DAY-4

25. Construct a C program to implement the I/O system calls of UNIX (fcntl, seek, stat, opendir, readdir)

PROGRAM

#include <stdio.h>

#include <stdlib.h>

#include <fcntl.h>

#include <unistd.h>

#include <sys/stat.h>

#include <dirent.h>

void demo\_fcntl(const char \*filename) {

int fd = open(filename, O\_RDWR);

if (fd == -1) {

perror("open");

exit(EXIT\_FAILURE);

}

// Get file descriptor flags

int flags = fcntl(fd, F\_GETFL);

if (flags == -1) {

perror("fcntl");

close(fd);

exit(EXIT\_FAILURE);

}

printf("File descriptor flags: %d\n", flags);

// Set file descriptor flags

if (fcntl(fd, F\_SETFL, flags | O\_APPEND) == -1) {

perror("fcntl");

close(fd);

exit(EXIT\_FAILURE);

}

printf("File descriptor flags set to O\_APPEND.\n");

close(fd);

}

void demo\_lseek(const char \*filename) {

int fd = open(filename, O\_RDWR);

if (fd == -1) {

perror("open");

exit(EXIT\_FAILURE);

}

// Seek to the end of the file

off\_t offset = lseek(fd, 0, SEEK\_END);

if (offset == (off\_t)-1) {

perror("lseek");

close(fd);

exit(EXIT\_FAILURE);

}

printf("File size: %ld bytes\n", offset);

close(fd);

}

void demo\_stat(const char \*filename) {

struct stat sb;

if (stat(filename, &sb) == -1) {

perror("stat");

exit(EXIT\_FAILURE);

}

printf("File type: ");

switch (sb.st\_mode & S\_IFMT) {

case S\_IFBLK: printf("block device\n"); break;

case S\_IFCHR: printf("character device\n"); break;

case S\_IFDIR: printf("directory\n"); break;

case S\_IFIFO: printf("FIFO/pipe\n"); break;

case S\_IFLNK: printf("symlink\n"); break;

case S\_IFREG: printf("regular file\n"); break;

case S\_IFSOCK: printf("socket\n"); break;

default: printf("unknown?\n"); break;

}

printf("I-node number: %ld\n", (long) sb.st\_ino);

printf("Mode: %lo (octal)\n", (unsigned long) sb.st\_mode);

printf("Link count: %ld\n", (long) sb.st\_nlink);

printf("Ownership: UID=%ld GID=%ld\n", (long) sb.st\_uid, (long) sb.st\_gid);

printf("Preferred I/O block size: %ld bytes\n", (long) sb.st\_blksize);

printf("File size: %lld bytes\n", (long long) sb.st\_size);

printf("Blocks allocated: %lld\n", (long long) sb.st\_blocks);

}

void demo\_readdir(const char \*dirpath) {

DIR \*dir;

struct dirent \*entry;

if ((dir = opendir(dirpath)) == NULL) {

perror("opendir");

exit(EXIT\_FAILURE);

}

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

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

}

closedir(dir);

}

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

if (argc != 3) {

fprintf(stderr, "Usage: %s <filename> <directory>\n", argv[0]);

exit(EXIT\_FAILURE);

}

const char \*filename = argv[1];

const char \*directory = argv[2];

printf("Demonstrating fcntl:\n");

demo\_fcntl(filename);

printf("\nDemonstrating lseek:\n");

demo\_lseek(filename);

printf("\nDemonstrating stat:\n");

demo\_stat(filename);

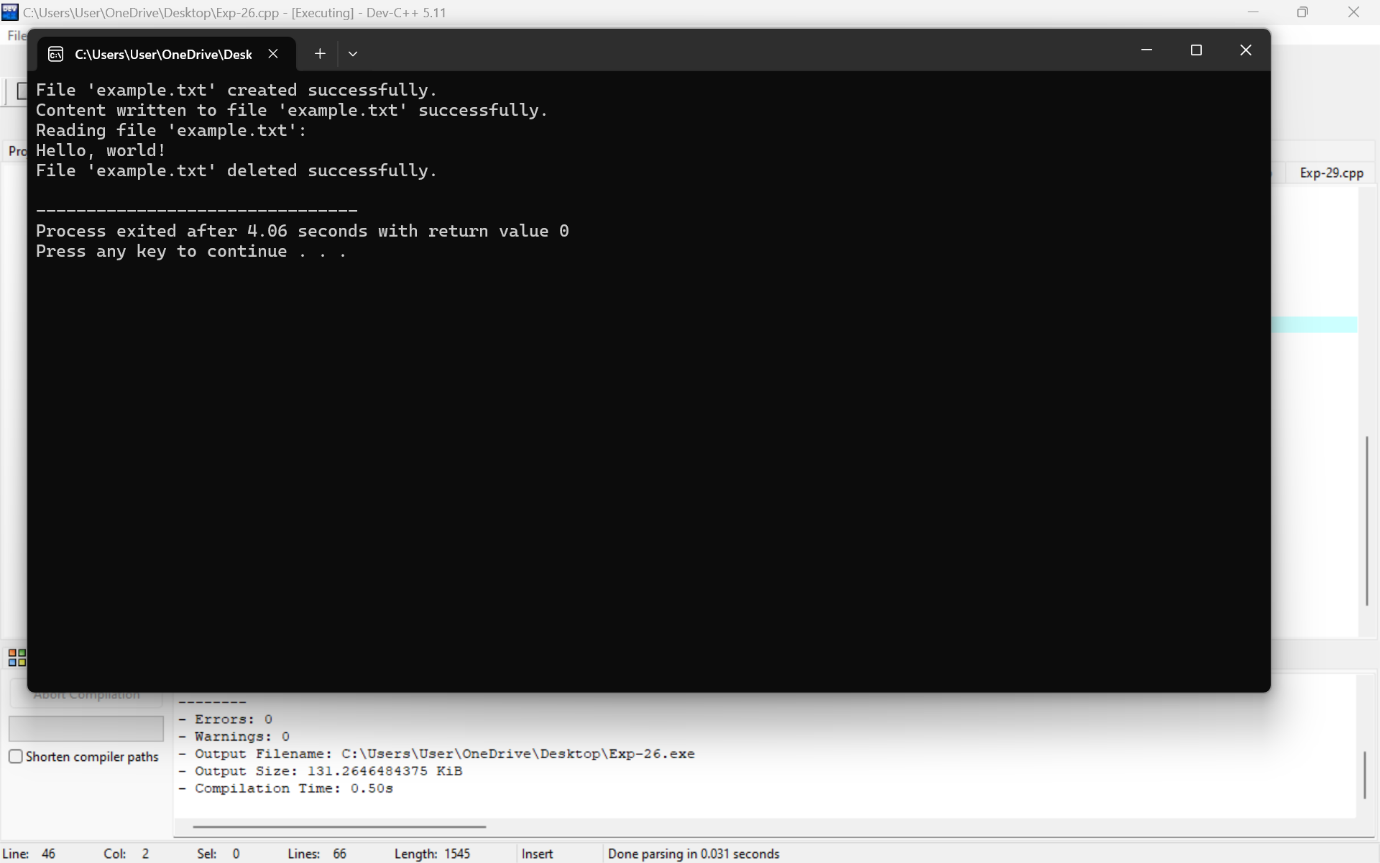
printf("\nDemonstrating readdir:\n");

demo\_readdir(directory);

return 0;

}

OUTPUT



26. Construct a C program to implement the file management operations.

PROGRAM

#include <stdio.h>

#include <stdlib.h>

void createFile(const char \*filename) {

FILE \*file = fopen(filename, "w");

if (file == NULL) {

perror("Error creating file");

exit(EXIT\_FAILURE);

}

printf("File '%s' created successfully.\n", filename);

fclose(file);

}

void writeFile(const char \*filename, const char \*content) {

FILE \*file = fopen(filename, "w");

if (file == NULL) {

perror("Error writing to file");

exit(EXIT\_FAILURE);

}

fprintf(file, "%s", content);

printf("Content written to file '%s' successfully.\n", filename);

fclose(file);

}

void readFile(const char \*filename) {

char ch;

FILE \*file = fopen(filename, "r");

if (file == NULL) {

perror("Error reading file");

exit(EXIT\_FAILURE);

}

printf("Reading file '%s':\n", filename);

while ((ch = fgetc(file)) != EOF) {

putchar(ch);

}

printf("\n");

fclose(file);

}

void deleteFile(const char \*filename) {

if (remove(filename) == 0) {

printf("File '%s' deleted successfully.\n", filename);

} else {

perror("Error deleting file");

}

}

int main() {

const char \*filename = "example.txt";

const char \*content = "Hello, world!";

// Create a file

createFile(filename);

// Write to the file

writeFile(filename, content);

// Read from the file

readFile(filename);

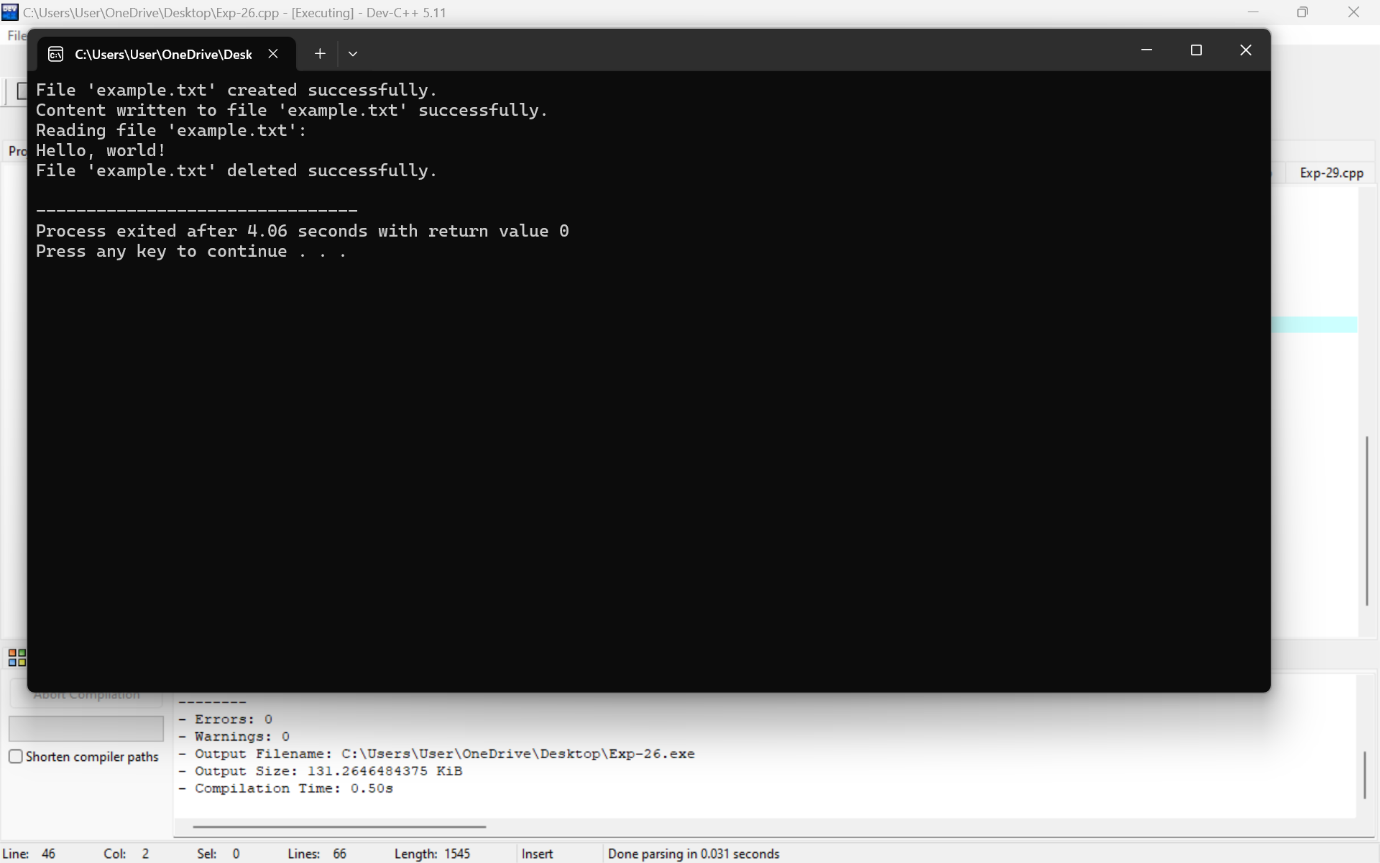
// Delete the file

deleteFile(filename);

return 0;

}

OUTPUT



27. Develop a C program for simulating the function of ls UNIX Command.

PROGRAM

#include <stdio.h>

#include <stdlib.h>

#include <dirent.h>

void list\_directory(const char \*path) {

struct dirent \*entry;

DIR \*dp = opendir(path);

if (dp == NULL) {

perror("opendir");

return;

}

while ((entry = readdir(dp))) {

if (entry->d\_name[0] != '.') { // Hide hidden files

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

}

}

closedir(dp);

}

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

const char \*path;

if (argc > 1) {

path = argv[1];

} else {

path = ".";

}

list\_directory(path);

return 0;

}

OUTPUT

28. Write a C program for simulation of GREP UNIX command

PROGRAM

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define MAX\_LINE\_LENGTH 1024

void grep(const char \*pattern, FILE \*file) {

char line[MAX\_LINE\_LENGTH];

while (fgets(line, sizeof(line), file)) {

if (strstr(line, pattern)) {

printf("%s", line);

}

}

}

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

if (argc != 3) {

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

return EXIT\_FAILURE;

}

const char \*pattern = argv[1];

const char \*filename = argv[2];

FILE \*file = fopen(filename, "r");

if (!file) {

perror("Error opening file");

return EXIT\_FAILURE;

}

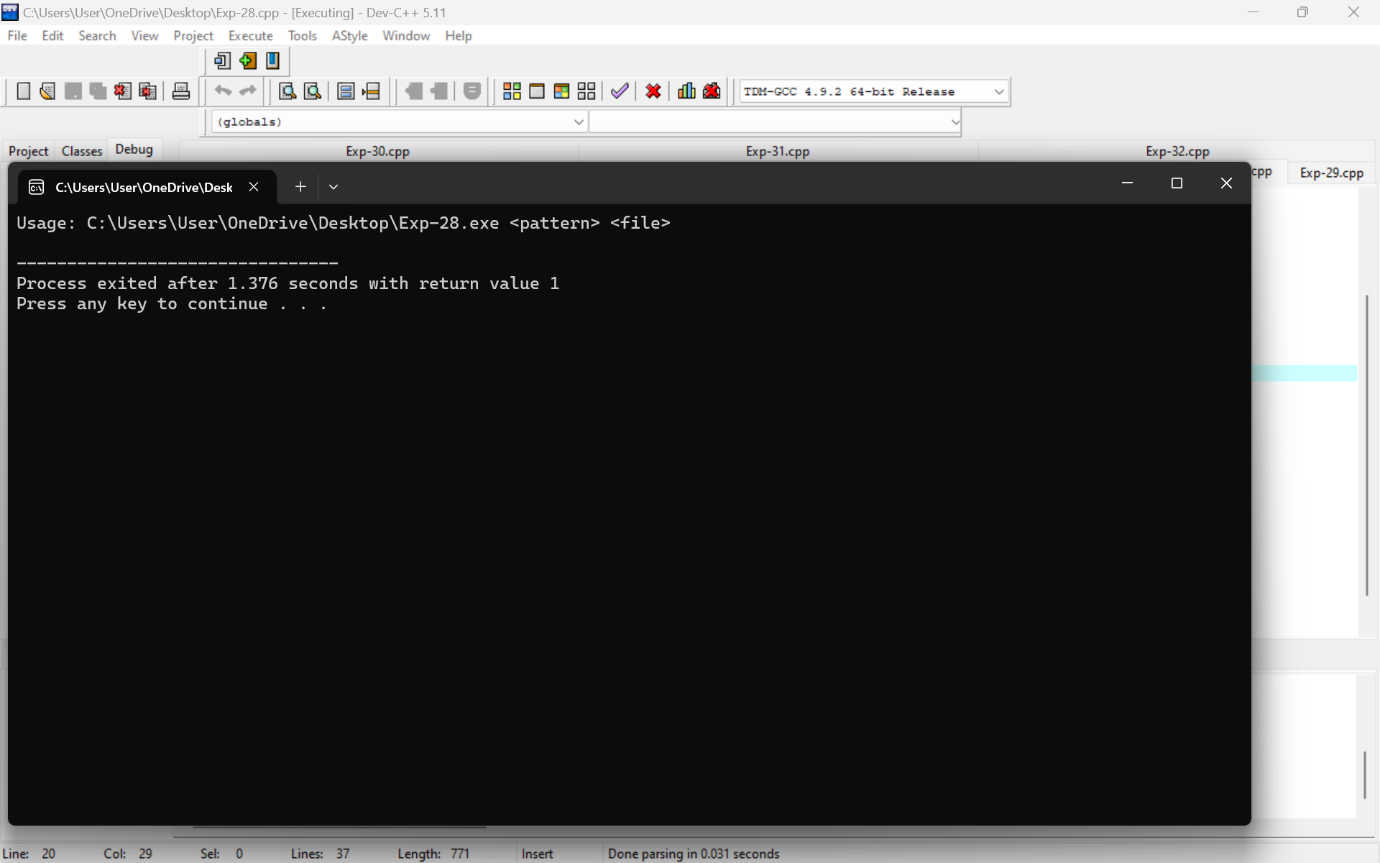
grep(pattern, file);

fclose(file);

return EXIT\_SUCCESS;

}

OUTPUT



29. Write a C program to simulate the solution of Classical Process Synchronization Problem

PROGRAM

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>

#define BUFFER\_SIZE 10

// Buffer and its associated variables

int buffer[BUFFER\_SIZE];

int count = 0;

int in = 0;

int out = 0;

// Semaphores and mutex

sem\_t empty;

sem\_t full;

pthread\_mutex\_t mutex;

// Producer function

void \*producer(void \*param) {

int item;

while (1) {

// Produce an item

item = rand() % 100;

// Wait if buffer is full

sem\_wait(&empty);

// Lock the buffer

pthread\_mutex\_lock(&mutex);

// Add the item to the buffer

buffer[in] = item;

in = (in + 1) % BUFFER\_SIZE;

count++;

printf("Producer produced: %d\n", item);

// Unlock the buffer

pthread\_mutex\_unlock(&mutex);

// Signal that buffer is not empty

sem\_post(&full);

// Simulate time taken to produce an item

sleep(rand() % 3);

}

}

// Consumer function

void \*consumer(void \*param) {

int item;

while (1) {

// Wait if buffer is empty

sem\_wait(&full);

// Lock the buffer

pthread\_mutex\_lock(&mutex);

// Remove the item from the buffer

item = buffer[out];

out = (out + 1) % BUFFER\_SIZE;

count--;

printf("Consumer consumed: %d\n", item);

// Unlock the buffer

pthread\_mutex\_unlock(&mutex);

// Signal that buffer is not full

sem\_post(&empty);

// Simulate time taken to consume an item

sleep(rand() % 3);

}

}

int main() {

pthread\_t producer\_thread, consumer\_thread;

// Initialize semaphores

sem\_init(&empty, 0, BUFFER\_SIZE);

sem\_init(&full, 0, 0);

// Initialize mutex

pthread\_mutex\_init(&mutex, NULL);

// Create producer and consumer threads

pthread\_create(&producer\_thread, NULL, producer, NULL);

pthread\_create(&consumer\_thread, NULL, consumer, NULL);

// Wait for the threads to finish (they actually run indefinitely in this example)

pthread\_join(producer\_thread, NULL);

pthread\_join(consumer\_thread, NULL);

// Destroy semaphores and mutex (unreachable in this example)

sem\_destroy(&empty);

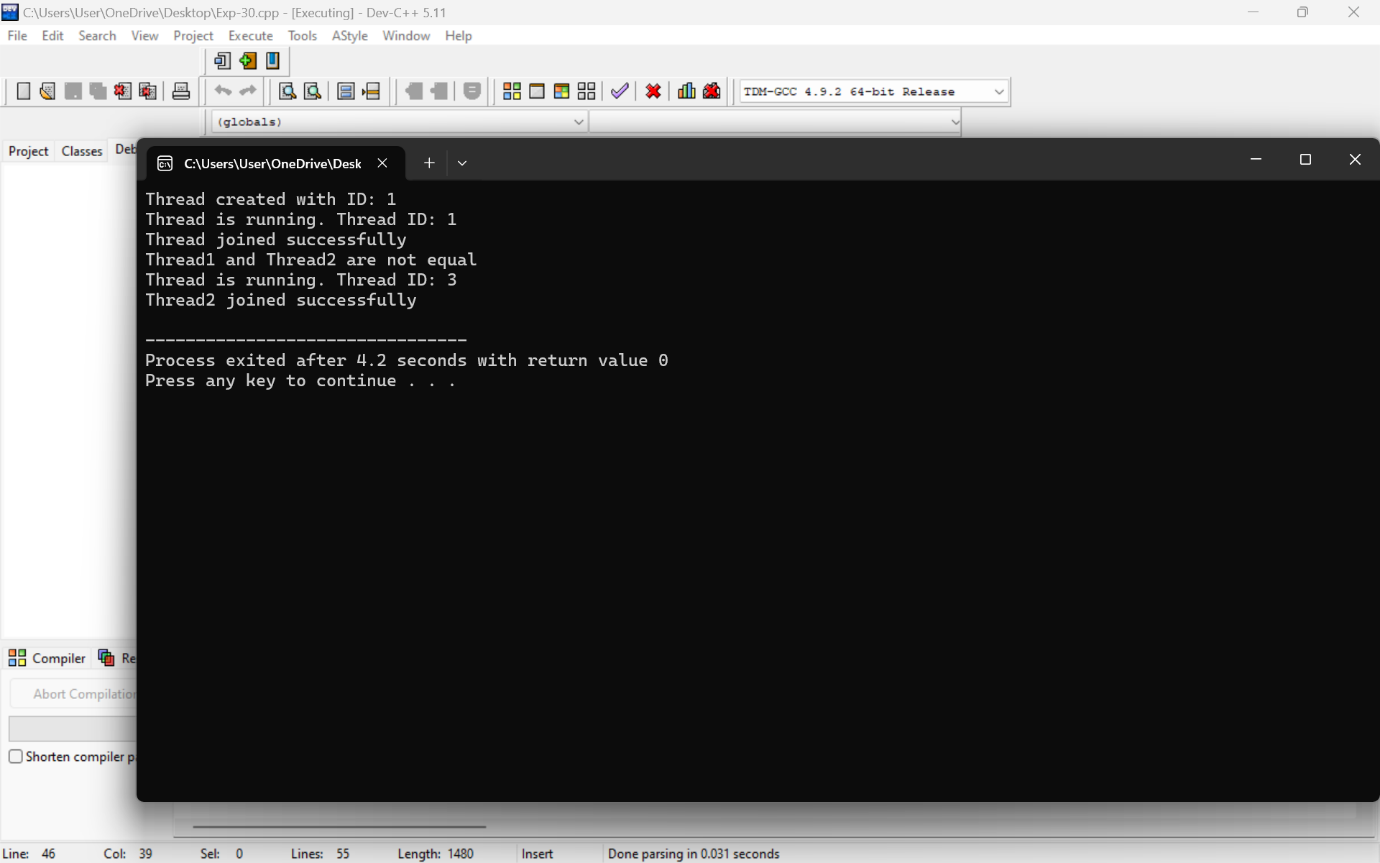
sem\_destroy(&full);

pthread\_mutex\_destroy(&mutex);

return 0;

}

OUTPUT



30. Write C programs to demonstrate the following thread related concepts.

(i) create (ii) join (iii) equal (iv) exit

PROGRAM

#include <pthread.h>

#include <stdio.h>

#include <stdlib.h>

// Thread function to be executed

void\* threadFunc(void\* arg) {

printf("Thread is running. Thread ID: %ld\n", pthread\_self());

pthread\_exit(NULL); // Exit the thread

}

int main() {

pthread\_t thread1, thread2;

int res;

// (i) Create a new thread

res = pthread\_create(&thread1, NULL, threadFunc, NULL);

if (res != 0) {

perror("Thread creation failed");

exit(EXIT\_FAILURE);

}

printf("Thread created with ID: %ld\n", thread1);

// (ii) Join the thread (wait for it to complete)

res = pthread\_join(thread1, NULL);

if (res != 0) {

perror("Thread join failed");

exit(EXIT\_FAILURE);

}

printf("Thread joined successfully\n");

// (iii) Create another thread to demonstrate pthread\_equal

res = pthread\_create(&thread2, NULL, threadFunc, NULL);

if (res != 0) {

perror("Thread creation failed");

exit(EXIT\_FAILURE);

}

// Check if two thread IDs are equal

if (pthread\_equal(thread1, thread2)) {

printf("Thread1 and Thread2 are equal\n");

} else {

printf("Thread1 and Thread2 are not equal\n");

}

// Join the second thread

res = pthread\_join(thread2, NULL);

if (res != 0) {

perror("Thread join failed");

exit(EXIT\_FAILURE);

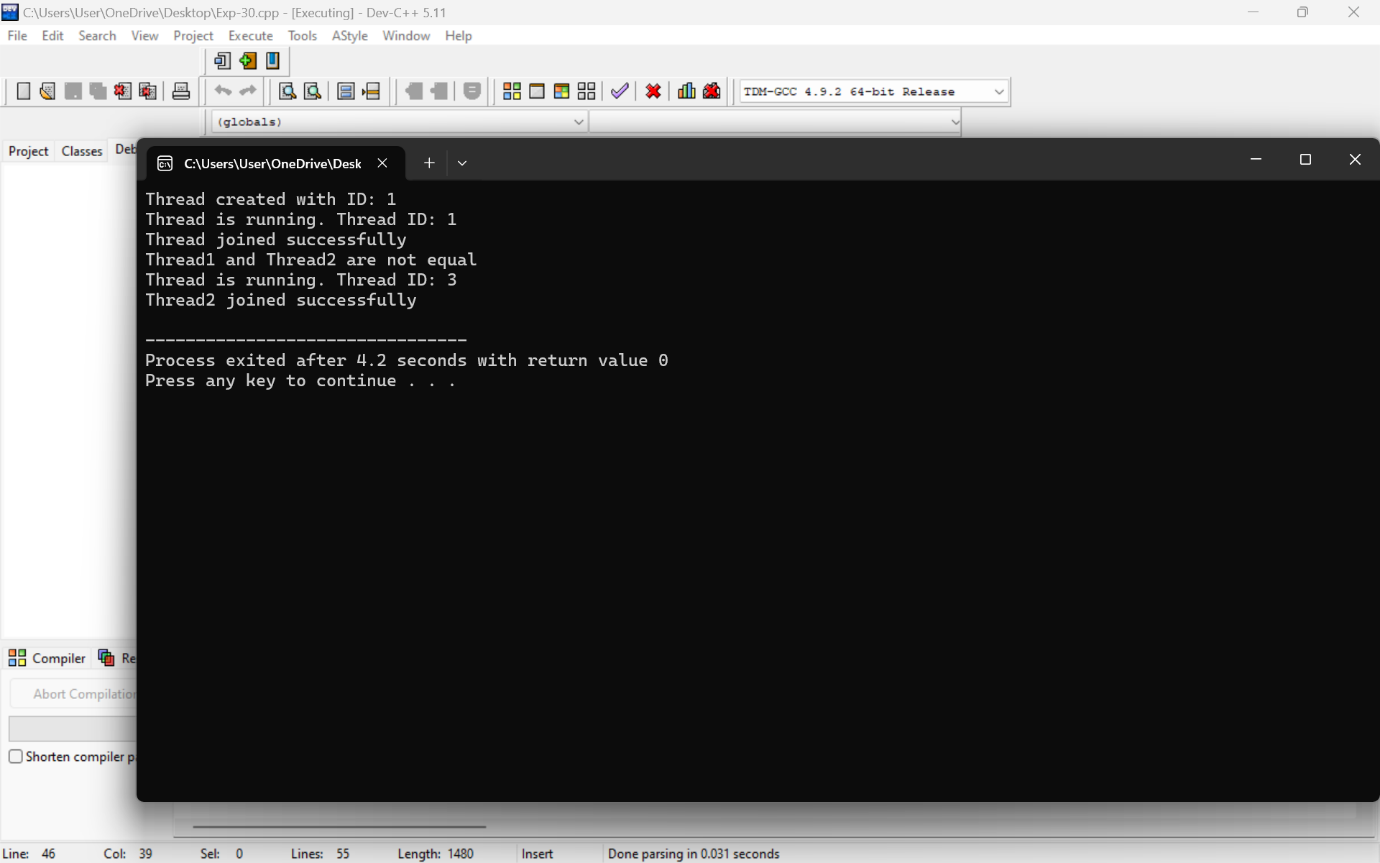
}

printf("Thread2 joined successfully\n");

return 0;

}

OUTPUT



31. Construct a C program to simulate the First in First Out paging technique of memory management.

PROGRAM

#include <stdio.h>

#include <stdbool.h>

#define FRAME\_SIZE 3 // Size of the page frame (number of frames)

// Function to check if a page is already in memory

bool isInMemory(int page, int frames[], int frameSize) {

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

if (frames[i] == page) {

return true;

}

}

return false;

}

// Function to display the current state of memory frames

void displayFrames(int frames[], int frameSize) {

printf("Current memory state: ");

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

if (frames[i] == -1) {

printf("- ");

} else {

printf("%d ", frames[i]);

}

}

printf("\n");

}

// Function to simulate FIFO page replacement algorithm

void FIFO(int pages[], int page\_count, int frameSize) {

int frames[frameSize]; // Array to store current memory frames

int frameIndex = 0; // Index to keep track of oldest frame to replace

int pageFaults = 0; // Count of page faults

// Initialize frames with -1 to indicate empty frames

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

frames[i] = -1;

}

// Iterate over each page in sequence

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

printf("Accessing page %d...\n", pages[i]);

// If page fault occurs (page is not in memory)

if (!isInMemory(pages[i], frames, frameSize)) {

frames[frameIndex] = pages[i]; // Replace the oldest frame

frameIndex = (frameIndex + 1) % frameSize; // Update oldest frame index

pageFaults++; // Increment page fault count

displayFrames(frames, frameSize); // Display current memory state

} else {

printf("Page %d is already in memory.\n", pages[i]);

displayFrames(frames, frameSize); // Display current memory state

}

}

printf("\nTotal Page Faults: %d\n", pageFaults);

}

int main() {

int pages[] = {1, 2, 3, 4, 2, 1, 5, 6, 2, 1, 2, 3, 7, 6, 3, 2, 1, 2, 3, 6};

int page\_count = sizeof(pages) / sizeof(pages[0]);

int frameSize = FRAME\_SIZE;

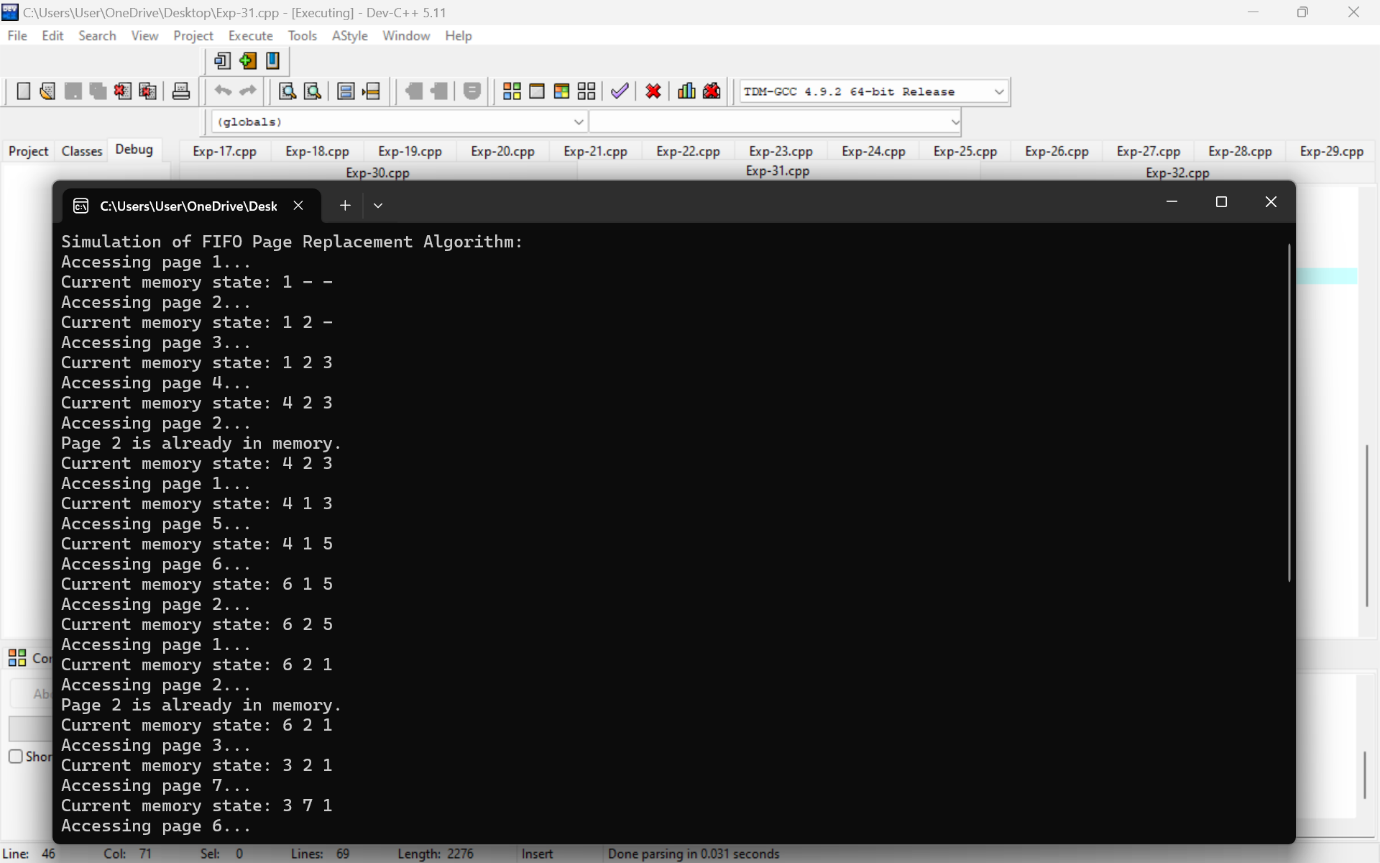
printf("Simulation of FIFO Page Replacement Algorithm:\n");

FIFO(pages, page\_count, frameSize);

return 0;

}

OUTPUT



32. Construct a C program to simulate the Least Recently Used paging technique of memory management.

PROGRAM

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#define MAX\_FRAMES 3 // Maximum number of frames in memory

// Function declarations

int lruPageReplacement(int pages[], int n, int capacity);

int main() {

int pages[] = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2}; // Reference string example

int n = sizeof(pages) / sizeof(pages[0]);

int capacity = MAX\_FRAMES;

int pageFaults = lruPageReplacement(pages, n, capacity);

printf("Total page faults: %d\n", pageFaults);

return 0;

}

// Function to simulate LRU page replacement algorithm

int lruPageReplacement(int pages[], int n, int capacity) {

int pageFaults = 0;

int frames[capacity];

int time[capacity]; // To store the time of last use of each frame

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

frames[i] = -1; // Initialize frames with -1 indicating empty

time[i] = 0;

}

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

bool found = false;

// Check if page is already in memory

for (int j = 0; j < capacity; j++) {

if (frames[j] == pages[i]) {

found = true;

time[j] = i; // Update time of last use to current reference index

break;

}

}

// If page is not present in memory, replace the least recently used page

if (!found) {

int lruIndex = 0;

// Find the frame with the least recently used page

for (int j = 1; j < capacity; j++) {

if (time[j] < time[lruIndex]) {

lruIndex = j;

}

}

frames[lruIndex] = pages[i]; // Replace the least recently used page

time[lruIndex] = i; // Update time of last use to current reference index

pageFaults++;

}

// Print the current state of frames after each page reference

printf("Current frames: ");

for (int j = 0; j < capacity; j++) {

if (frames[j] == -1) {

printf(" - ");

} else {

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

}

}

printf("\n");

}

return pageFaults;

}

OUTPUT

