**DAY-2**

9.Illustrate the concept of inter-process communication using shared memory with a C program.

PROGRAM

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#include <unistd.h>

#define SHM\_SIZE 1024 // Size of the shared memory segment

int main() {

// Create a key for the shared memory segment

key\_t key = ftok("shmfile", 65);

// Create the shared memory segment

int shmid = shmget(key, SHM\_SIZE, 0666 | IPC\_CREAT);

if (shmid == -1) {

perror("shmget failed");

exit(1);

}

// Fork the process to create a child process

pid\_t pid = fork();

if (pid < 0) {

perror("fork failed");

exit(1);

} else if (pid == 0) {

// Child process

// Attach to the shared memory segment

char \*shared\_memory = (char \*)shmat(shmid, NULL, 0);

if (shared\_memory == (char \*)(-1)) {

perror("shmat failed");

exit(1);

}

// Read data from shared memory

printf("Child process read: %s\n", shared\_memory);

// Detach from the shared memory segment

if (shmdt(shared\_memory) == -1) {

perror("shmdt failed");

exit(1);

}

// Exit the child process

exit(0);

} else {

// Parent process

// Attach to the shared memory segment

char \*shared\_memory = (char \*)shmat(shmid, NULL, 0);

if (shared\_memory == (char \*)(-1)) {

perror("shmat failed");

exit(1);

}

// Write data to shared memory

const char \*message = "Hello from parent process!";

strncpy(shared\_memory, message, SHM\_SIZE);

// Wait for the child process to complete

wait(NULL);

// Detach from the shared memory segment

if (shmdt(shared\_memory) == -1) {

perror("shmdt failed");

exit(1);

}

// Remove the shared memory segment

if (shmctl(shmid, IPC\_RMID, NULL) == -1) {

perror("shmctl failed");

exit(1);

}

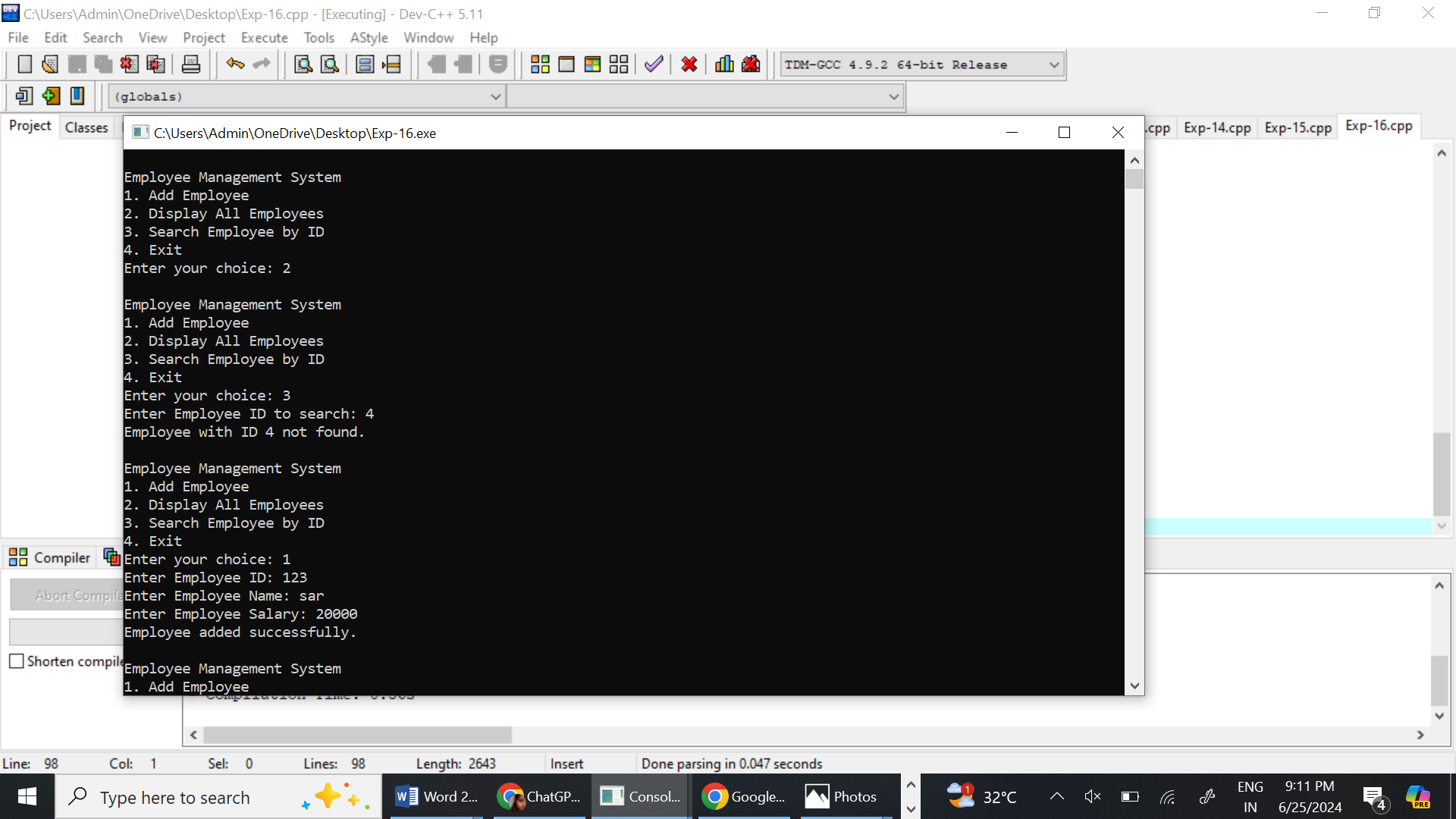
printf("Parent process completed.\n");

}

return 0;

}

OUTPUT



10. Illustrate the concept of inter-process communication using message queue with a C program.

PROGRAM

#include <windows.h>

#include <iostream>

#include <string>

int main() {

HANDLE hPipe;

LPCTSTR pipeName = TEXT("\\\\.\\pipe\\MyPipe");

// Try to open a named pipe

hPipe = CreateFile(

pipeName, // pipe name

GENERIC\_WRITE, // write access

0, // no sharing

NULL, // default security attributes

OPEN\_EXISTING, // opens existing pipe

0, // default attributes

NULL); // no template file

if (hPipe == INVALID\_HANDLE\_VALUE) {

std::cerr << "Failed to connect to pipe." << std::endl;

return 1;

}

std::string message;

std::cout << "Enter a message to send: ";

std::getline(std::cin, message);

// Write to the pipe

DWORD bytesWritten;

if (!WriteFile(hPipe, message.c\_str(), message.size() + 1, &bytesWritten, NULL)) {

std::cerr << "Failed to write to pipe." << std::endl;

CloseHandle(hPipe);

return 1;

}

std::cout << "Message sent: " << message << std::endl;

CloseHandle(hPipe);

return 0;

}

#include <windows.h>

#include <iostream>

int main() {

HANDLE hPipe;

LPCTSTR pipeName = TEXT("\\\\.\\pipe\\MyPipe");

// Create a named pipe

hPipe = CreateNamedPipe(

pipeName, // pipe name

PIPE\_ACCESS\_INBOUND, // read access

PIPE\_TYPE\_MESSAGE | PIPE\_READMODE\_MESSAGE | PIPE\_WAIT, // message type pipe, message-read mode, blocking mode

PIPE\_UNLIMITED\_INSTANCES, // max instances

512, // output buffer size

512, // input buffer size

0, // client time-out

NULL); // default security attributes

if (hPipe == INVALID\_HANDLE\_VALUE) {

std::cerr << "Failed to create pipe." << std::endl;

return 1;

}

std::cout << "Waiting for client to connect..." << std::endl;

// Wait for the client to connect

if (!ConnectNamedPipe(hPipe, NULL)) {

std::cerr << "Failed to connect to client." << std::endl;

CloseHandle(hPipe);

return 1;

}

char buffer[512];

DWORD bytesRead;

if (!ReadFile(hPipe, buffer, sizeof(buffer), &bytesRead, NULL)) {

std::cerr << "Failed to read from pipe." << std::endl;

CloseHandle(hPipe);

return 1;

}

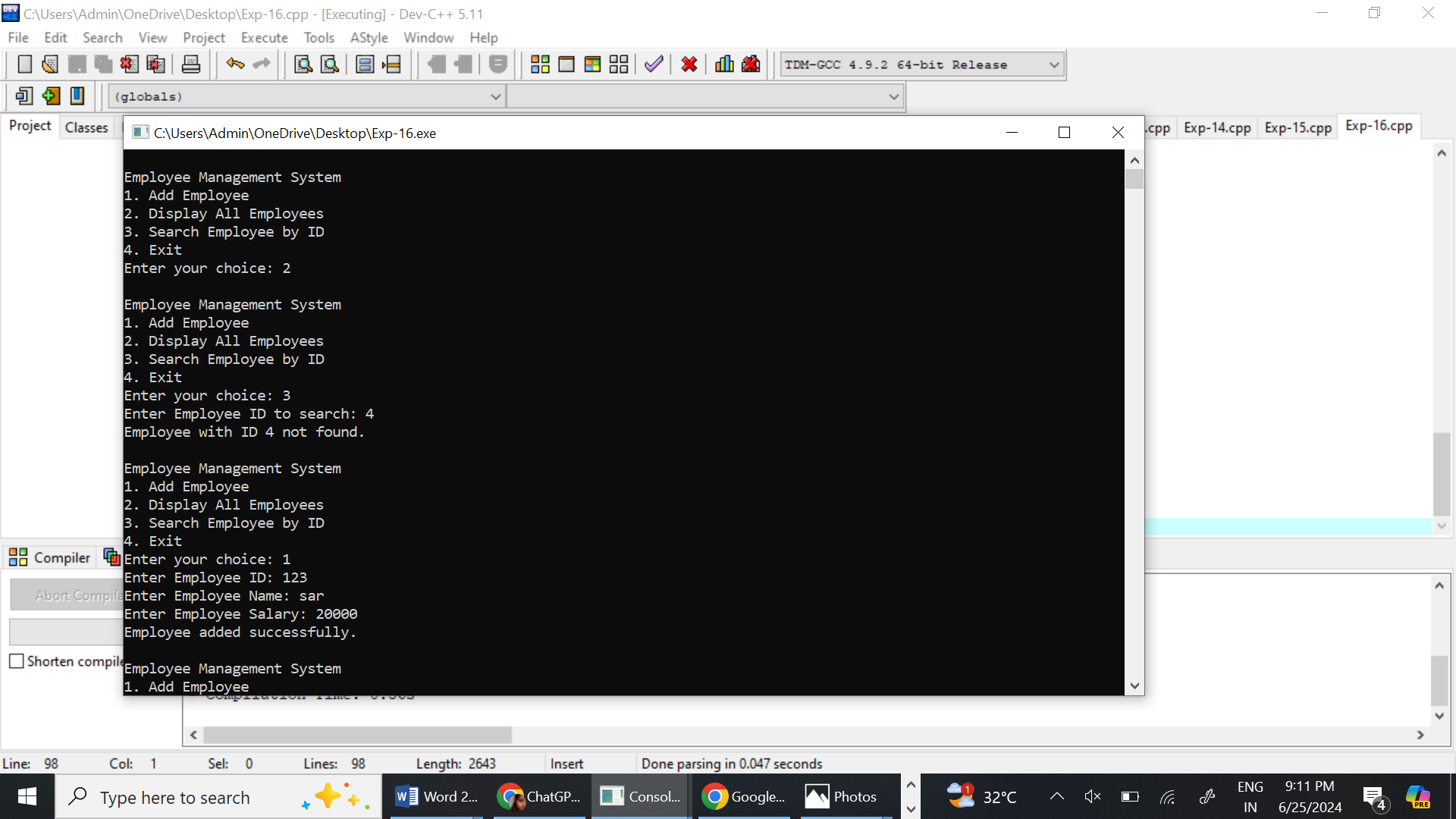
std::cout << "Message received: " << buffer << std::endl;

CloseHandle(hPipe);

return 0;

}

OUTPUT



11.Illustrate the concept of multithreading using a C program.

PROGRAM

#include <pthread.h>

#include <stdio.h>

#include <stdlib.h>

// Function to be executed by the threads

void \*print\_message(void \*threadid) {

long tid;

tid = (long)threadid;

printf("Hello from thread #%ld!\n", tid);

pthread\_exit(NULL);

}

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

pthread\_t threads[2];

int rc;

long t;

// Create two threads

for(t = 0; t < 2; t++) {

printf("In main: creating thread %ld\n", t);

rc = pthread\_create(&threads[t], NULL, print\_message, (void \*)t);

if (rc) {

printf("ERROR; return code from pthread\_create() is %d\n", rc);

exit(-1);

}

}

// Wait for the threads to complete

for(t = 0; t < 2; t++) {

pthread\_join(threads[t], NULL);

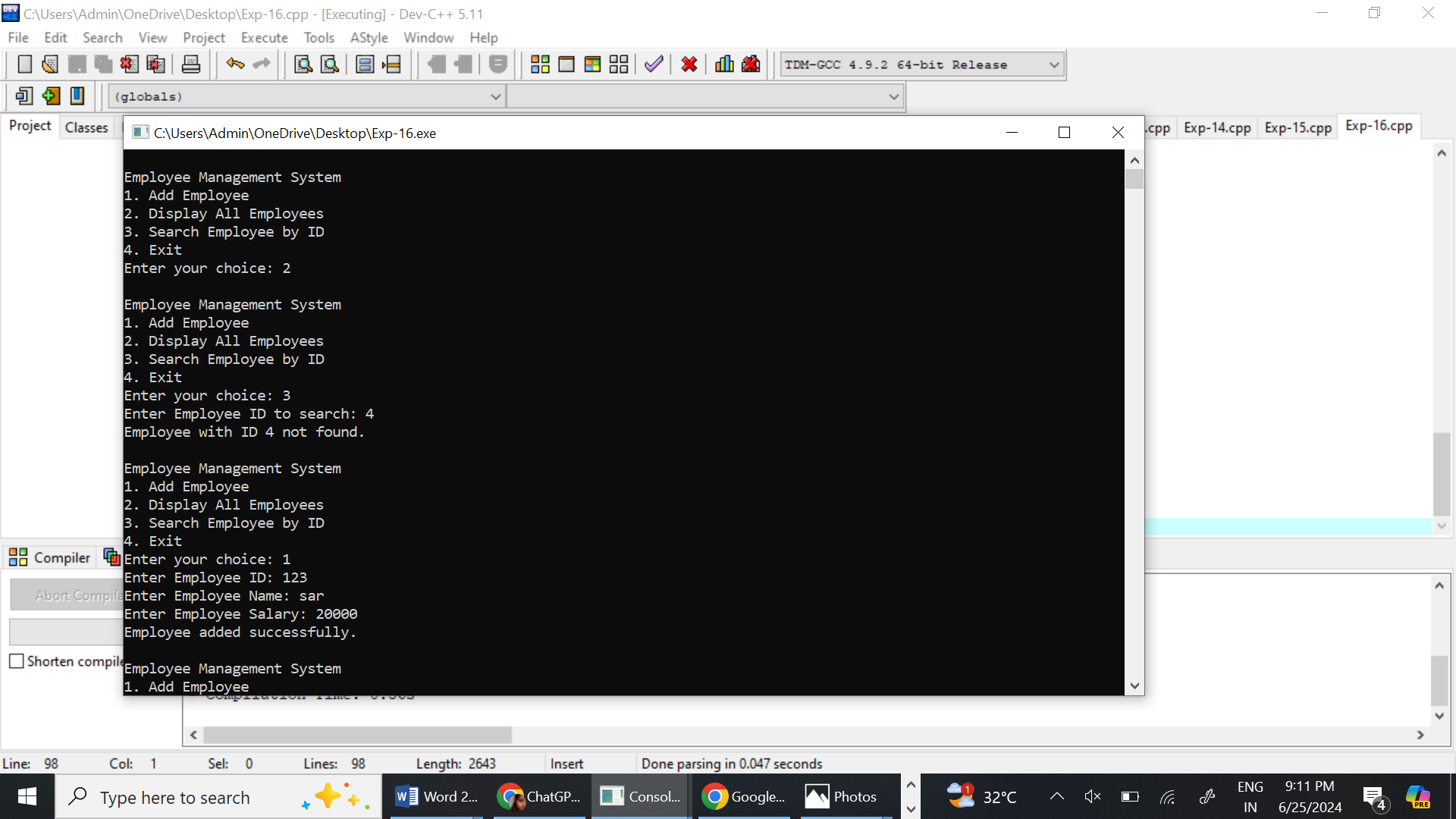
}

printf("Main: program exiting.\n");

pthread\_exit(NULL);

}

OUTPUT



12. Design a C program to simulate the concept of Dining-Philosophers problem

#include <pthread.h>

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#define NUM\_PHILOSOPHERS 5

pthread\_mutex\_t forks[NUM\_PHILOSOPHERS];

pthread\_t philosophers[NUM\_PHILOSOPHERS];

void \*philosopher(void \*num) {

int id = \*(int \*)num;

while (1) {

printf("Philosopher %d is thinking.\n", id);

usleep(rand() % 1000000);

printf("Philosopher %d is hungry.\n", id);

pthread\_mutex\_lock(&forks[id]);

pthread\_mutex\_lock(&forks[(id + 1) % NUM\_PHILOSOPHERS]);

printf("Philosopher %d is eating.\n", id);

usleep(rand() % 1000000);

pthread\_mutex\_unlock(&forks[id]);

pthread\_mutex\_unlock(&forks[(id + 1) % NUM\_PHILOSOPHERS]);

printf("Philosopher %d is done eating.\n", id);

}

return NULL;

}

int main() {

int i;

int ids[NUM\_PHILOSOPHERS];

// Initialize mutexes

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

pthread\_mutex\_init(&forks[i], NULL);

}

// Create philosopher threads

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

ids[i] = i;

pthread\_create(&philosophers[i], NULL, philosopher, &ids[i]);

}

// Join philosopher threads

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

pthread\_join(philosophers[i], NULL);

}

// Destroy mutexes

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

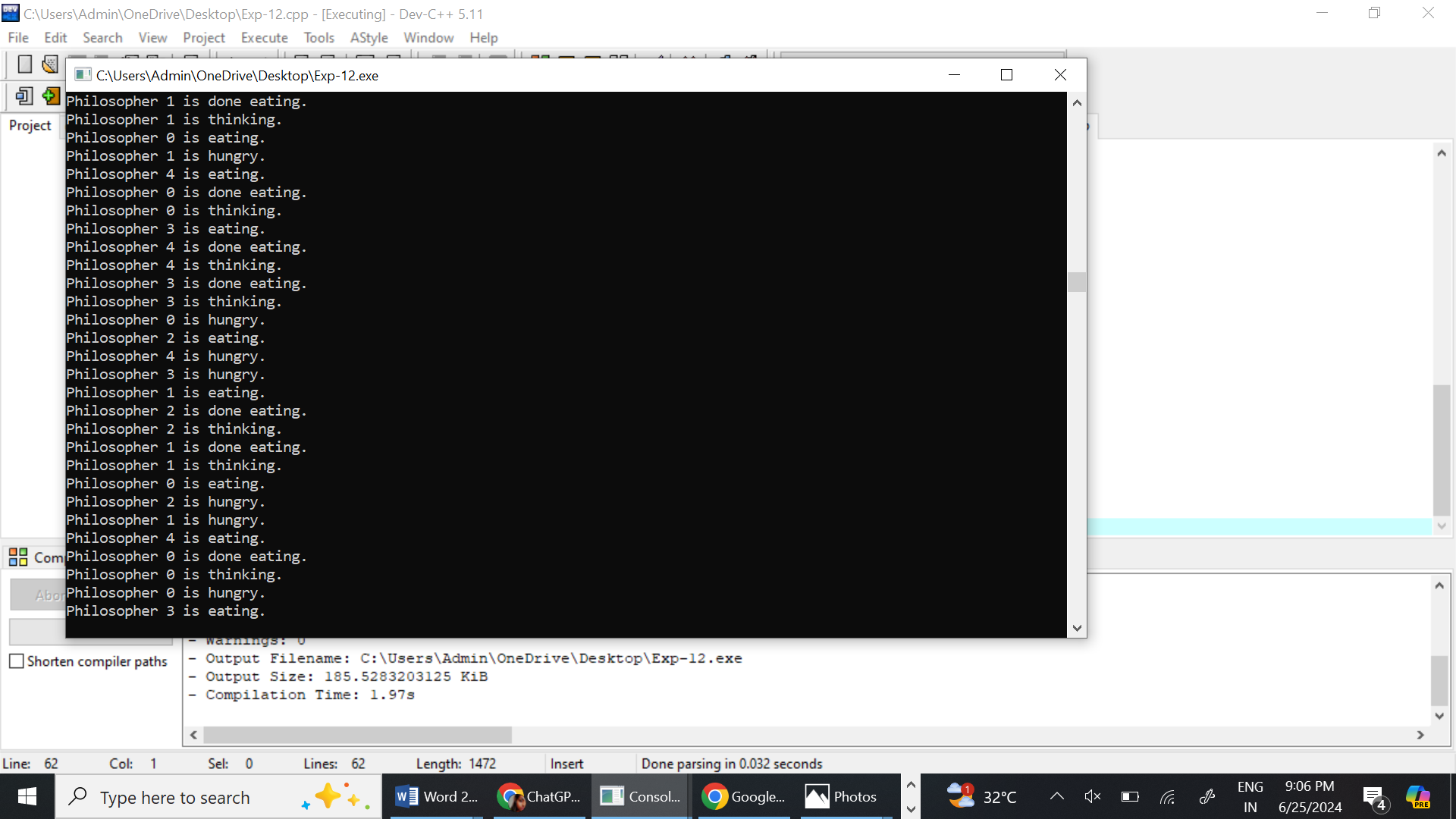
pthread\_mutex\_destroy(&forks[i]);

}

return 0;

}

OUTPUT



13. Construct a C program for implementation the various memory allocation strategies

PROGRAM

#include <stdio.h>

#include <stdlib.h>

#define MAX\_BLOCKS 100

typedef struct {

int size;

int is\_allocated;

} MemoryBlock;

void first\_fit(MemoryBlock blocks[], int n, int process\_size) {

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

if (!blocks[i].is\_allocated && blocks[i].size >= process\_size) {

blocks[i].is\_allocated = 1;

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

return;

}

}

printf("No suitable block found for the process.\n");

}

void best\_fit(MemoryBlock blocks[], int n, int process\_size) {

int best\_idx = -1;

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

if (!blocks[i].is\_allocated && blocks[i].size >= process\_size) {

if (best\_idx == -1 || blocks[i].size < blocks[best\_idx].size) {

best\_idx = i;

}

}

}

if (best\_idx != -1) {

blocks[best\_idx].is\_allocated = 1;

printf("Process allocated in block %d\n", best\_idx + 1);

} else {

printf("No suitable block found for the process.\n");

}

}

void worst\_fit(MemoryBlock blocks[], int n, int process\_size) {

int worst\_idx = -1;

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

if (!blocks[i].is\_allocated && blocks[i].size >= process\_size) {

if (worst\_idx == -1 || blocks[i].size > blocks[worst\_idx].size) {

worst\_idx = i;

}

}

}

if (worst\_idx != -1) {

blocks[worst\_idx].is\_allocated = 1;

printf("Process allocated in block %d\n", worst\_idx + 1);

} else {

printf("No suitable block found for the process.\n");

}

}

void print\_memory\_blocks(MemoryBlock blocks[], int n) {

printf("Memory Blocks:\n");

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

printf("Block %d: Size = %d, %s\n", i + 1, blocks[i].size, blocks[i].is\_allocated ? "Allocated" : "Free");

}

}

int main() {

int n;

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

scanf("%d", &n);

MemoryBlock blocks[MAX\_BLOCKS];

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

printf("Enter size of block %d: ", i + 1);

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

blocks[i].is\_allocated = 0;

}

int choice, process\_size;

while (1) {

printf("\nMemory Allocation Strategies:\n");

printf("1. First Fit\n2. Best Fit\n3. Worst Fit\n4. Print Memory Blocks\n5. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

if (choice == 5) {

break;

}

switch (choice) {

case 1:

printf("Enter the size of the process: ");

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

first\_fit(blocks, n, process\_size);

break;

case 2:

printf("Enter the size of the process: ");

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

best\_fit(blocks, n, process\_size);

break;

case 3:

printf("Enter the size of the process: ");

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

worst\_fit(blocks, n, process\_size);

break;

case 4:

print\_memory\_blocks(blocks, n);

break;

default:

printf("Invalid choice. Please try again.\n");

break;

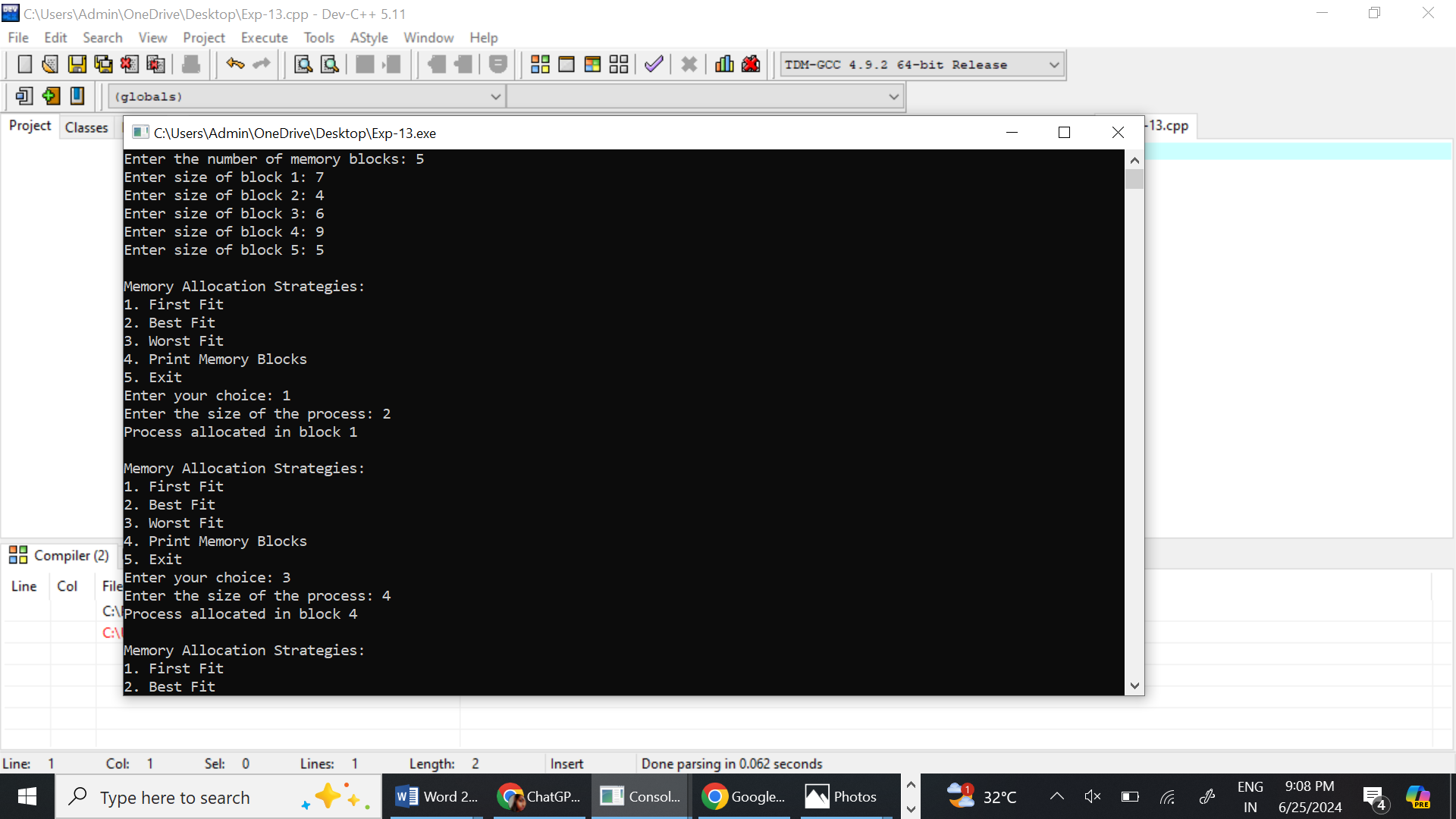
}

}

return 0;

}

OUTPUT



14. Construct a C program to organize the file using single level directory.

PROGRAM

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

// Define maximum number of files and file name length

#define MAX\_FILES 100

#define MAX\_FILENAME\_LENGTH 100

// Structure to represent a file

typedef struct {

char name[MAX\_FILENAME\_LENGTH];

} File;

// Array to hold files in the directory

File directory[MAX\_FILES];

int file\_count = 0;

// Function to create a file

void create\_file() {

if (file\_count >= MAX\_FILES) {

printf("Directory is full. Cannot create more files.\n");

return;

}

char filename[MAX\_FILENAME\_LENGTH];

printf("Enter the name of the file to create: ");

scanf("%s", filename);

// Check if file already exists

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

if (strcmp(directory[i].name, filename) == 0) {

printf("File already exists.\n");

return;

}

}

// Create the file

strcpy(directory[file\_count].name, filename);

file\_count++;

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

}

// Function to list all files

void list\_files() {

if (file\_count == 0) {

printf("No files in the directory.\n");

return;

}

printf("Files in the directory:\n");

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

printf("%s\n", directory[i].name);

}

}

// Function to delete a file

void delete\_file() {

if (file\_count == 0) {

printf("No files to delete.\n");

return;

}

char filename[MAX\_FILENAME\_LENGTH];

printf("Enter the name of the file to delete: ");

scanf("%s", filename);

// Find the file and delete it

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

if (strcmp(directory[i].name, filename) == 0) {

// Shift remaining files

for (int j = i; j < file\_count - 1; j++) {

strcpy(directory[j].name, directory[j + 1].name);

}

file\_count--;

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

return;

}

}

printf("File not found.\n");

}

int main() {

int choice;

while (1) {

printf("\nSingle-Level Directory\n");

printf("1. Create File\n");

printf("2. List Files\n");

printf("3. Delete File\n");

printf("4. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

create\_file();

break;

case 2:

list\_files();

break;

case 3:

delete\_file();

break;

case 4:

exit(0);

default:

printf("Invalid choice. Please try again.\n");

}

}

return 0;

}

OUTPUT

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

// Define maximum number of files and file name length

#define MAX\_FILES 100

#define MAX\_FILENAME\_LENGTH 100

// Structure to represent a file

typedef struct {

char name[MAX\_FILENAME\_LENGTH];

} File;

// Array to hold files in the directory

File directory[MAX\_FILES];

int file\_count = 0;

// Function to create a file

void create\_file() {

if (file\_count >= MAX\_FILES) {

printf("Directory is full. Cannot create more files.\n");

return;

}

char filename[MAX\_FILENAME\_LENGTH];

printf("Enter the name of the file to create: ");

scanf("%s", filename);

// Check if file already exists

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

if (strcmp(directory[i].name, filename) == 0) {

printf("File already exists.\n");

return;

}

}

// Create the file

strcpy(directory[file\_count].name, filename);

file\_count++;

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

}

// Function to list all files

void list\_files() {

if (file\_count == 0) {

printf("No files in the directory.\n");

return;

}

printf("Files in the directory:\n");

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

printf("%s\n", directory[i].name);

}

}

// Function to delete a file

void delete\_file() {

if (file\_count == 0) {

printf("No files to delete.\n");

return;

}

char filename[MAX\_FILENAME\_LENGTH];

printf("Enter the name of the file to delete: ");

scanf("%s", filename);

// Find the file and delete it

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

if (strcmp(directory[i].name, filename) == 0) {

// Shift remaining files

for (int j = i; j < file\_count - 1; j++) {

strcpy(directory[j].name, directory[j + 1].name);

}

file\_count--;

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

return;

}

}

printf("File not found.\n");

}

int main() {

int choice;

while (1) {

printf("\nSingle-Level Directory\n");

printf("1. Create File\n");

printf("2. List Files\n");

printf("3. Delete File\n");

printf("4. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

create\_file();

break;

case 2:

list\_files();

break;

case 3:

delete\_file();

break;

case 4:

exit(0);

default:

printf("Invalid choice. Please try again.\n");

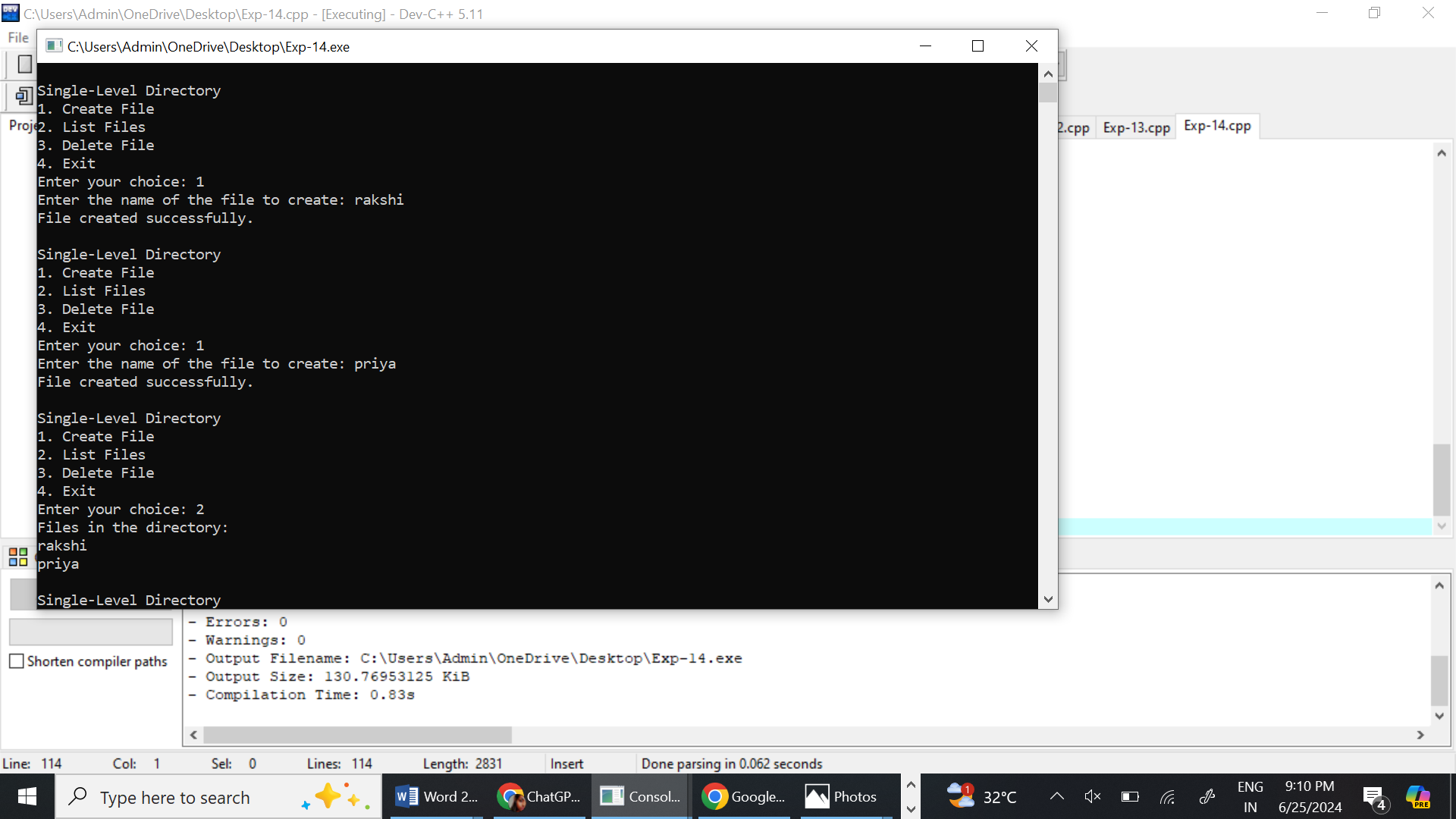
}

}

return 0;

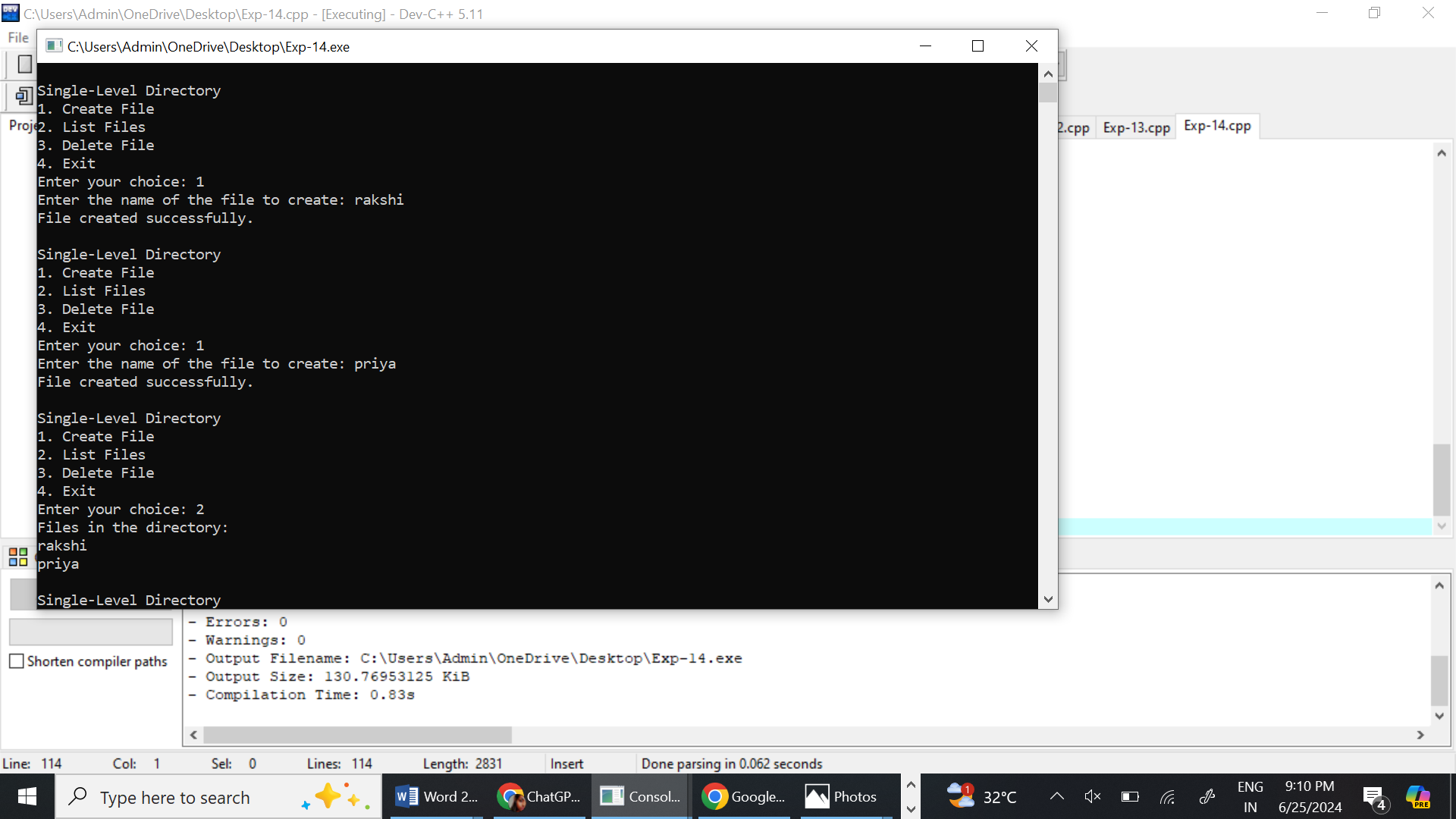
}

OUTPUT



15. Design a C program to organize the file using two level directory structure.

PROGRAM



16. Develop a C program for implementing random access file for processing the employee details.

PROGRAM

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define FILENAME "employees.dat"

typedef struct {

int id;

char name[50];

float salary;

} Employee;

void addEmployee(FILE \*file) {

Employee emp;

printf("Enter Employee ID: ");

scanf("%d", &emp.id);

printf("Enter Employee Name: ");

scanf("%s", emp.name);

printf("Enter Employee Salary: ");

scanf("%f", &emp.salary);

fseek(file, 0, SEEK\_END); // Move to the end of the file

fwrite(&emp, sizeof(Employee), 1, file); // Write the employee record to the file

printf("Employee added successfully.\n");

}

void displayEmployees(FILE \*file) {

Employee emp;

fseek(file, 0, SEEK\_SET); // Move to the start of the file

while(fread(&emp, sizeof(Employee), 1, file)) {

printf("Employee ID: %d\n", emp.id);

printf("Employee Name: %s\n", emp.name);

printf("Employee Salary: %.2f\n\n", emp.salary);

}

}

void searchEmployee(FILE \*file) {

int id;

Employee emp;

printf("Enter Employee ID to search: ");

scanf("%d", &id);

fseek(file, 0, SEEK\_SET); // Move to the start of the file

while(fread(&emp, sizeof(Employee), 1, file)) {

if(emp.id == id) {

printf("Employee found:\n");

printf("Employee ID: %d\n", emp.id);

printf("Employee Name: %s\n", emp.name);

printf("Employee Salary: %.2f\n", emp.salary);

return;

}

}

printf("Employee with ID %d not found.\n", id);

}

int main() {

FILE \*file;

int choice;

file = fopen(FILENAME, "rb+");

if(file == NULL) {

file = fopen(FILENAME, "wb+");

if(file == NULL) {

printf("Unable to open file.\n");

return 1;

}

}

while(1) {

printf("\nEmployee Management System\n");

printf("1. Add Employee\n");

printf("2. Display All Employees\n");

printf("3. Search Employee by ID\n");

printf("4. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch(choice) {

case 1:

addEmployee(file);

break;

case 2:

displayEmployees(file);

break;

case 3:

searchEmployee(file);

break;

case 4:

fclose(file);

exit(0);

default:

printf("Invalid choice. Please try again.\n");

}

}

return 0;

}

OUTPUT

