**DAY-3**

17.Illustrate the deadlock avoidance concept by simulating Banker’s algorithm with C.

PROGRAM

#include <stdio.h>

#include <stdbool.h>

#define NUM\_PROCESSES 5

#define NUM\_RESOURCES 3

int available[NUM\_RESOURCES] = {3, 3, 2};

int maximum[NUM\_PROCESSES][NUM\_RESOURCES] = {

{7, 5, 3},

{3, 2, 2},

{9, 0, 2},

{2, 2, 2},

{4, 3, 3}

};

int allocation[NUM\_PROCESSES][NUM\_RESOURCES] = {

{0, 1, 0},

{2, 0, 0},

{3, 0, 2},

{2, 1, 1},

{0, 0, 2}

};

int need[NUM\_PROCESSES][NUM\_RESOURCES];

void calculate\_need() {

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

for (int j = 0; j < NUM\_RESOURCES; j++) {

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

}

}

}

bool is\_safe() {

bool finish[NUM\_PROCESSES] = {false};

int work[NUM\_RESOURCES];

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

work[i] = available[i];

}

while (true) {

bool found = false;

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

if (!finish[i]) {

bool possible = true;

for (int j = 0; j < NUM\_RESOURCES; j++) {

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

possible = false;

break;

}

}

if (possible) {

for (int j = 0; j < NUM\_RESOURCES; j++) {

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

}

finish[i] = true;

found = true;

}

}

}

if (!found) {

break;

}

}

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

if (!finish[i]) {

return false;

}

}

return true;

}

bool request\_resources(int process\_id, int request[]) {

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

if (request[i] > need[process\_id][i]) {

printf("Error: Process has exceeded its maximum claim.\n");

return false;

}

if (request[i] > available[i]) {

printf("Error: Resources not available.\n");

return false;

}

}

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

available[i] -= request[i];

allocation[process\_id][i] += request[i];

need[process\_id][i] -= request[i];

}

if (is\_safe()) {

printf("Resources allocated successfully.\n");

return true;

} else {

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

available[i] += request[i];

allocation[process\_id][i] -= request[i];

need[process\_id][i] += request[i];

}

printf("Error: System is not in a safe state.\n");

return false;

}

}

void print\_state() {

printf("Current state:\n");

printf("Available resources:\n");

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

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

}

printf("\n");

printf("Maximum resources:\n");

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

for (int j = 0; j < NUM\_RESOURCES; j++) {

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

}

printf("\n");

}

printf("Allocated resources:\n");

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

for (int j = 0; j < NUM\_RESOURCES; j++) {

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

}

printf("\n");

}

printf("Need resources:\n");

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

for (int j = 0; j < NUM\_RESOURCES; j++) {

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

}

printf("\n");

}

}

int main() {

calculate\_need();

print\_state();

int process\_id = 1;

int request[] = {1, 0, 2};

printf("Process %d requesting resources:\n", process\_id);

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

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

}

printf("\n");

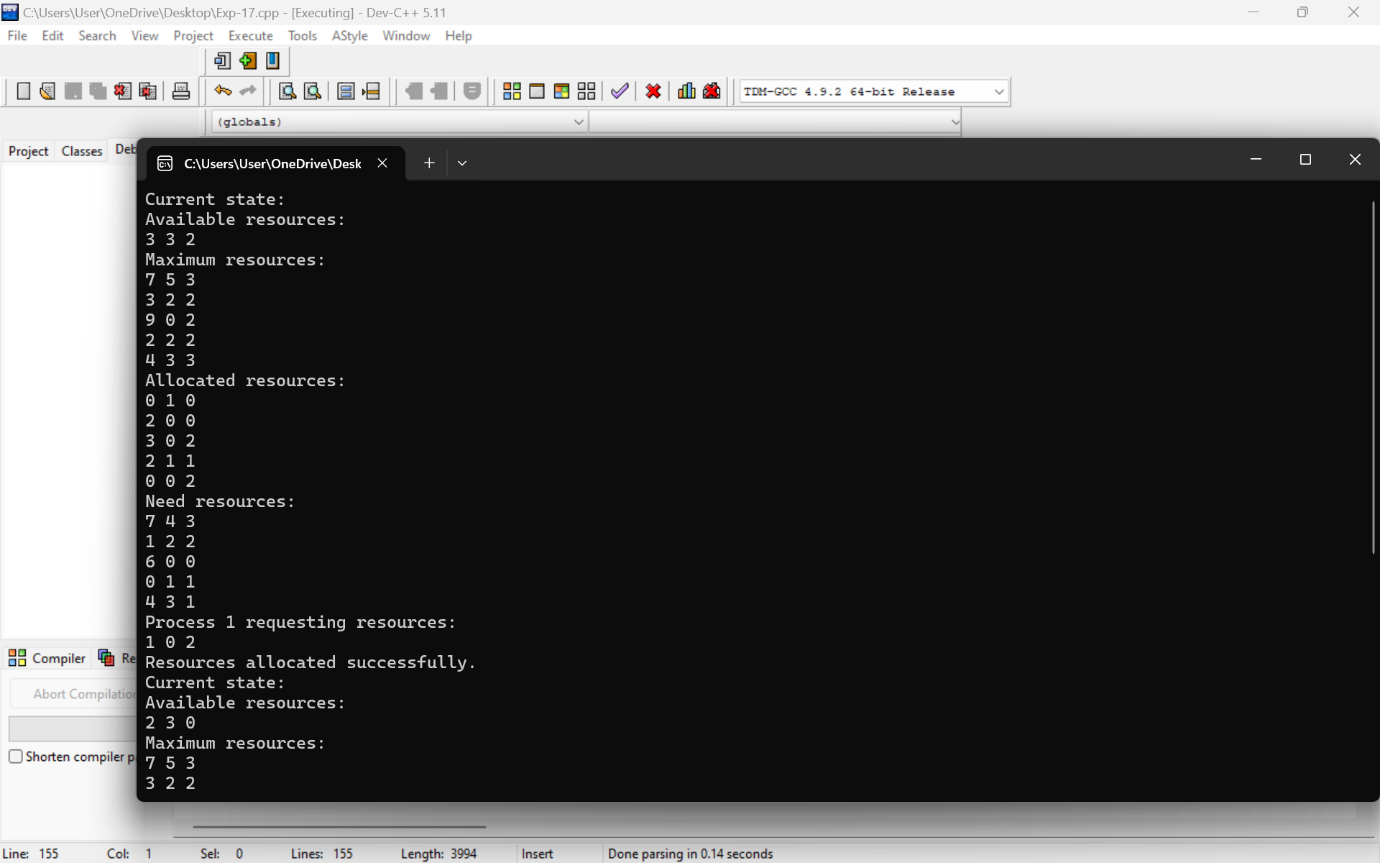
request\_resources(process\_id, request);

print\_state();

return 0;

}

OUTPUT



18.Construct a C program to simulate producer-consumer problem using semaphores.

PROGRAM

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>

#define BUFFER\_SIZE 10

#define NUM\_ITEMS 20

int buffer[BUFFER\_SIZE];

int in = 0, out = 0;

sem\_t empty;

sem\_t full;

pthread\_mutex\_t mutex;

void\* producer(void\* arg) {

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

// Produce an item (here, just the index for simplicity)

int item = i;

// Wait for an empty slot

sem\_wait(&empty);

// Lock the buffer

pthread\_mutex\_lock(&mutex);

// Add the item to the buffer

buffer[in] = item;

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

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

// Unlock the buffer

pthread\_mutex\_unlock(&mutex);

// Signal that there's a new item in the buffer

sem\_post(&full);

}

return NULL;

}

void\* consumer(void\* arg) {

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

// Wait for a filled slot

sem\_wait(&full);

// Lock the buffer

pthread\_mutex\_lock(&mutex);

// Remove the item from the buffer

int item = buffer[out];

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

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

// Unlock the buffer

pthread\_mutex\_unlock(&mutex);

// Signal that there's an empty slot in the buffer

sem\_post(&empty);

}

return NULL;

}

int main() {

pthread\_t prod\_thread, cons\_thread;

// Initialize the semaphores and mutex

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

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

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

// Create the producer and consumer threads

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

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

// Wait for the threads to finish

pthread\_join(prod\_thread, NULL);

pthread\_join(cons\_thread, NULL);

// Destroy the semaphores and mutex

sem\_destroy(&empty);

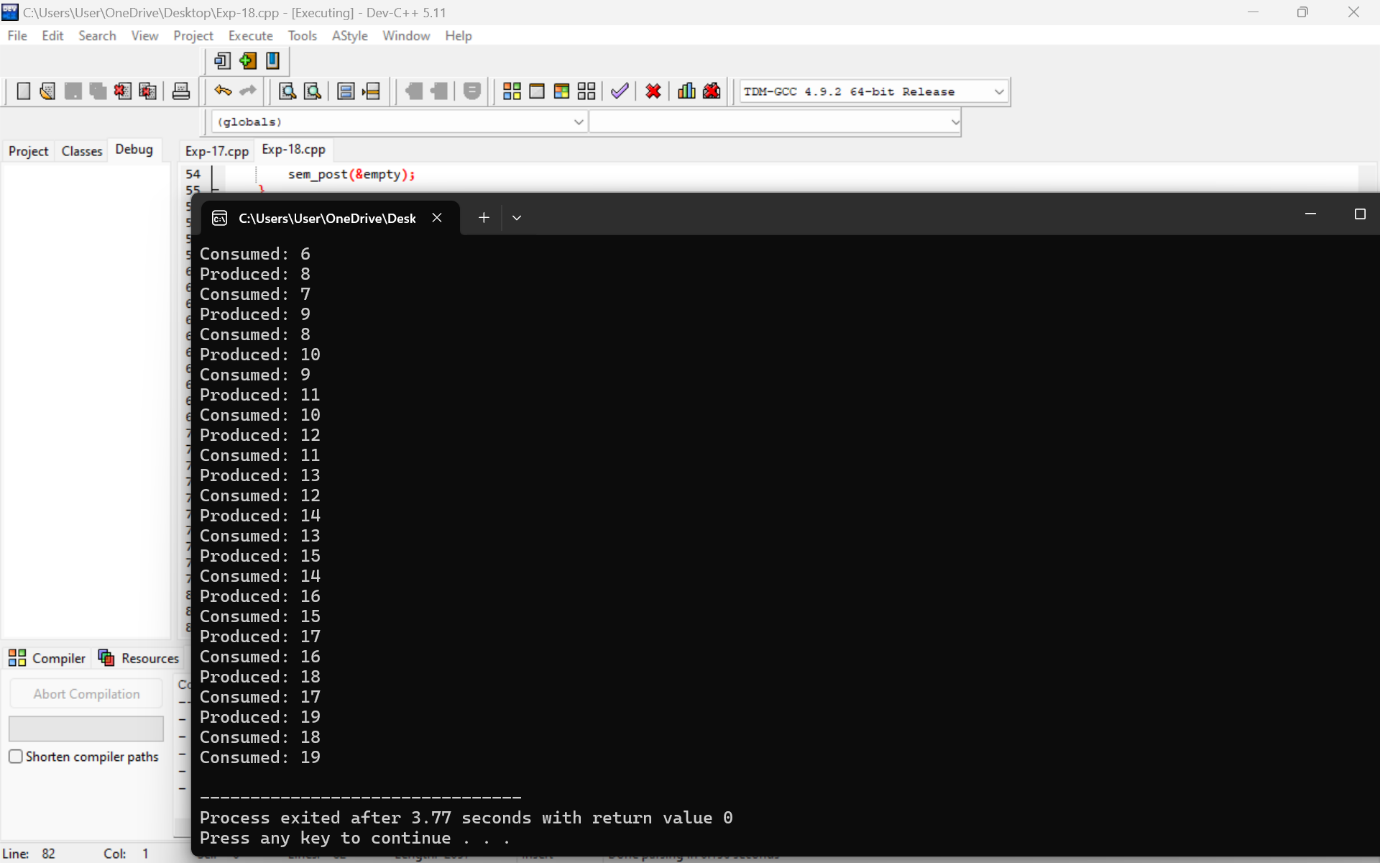
sem\_destroy(&full);

pthread\_mutex\_destroy(&mutex);

return 0;

}

OUTPUT



1. Design a C program to implement process synchronization using mutex locks.

PROGRAM

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

// Shared counter

int counter = 0;

// Mutex lock

pthread\_mutex\_t lock;

// Function to increment the counter

void\* increment(void\* arg) {

for (int i = 0; i < 1000000; ++i) {

// Lock the mutex before accessing the shared counter

pthread\_mutex\_lock(&lock);

++counter;

// Unlock the mutex after accessing the shared counter

pthread\_mutex\_unlock(&lock);

}

return NULL;

}

int main() {

// Initialize the mutex lock

if (pthread\_mutex\_init(&lock, NULL) != 0) {

printf("Mutex initialization failed\n");

return 1;

}

pthread\_t thread1, thread2;

// Create two threads

pthread\_create(&thread1, NULL, increment, NULL);

pthread\_create(&thread2, NULL, increment, NULL);

// Wait for both threads to finish

pthread\_join(thread1, NULL);

pthread\_join(thread2, NULL);

// Destroy the mutex lock

pthread\_mutex\_destroy(&lock);

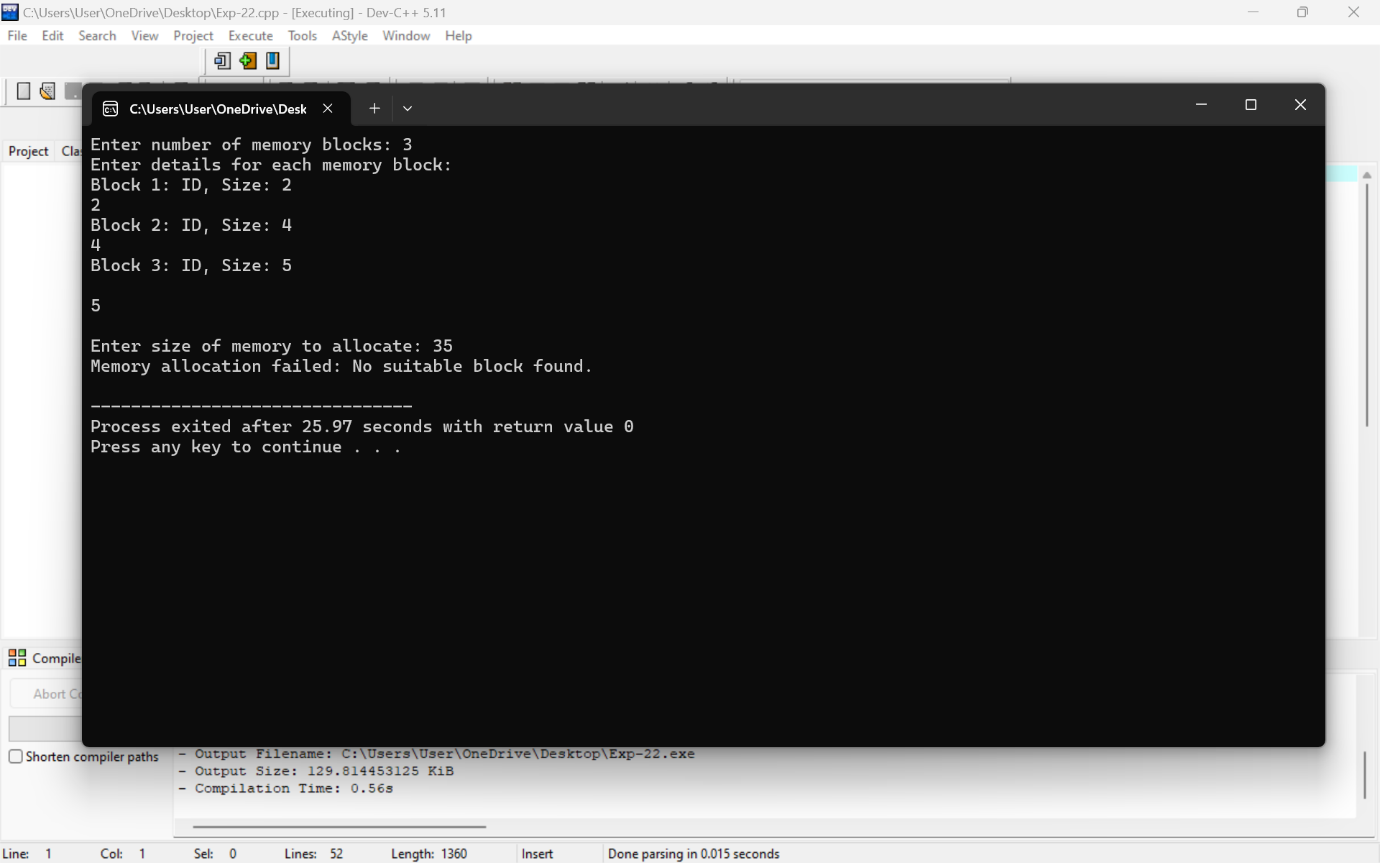
// Print the final value of the counter

printf("Final counter value: %d\n", counter);

return 0;

}

OUTPUT



1. Construct a C program to simulate Reader-Writer problem using Semaphores.

PROGRAM

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>

#include <unistd.h>

#define NUM\_READERS 5

#define NUM\_WRITERS 2

sem\_t mutex, writeblock;

int data = 0, readers\_count = 0;

void \*reader(void \*arg) {

int reader\_id = \*((int \*)arg);

while (1) {

usleep(100000); // Simulate reading time

sem\_wait(&mutex);

readers\_count++;

if (readers\_count == 1) {

sem\_wait(&writeblock);

}

sem\_post(&mutex);

printf("Reader %d reads data: %d\n", reader\_id, data);

sem\_wait(&mutex);

readers\_count--;

if (readers\_count == 0) {

sem\_post(&writeblock);

}

sem\_post(&mutex);

}

}

void \*writer(void \*arg) {

int writer\_id = \*((int \*)arg);

while (1) {

usleep(1000000); // Simulate writing time

sem\_wait(&writeblock);

data++;

printf("Writer %d writes data: %d\n", writer\_id, data);

sem\_post(&writeblock);

}

}

int main() {

pthread\_t readers[NUM\_READERS], writers[NUM\_WRITERS];

int reader\_ids[NUM\_READERS], writer\_ids[NUM\_WRITERS];

sem\_init(&mutex, 0, 1);

sem\_init(&writeblock, 0, 1);

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

reader\_ids[i] = i + 1;

pthread\_create(&readers[i], NULL, reader, &reader\_ids[i]);

}

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

writer\_ids[i] = i + 1;

pthread\_create(&writers[i], NULL, writer, &writer\_ids[i]);

}

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

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

}

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

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

}

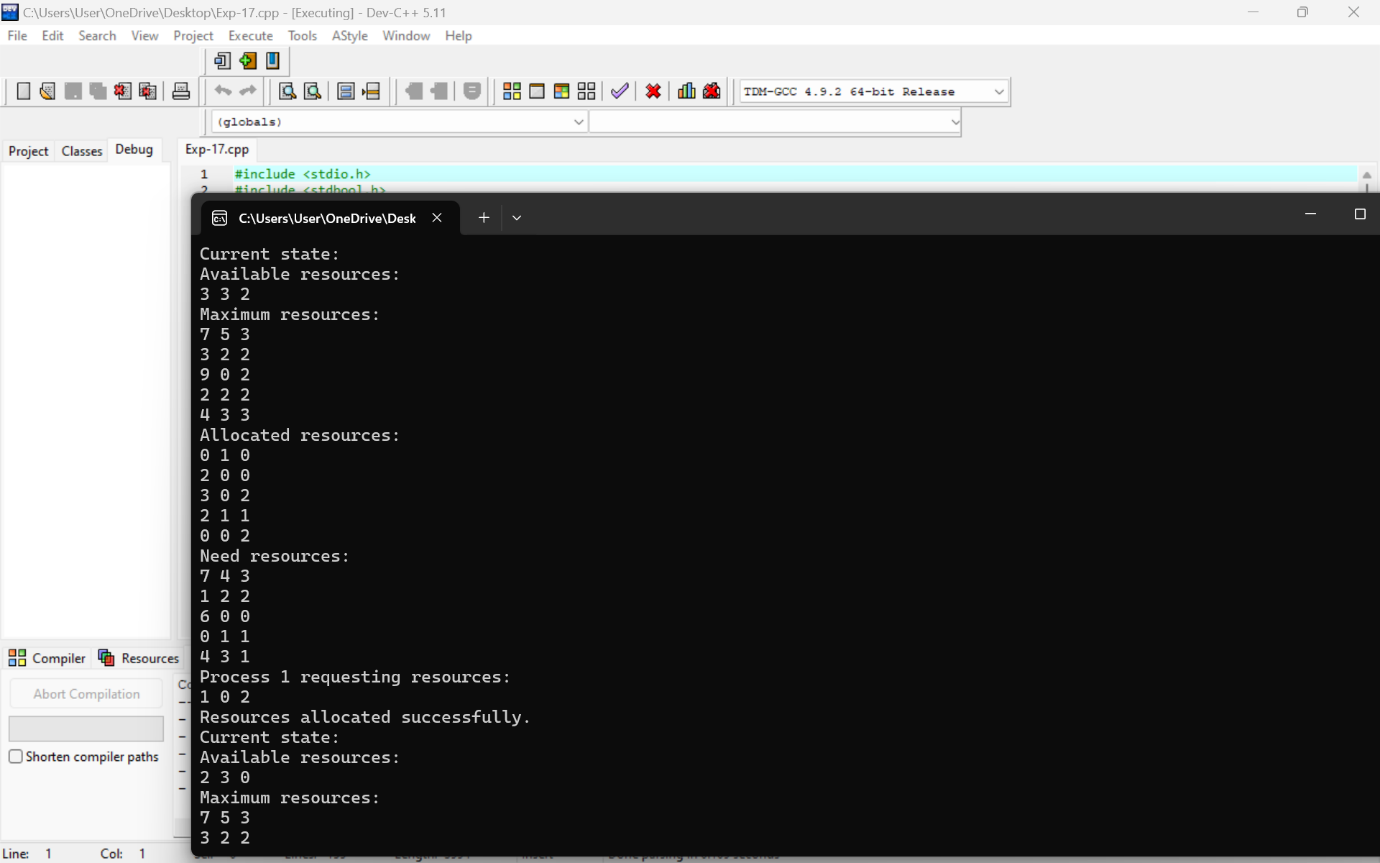
sem\_destroy(&mutex);

sem\_destroy(&writeblock);

return 0;

}

OUTPUT



1. Develop a C program to implement worst fit algorithm of memory management.
2. Construct a C program to implement best fit algorithm of memory management.

PROGRAM

#include <stdio.h>

#include <stdlib.h>

#define MAX\_PARTITIONS 50

struct MemoryPartition {

int size;

int allocated;

};

void worstFit(int blockSize[], int m, int processSize[], int n) {

struct MemoryPartition partitions[MAX\_PARTITIONS];

int allocation[MAX\_PARTITIONS];

// Initialize memory partitions

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

partitions[i].size = blockSize[i];

partitions[i].allocated = -1; // -1 means not allocated

}

// Process each process

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

// Find the worst fit partition

int worstIdx = -1;

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

if (partitions[j].allocated == -1 && partitions[j].size >= processSize[i]) {

if (worstIdx == -1 || partitions[j].size > partitions[worstIdx].size) {

worstIdx = j;

}

}

}

// Allocate memory if we found a partition

if (worstIdx != -1) {

allocation[i] = worstIdx;

partitions[worstIdx].allocated = i;

} else {

allocation[i] = -1; // No suitable partition found

}

}

// Print the allocation result

printf("\nProcess No.\tProcess Size\tBlock no.\n");

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

printf("%d\t\t%d\t\t", i + 1, processSize[i]);

if (allocation[i] != -1) {

printf("%d\n", allocation[i] + 1); // +1 to display block number starting from 1

} else {

printf("Not Allocated\n");

}

}

}

int main() {

int blockSize[] = {100, 500, 200, 300, 600};

int processSize[] = {212, 417, 112, 426};

int m = sizeof(blockSize) / sizeof(blockSize[0]);

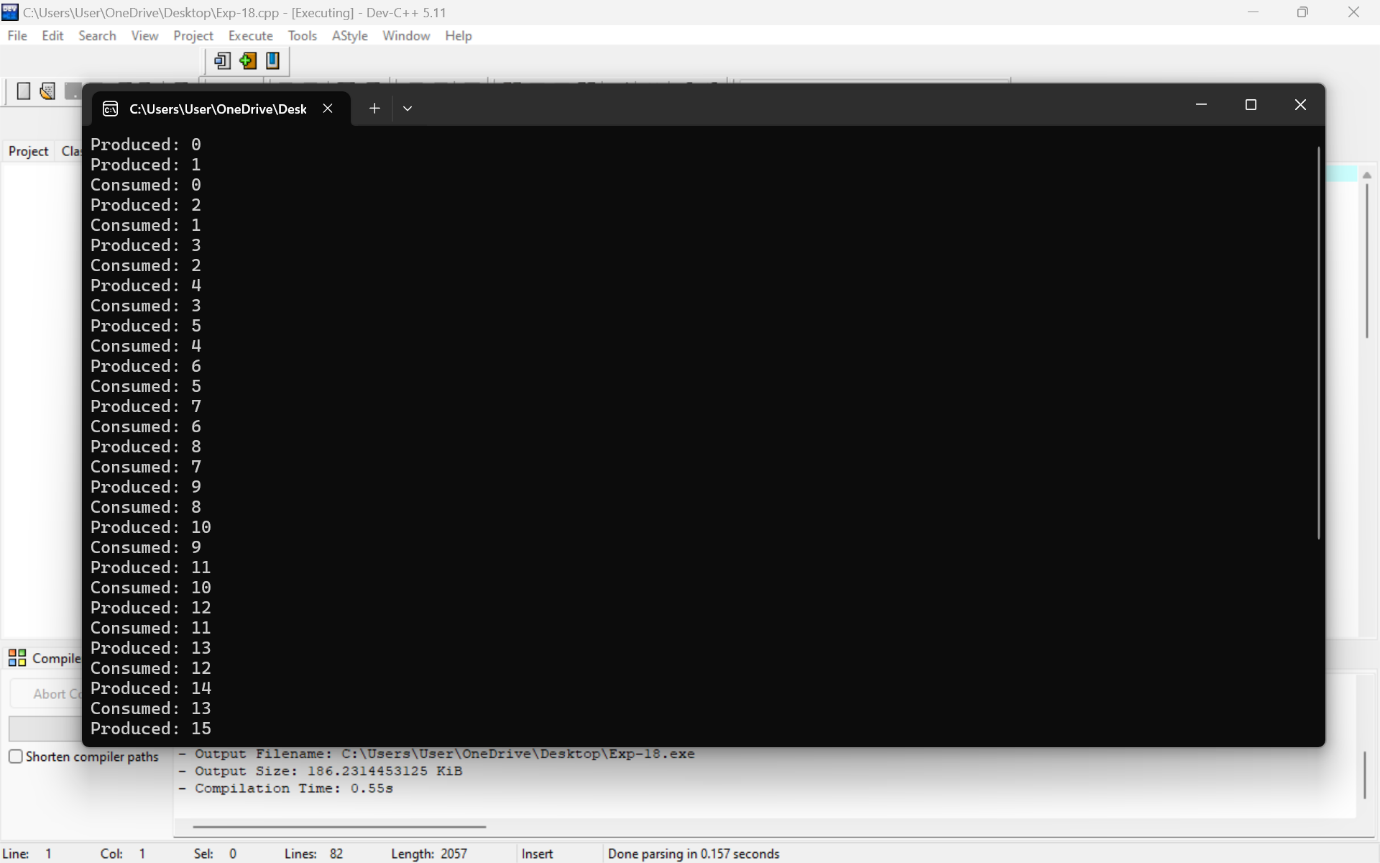
int n = sizeof(processSize) / sizeof(processSize[0]);

worstFit(blockSize, m, processSize, n);

return 0;

}

OUTPUT



1. Construct a C program to implement first fit algorithm of memory management.

PROGRAM

#include <stdio.h>

#include <stdlib.h>

#define MAX\_MEMORY 100

struct MemoryBlock {

int id;

int size;

int allocated;

};

void bestFit(struct MemoryBlock memory[], int numBlocks, int requestSize) {

int bestFitIndex = -1;

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

if (!memory[i].allocated && memory[i].size >= requestSize) {

if (bestFitIndex == -1 || memory[i].size < memory[bestFitIndex].size) {

bestFitIndex = i;

}

}

}

if (bestFitIndex == -1) {

printf("Memory allocation failed: No suitable block found.\n");

} else {

memory[bestFitIndex].allocated = 1;

printf("Memory allocated successfully at block %d.\n", bestFitIndex + 1);

}

}

int main() {

struct MemoryBlock memory[MAX\_MEMORY];

int numBlocks, requestSize;

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

scanf("%d", &numBlocks);

printf("Enter details for each memory block:\n");

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

printf("Block %d: ID, Size: ", i + 1);

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

memory[i].allocated = 0;

}

printf("\nEnter size of memory to allocate: ");

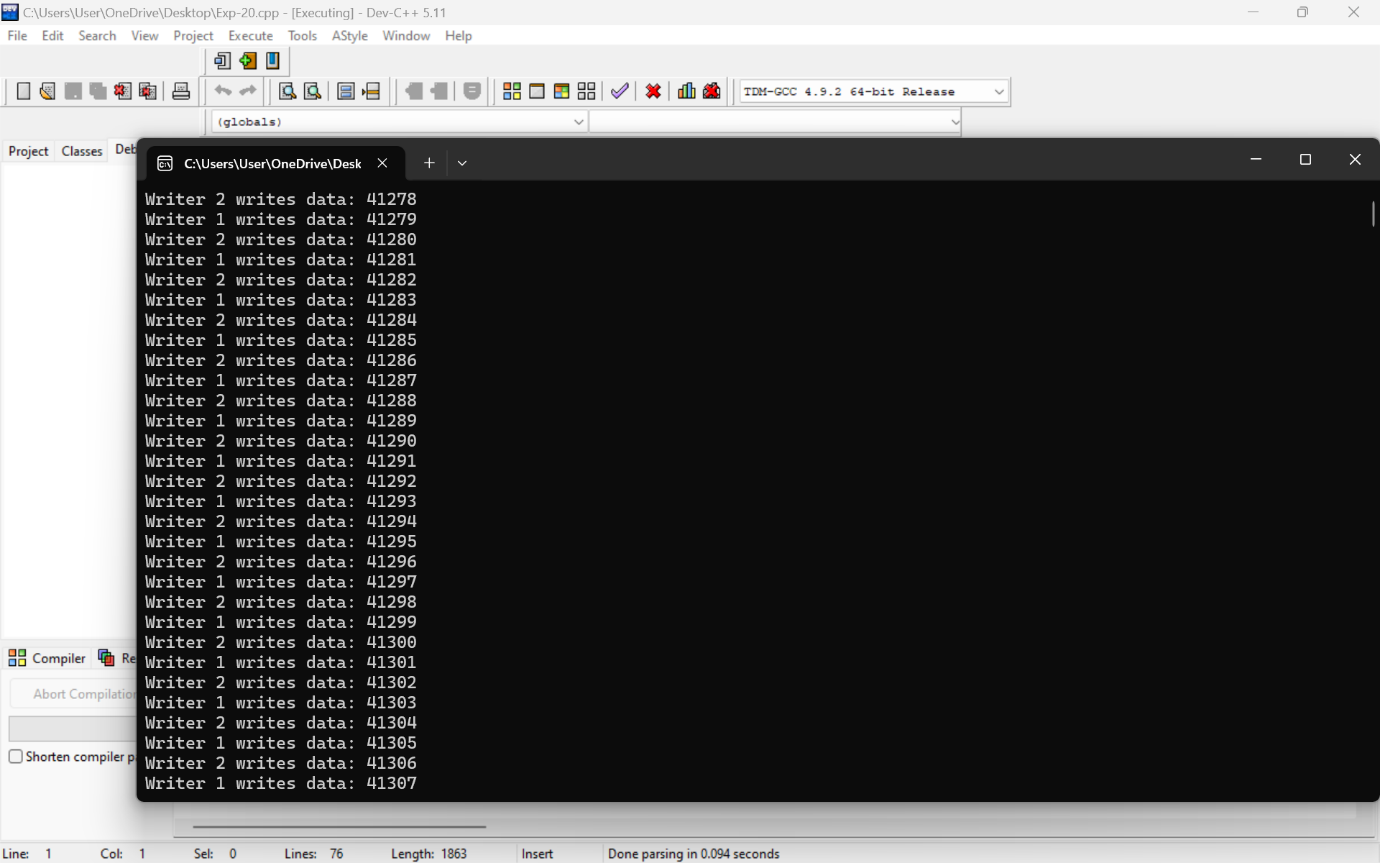
scanf("%d", &requestSize);

bestFit(memory, numBlocks, requestSize);

return 0;

}

OUTPUT



1. Design a C program to demonstrate UNIX system calls for file management.

PROGRAM

#include <stdio.h>

#include <stdlib.h>

#include <fcntl.h>

#include <unistd.h>

#include <string.h>

#define FILENAME "example.txt"

#define BUFFER\_SIZE 1024

int main() {

int fd;

ssize\_t bytes\_written, bytes\_read;

char write\_buffer[] = "Hello, UNIX system calls!";

char read\_buffer[BUFFER\_SIZE];

// Create a new file (or open it if it already exists) with read and write permissions

fd = open(FILENAME, O\_CREAT | O\_WRONLY | O\_TRUNC, S\_IRUSR | S\_IWUSR);

if (fd == -1) {

perror("Error opening file for writing");

exit(EXIT\_FAILURE);

}

// Write data to the file

bytes\_written = write(fd, write\_buffer, strlen(write\_buffer));

if (bytes\_written == -1) {

perror("Error writing to file");

close(fd);

exit(EXIT\_FAILURE);

}

printf("Wrote %ld bytes to %s\n", bytes\_written, FILENAME);

// Close the file

if (close(fd) == -1) {

perror("Error closing file after writing");

exit(EXIT\_FAILURE);

}

// Open the file for reading

fd = open(FILENAME, O\_RDONLY);

if (fd == -1) {

perror("Error opening file for reading");

exit(EXIT\_FAILURE);

}

// Read data from the file

bytes\_read = read(fd, read\_buffer, BUFFER\_SIZE - 1);

if (bytes\_read == -1) {

perror("Error reading from file");

close(fd);

exit(EXIT\_FAILURE);

}

// Null-terminate the read buffer to make it a proper string

read\_buffer[bytes\_read] = '\0';

printf("Read %ld bytes from %s: %s\n", bytes\_read, FILENAME, read\_buffer);

// Close the file

if (close(fd) == -1) {

perror("Error closing file after reading");

exit(EXIT\_FAILURE);

}

// Delete the file

if (unlink(FILENAME) == -1) {

perror("Error deleting file");

exit(EXIT\_FAILURE);

}

printf("Deleted %s\n", FILENAME);

return 0;

}

OUTPUT

