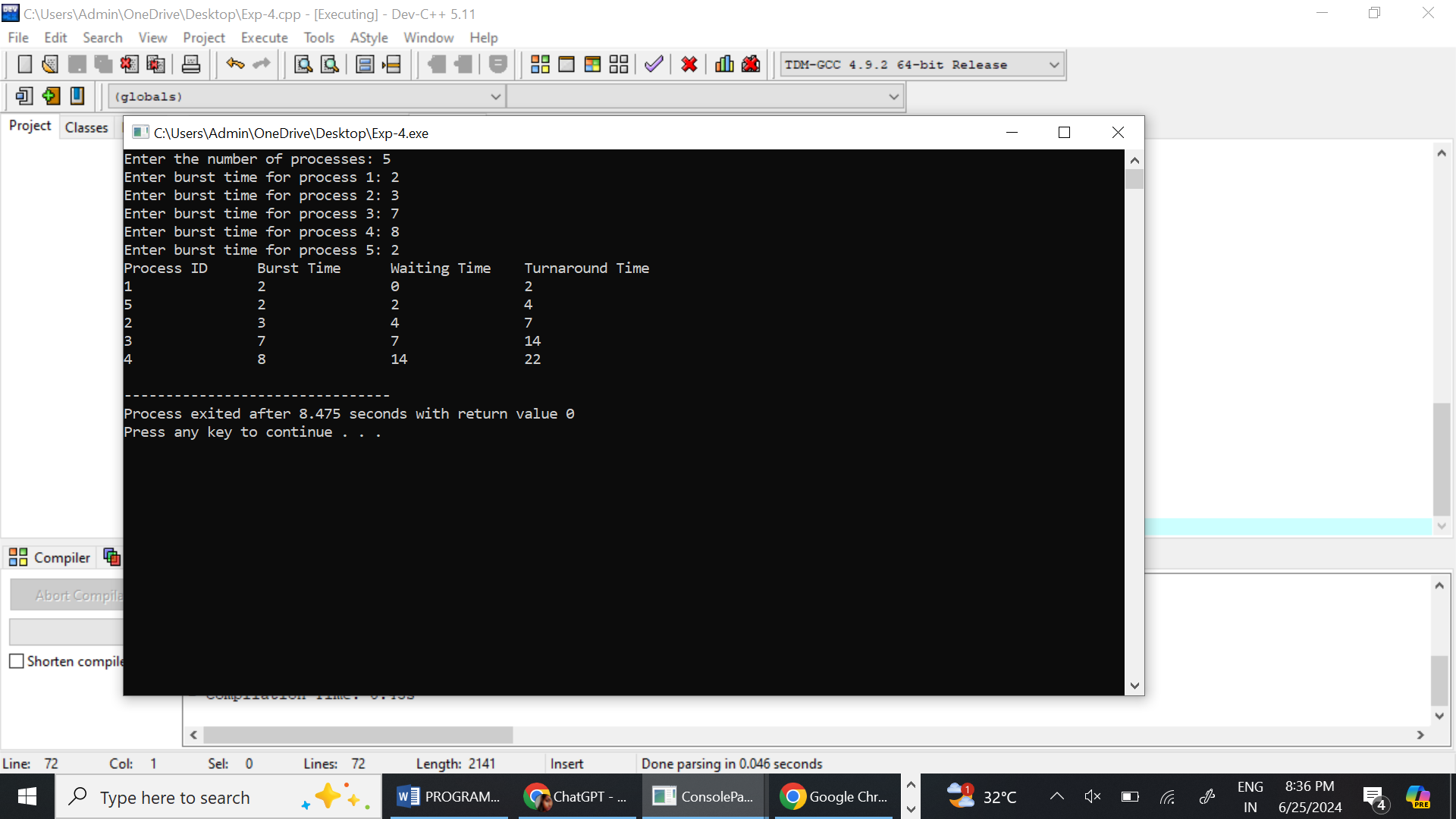
// Print all processes with their burst time, waiting time and turnaround time

printProcesses(processes, n);

return 0;

}

OUTPUT



1. Construct a scheduling program with C that selects the waiting process with the highest priority to execute next.

PROGRAM

#include <stdio.h>

#include <stdlib.h>

typedef struct {

int id;

int priority;

// Other fields like arrival\_time, burst\_time can be added as needed.

} Process;

#define MAX\_PROCESSES 100

typedef struct {

Process processes[MAX\_PROCESSES];

int size;

} PriorityQueue;

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

Process temp = \*a;

\*a = \*b;

\*b = temp;

}

void push(PriorityQueue \*pq, Process p) {

if (pq->size == MAX\_PROCESSES) {

printf("Priority queue is full\n");

return;

}

pq->processes[pq->size] = p;

int i = pq->size;

pq->size++;

while (i > 0 && pq->processes[i].priority > pq->processes[(i - 1) / 2].priority) {

swap(&pq->processes[i], &pq->processes[(i - 1) / 2]);

i = (i - 1) / 2;

}

}

Process pop(PriorityQueue \*pq) {

if (pq->size == 0) {

printf("Priority queue is empty\n");

exit(1);

}

Process top = pq->processes[0];

pq->processes[0] = pq->processes[pq->size - 1];

pq->size--;

int i = 0;

while ((2 \* i + 1) < pq->size) {

int maxChild = 2 \* i + 1;

if ((2 \* i + 2) < pq->size && pq->processes[2 \* i + 2].priority > pq->processes[2 \* i + 1].priority) {

maxChild = 2 \* i + 2;

}

if (pq->processes[i].priority >= pq->processes[maxChild].priority) {

break;

}

swap(&pq->processes[i], &pq->processes[maxChild]);

i = maxChild;

}

return top;

}

void initializePriorityQueue(PriorityQueue \*pq) {

pq->size = 0;

}

void schedule(PriorityQueue \*pq) {

while (pq->size > 0) {

Process p = pop(pq);

printf("Executing process ID %d with priority %d\n", p.id, p.priority);

// Simulate process execution with a sleep or just print statement.

// In a real system, you'd have more complex logic here.

}

}

int main() {

PriorityQueue pq;

initializePriorityQueue(&pq);

// Example processes

Process p1 = {1, 3};

Process p2 = {2, 1};

Process p3 = {3, 2};

Process p4 = {4, 4};

push(&pq, p1);

push(&pq, p2);

push(&pq, p3);

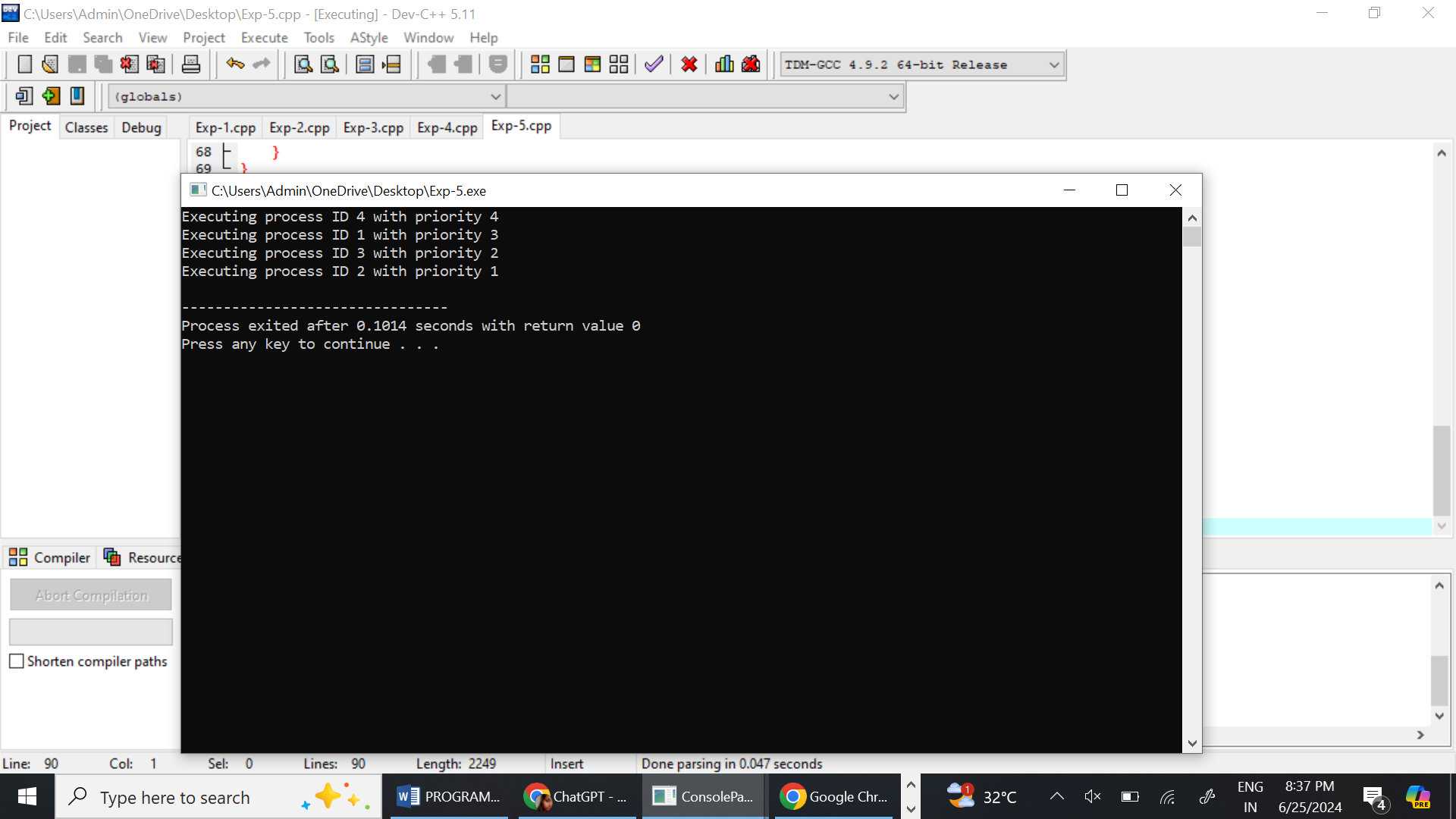
push(&pq, p4);

schedule(&pq);

return 0;

}

OUTPUT



1. Construct a C program to implement pre-emptive priority scheduling algorithm.

PROGRAM

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#define MAX 100

typedef struct {

int id;

int burstTime;

int remainingTime;

int priority;

int arrivalTime;

int waitingTime;

int turnaroundTime;

bool isCompleted;

} Process;

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

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

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

if(p[j].arrivalTime > p[j+1].arrivalTime) {

Process temp = p[j];

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

p[j+1] = temp;

}

}

}

}

int main() {

int n, currentTime = 0, completed = 0, prev = -1;

Process p[MAX];

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

scanf("%d", &n);

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

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

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

p[i].id = i+1;

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

p[i].waitingTime = 0;

p[i].turnaroundTime = 0;

p[i].isCompleted = false;

}

sortByArrivalTime(p, n);

while(completed != n) {

int idx = -1;

int minPriority = \_\_INT\_MAX\_\_;

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

if(p[i].arrivalTime <= currentTime && !p[i].isCompleted) {

if(p[i].priority < minPriority) {

minPriority = p[i].priority;

idx = i;

}

if(p[i].priority == minPriority) {

if(p[i].arrivalTime < p[idx].arrivalTime) {

idx = i;

}

}

}

}

if(idx != -1) {

if(prev != idx) {

printf("At time %d, process %d is selected for execution\n", currentTime, p[idx].id);

prev = idx;

}

p[idx].remainingTime--;

currentTime++;

if(p[idx].remainingTime == 0) {

p[idx].isCompleted = true;

completed++;

p[idx].turnaroundTime = currentTime - p[idx].arrivalTime;

p[idx].waitingTime = p[idx].turnaroundTime - p[idx].burstTime;

printf("Process %d completed at time %d\n", p[idx].id, currentTime);

}

} else {

currentTime++;

}

}

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

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

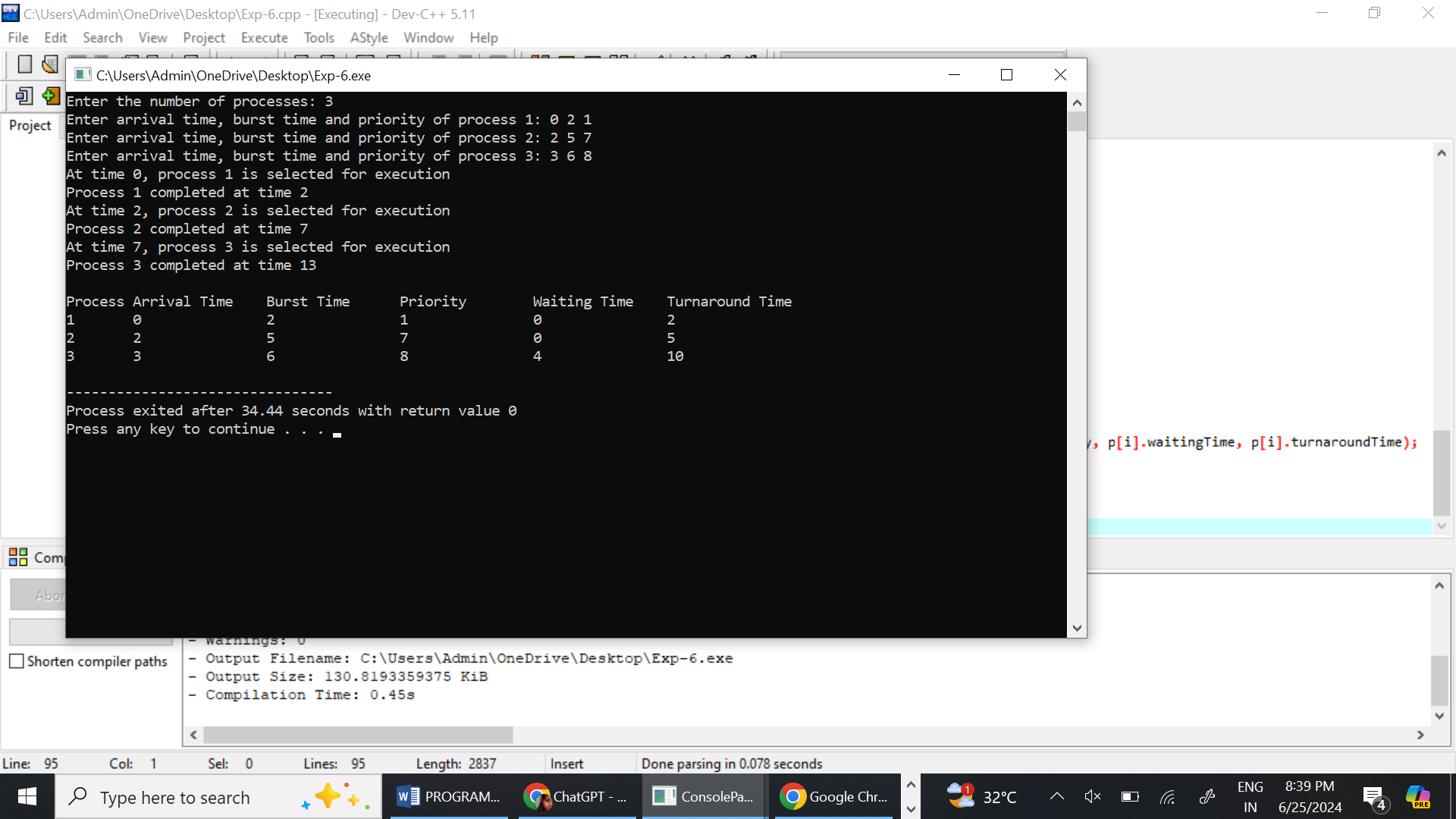
printf("%d\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n", p[i].id, p[i].arrivalTime, p[i].burstTime, p[i].priority, p[i].waitingTime, p[i].turnaroundTime);

}

return 0;

}

OUTPUT



1. Construct a C program to implement non-preemptive SJF algorithm.

PROGRAM

#include <stdio.h>

struct Process {

int id;

int burstTime;

int waitingTime;

int turnaroundTime;

};

void calculateWaitingTime(struct Process proc[], int n) {

proc[0].waitingTime = 0;

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

proc[i].waitingTime = proc[i - 1].waitingTime + proc[i - 1].burstTime;

}

}

void calculateTurnaroundTime(struct Process proc[], int n) {

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

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

}

}

void sortProcessesByBurstTime(struct Process proc[], int n) {

struct Process temp;

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

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

if (proc[j].burstTime > proc[j + 1].burstTime) {

temp = proc[j];

proc[j] = proc[j + 1];

proc[j + 1] = temp;

}

}

}

}

void printGanttChart(struct Process proc[], int n) {

printf("\nGantt Chart:\n ");

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

printf("| P%d ", proc[i].id);

}

printf("|\n");

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

printf(" %d ", proc[i].waitingTime);

}

printf("%d\n", proc[n - 1].waitingTime + proc[n - 1].burstTime);

}

int main() {

int n;

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

scanf("%d", &n);

struct Process proc[n];

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

proc[i].id = i + 1;

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

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

}

sortProcessesByBurstTime(proc, n);

calculateWaitingTime(proc, n);

calculateTurnaroundTime(proc, n);

printf("\nProcess ID\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\n", proc[i].id, proc[i].burstTime, proc[i].waitingTime, proc[i].turnaroundTime);

}

float totalWaitingTime = 0, totalTurnaroundTime = 0;

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

totalWaitingTime += proc[i].waitingTime;

totalTurnaroundTime += proc[i].turnaroundTime;

}

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

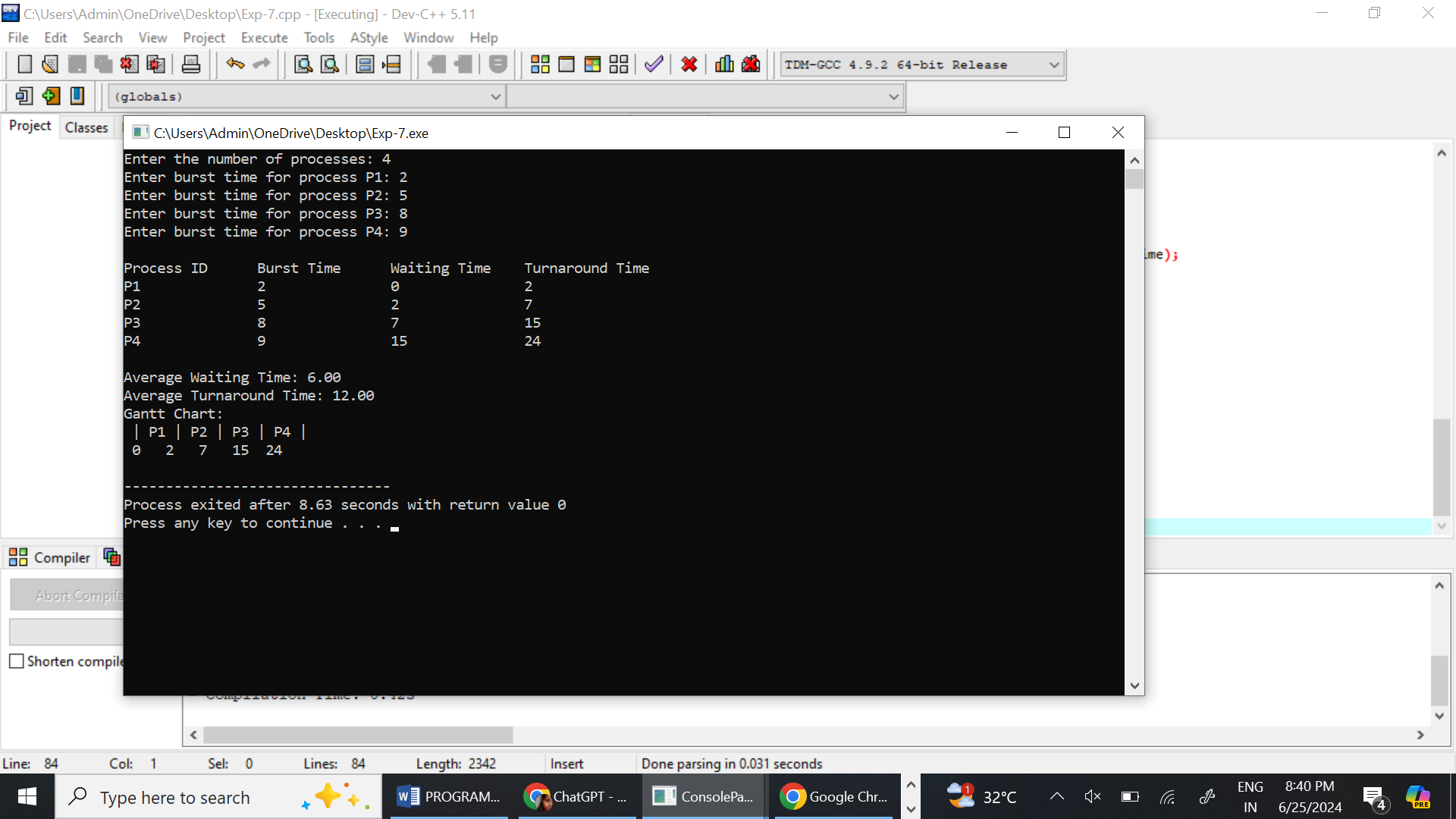
printf("\nAverage Turnaround Time: %.2f", totalTurnaroundTime / n);

printGanttChart(proc, n);

return 0;

}

OUTPUT



1. Construct a C program to simulate Round Robin scheduling algorithm with C.

PROGRAM

#include <stdio.h>

// Structure to represent a process

typedef struct {

int pid; // Process ID

int burst\_time; // Burst Time of the process

int remaining\_time; // Remaining Time of the process

} Process;

// Function to perform Round Robin scheduling

void roundRobinScheduling(Process processes[], int n, int time\_quantum) {

int total\_time = 0; // Total time passed

int complete = 0; // Number of processes completed

// Loop until all processes are completed

while (complete < n) {

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

// If process has remaining time greater than 0

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

// Check if remaining time is less than or equal to time quantum

if (processes[i].remaining\_time <= time\_quantum) {

total\_time += processes[i].remaining\_time;

printf("Process %d executed for %d units of time.\n", processes[i].pid, processes[i].remaining\_time);

processes[i].remaining\_time = 0; // Process is completed

complete++; // Increment the number of completed processes

} else {

// Process runs for the time quantum

total\_time += time\_quantum;

processes[i].remaining\_time -= time\_quantum;

printf("Process %d executed for %d units of time.\n", processes[i].pid, time\_quantum);

}

}

}

}

printf("Total time taken for all processes to complete: %d units\n", total\_time);

}

int main() {

int n; // Number of processes

int time\_quantum; // Time Quantum

// Get the number of processes

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

scanf("%d", &n);

Process processes[n];

// Get the burst time of each process

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

processes[i].pid = i + 1;

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

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

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

}

// Get the time quantum

printf("Enter the time quantum: ");

scanf("%d", &time\_quantum);

// Perform Round Robin Scheduling

roundRobinScheduling(processes, n, time\_quantum);

return 0;

}

OUTPUT

