16. #include <stdio.h>

int main()

{

int pid[15];

int bt[15];

int n;

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

scanf("%d",&n);

printf("Enter process id of all the processes: ");

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

{

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

}

printf("Enter burst time of all the processes: ");

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

{

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

}

int i, wt[n];

wt[0]=0;

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

{

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

}

printf("Process ID Burst Time Waiting Time TurnAround Time\n");

float twt=0.0;

float tat= 0.0;

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

{

printf("%d\t\t", pid[i]);

printf("%d\t\t", bt[i]);

printf("%d\t\t", wt[i]);

//calculating and printing turnaround time of each process

printf("%d\t\t", bt[i]+wt[i]);

printf("\n");

//for calculating total waiting time

twt += wt[i];

//for calculating total turnaround time

tat += (wt[i]+bt[i]);

}

float att,awt;

//for calculating average waiting time

awt = twt/n;

//for calculating average turnaround time

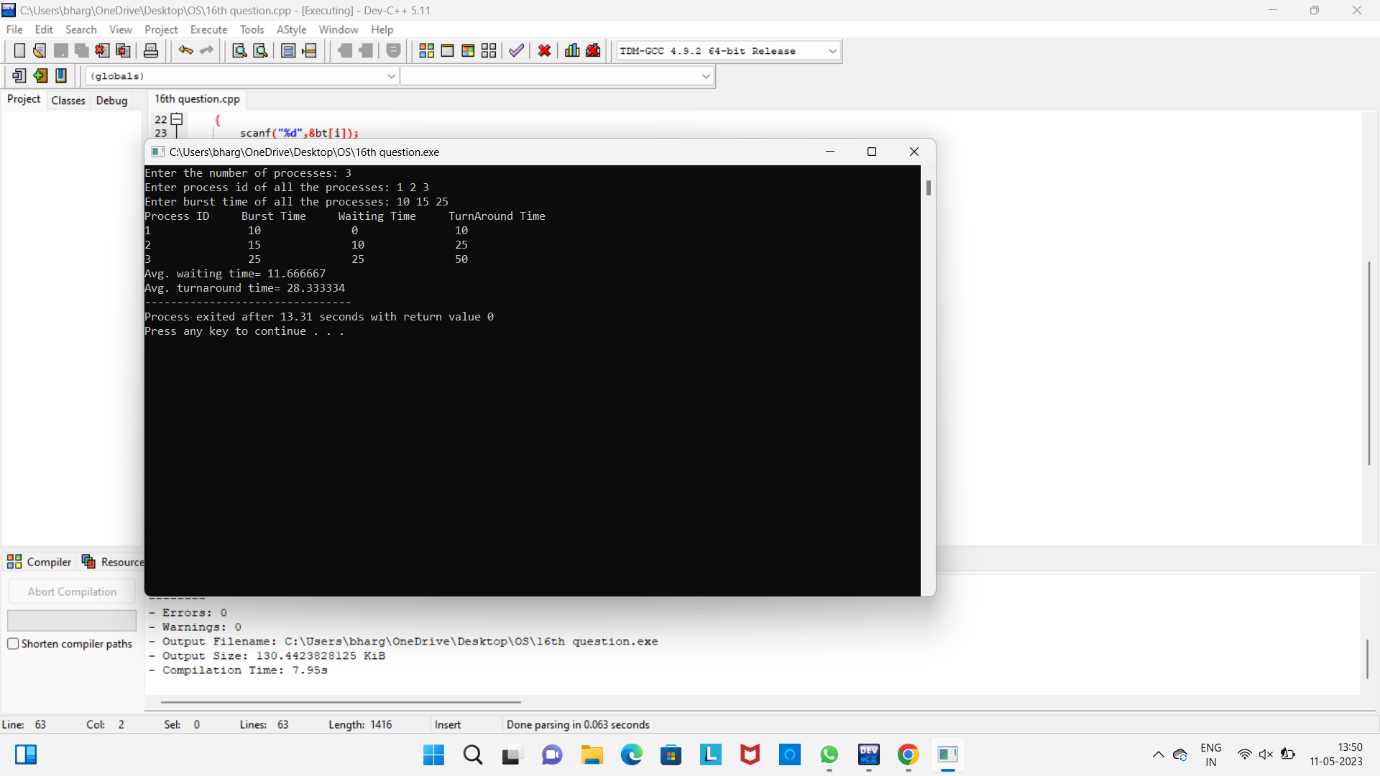
att = tat/n;

printf("Avg. waiting time= %f\n",awt);

printf("Avg. turnaround time= %f",att);

}

Output



17. #include <stdio.h>

struct Process {

char name[3];

int burst\_time;

int remaining\_time;

int waiting\_time;

int turnaround\_time;

};

void round\_robin(struct Process processes[], int n, int quantum) {

int i, j;

int waiting\_time = 0;

int turnaround\_time = 0;

// Initialize the remaining time for each process

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

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

}

// Process the queue until it becomes empty

while (1) {

int completed = 1;

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

// Execute the process for the quantum time

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

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

printf("Running %s for %d ms\n", processes[i].name, quantum);

processes[i].remaining\_time -= quantum;

} else {

quantum = processes[i].remaining\_time;

printf("Running %s for %d ms\n", processes[i].name, quantum);

processes[i].remaining\_time = 0;

}

completed = 0;

// Check if the process is completed

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

processes[i].turnaround\_time = waiting\_time + processes[i].burst\_time;

processes[i].waiting\_time = processes[i].turnaround\_time - processes[i].burst\_time;

turnaround\_time += processes[i].turnaround\_time;

waiting\_time += processes[i].waiting\_time;

}

}

}

// Break the loop if all processes are completed

if (completed == 1) {

break;

}

}

// Calculate the average waiting time and average turnaround time

float avg\_waiting\_time = (float)waiting\_time / n;

float avg\_turnaround\_time = (float)turnaround\_time / n;

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

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

}

int main() {

struct Process processes[] = {

{"P1", 24},

{"P2", 3},

{"P3", 3}

};

int num\_processes = sizeof(processes) / sizeof(processes[0]);

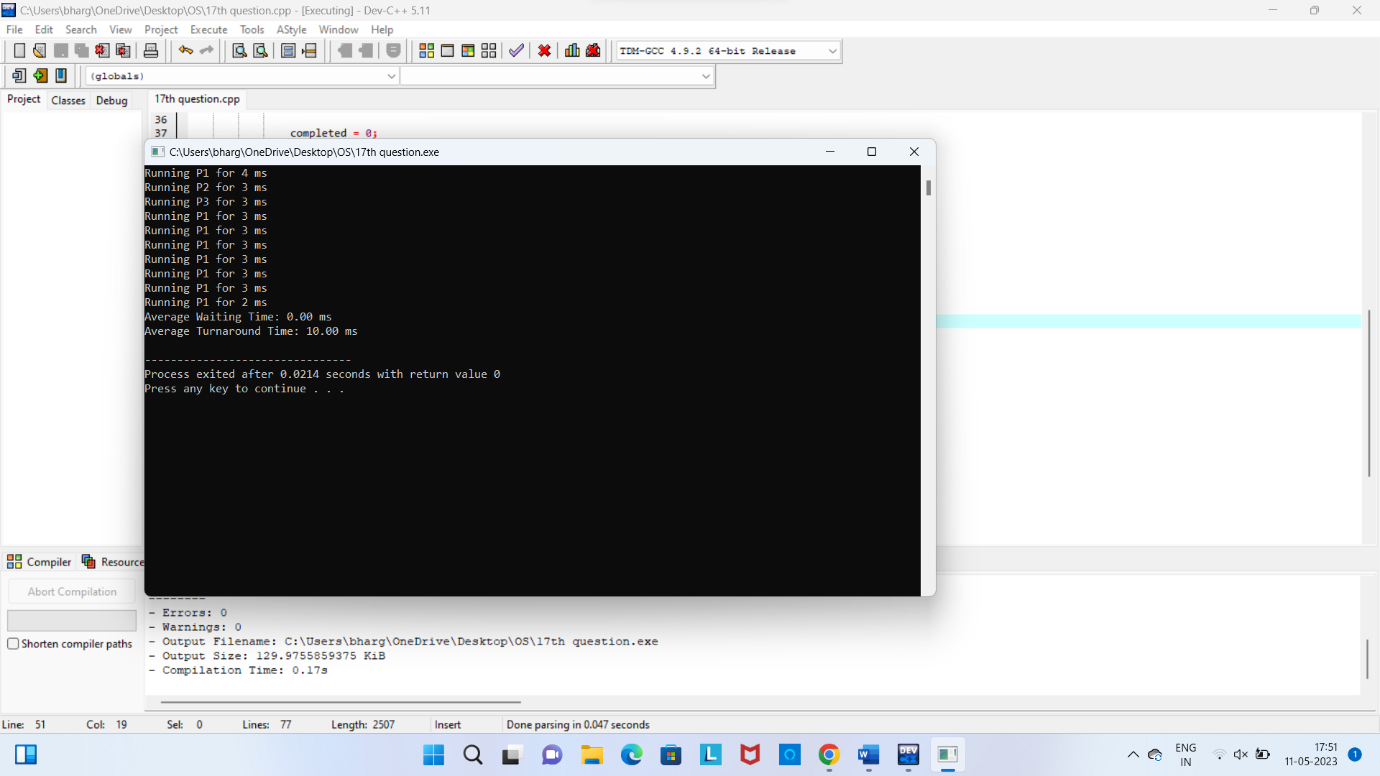
int quantum\_time = 4;

round\_robin(processes, num\_processes, quantum\_time);

return 0;

}

Output



18. #include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#define BUFFER\_SIZE 5 // Size of the buffer

int buffer[BUFFER\_SIZE]; // Shared buffer

int count = 0; // Number of items in the buffer

pthread\_mutex\_t mutex = PTHREAD\_MUTEX\_INITIALIZER; // Mutex for buffer access

pthread\_cond\_t producer\_cond = PTHREAD\_COND\_INITIALIZER; // Condition variable for producer

pthread\_cond\_t consumer\_cond = PTHREAD\_COND\_INITIALIZER; // Condition variable for consumer

void \*producer(void \*arg) {

int item = 1;

while (1) {

pthread\_mutex\_lock(&mutex); // Acquire the mutex lock

// Check if the buffer is full

if (count == BUFFER\_SIZE) {

pthread\_cond\_wait(&producer\_cond, &mutex); // Wait until the buffer is not full

}

// Produce an item

buffer[count] = item;

count++;

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

item++;

pthread\_cond\_signal(&consumer\_cond); // Signal the consumer that an item is available

pthread\_mutex\_unlock(&mutex); // Release the mutex lock

}

return NULL;

}

void \*consumer(void \*arg) {

while (1) {

pthread\_mutex\_lock(&mutex); // Acquire the mutex lock

// Check if the buffer is empty

if (count == 0) {

pthread\_cond\_wait(&consumer\_cond, &mutex); // Wait until the buffer is not empty

}

// Consume an item

int item = buffer[count - 1];

count--;

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

pthread\_cond\_signal(&producer\_cond); // Signal the producer that a slot is available

pthread\_mutex\_unlock(&mutex); // Release the mutex lock

}

return NULL;

}

int main() {

pthread\_t producer\_thread, consumer\_thread;

// Create the producer thread

if (pthread\_create(&producer\_thread, NULL, producer, NULL) != 0) {

fprintf(stderr, "Error creating producer thread\n");

return 1;

}

// Create the consumer thread

if (pthread\_create(&consumer\_thread, NULL, consumer, NULL) != 0) {

fprintf(stderr, "Error creating consumer thread\n");

return 1;

}

// Wait for the threads to finish

if (pthread\_join(producer\_thread, NULL) != 0) {

fprintf(stderr, "Error joining producer thread\n");

return 1;

}

if (pthread\_join(consumer\_thread, NULL) != 0) {

fprintf(stderr, "Error joining consumer thread\n");

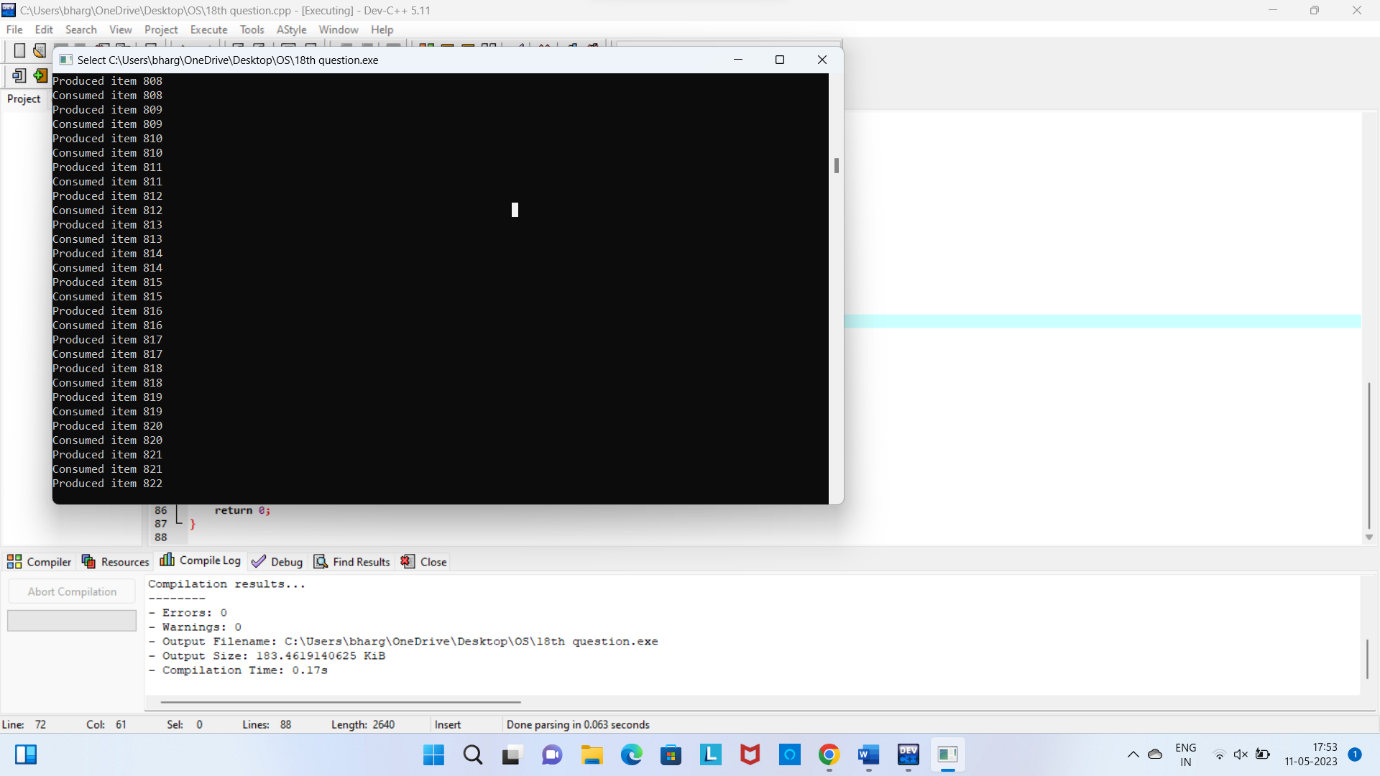
return 1;

}

return 0;

}

Output



19 #include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>

int shared\_data; // Shared memory

sem\_t semaphore; // Semaphore for synchronization

void \*thread1(void \*arg) {

int doubled\_data;

// Wait for the semaphore

sem\_wait(&semaphore);

// Read the shared memory and perform the operation

doubled\_data = shared\_data \* 2;

// Print the result

printf("Thread 1: Doubled data = %d\n", doubled\_data);

// Release the semaphore

sem\_post(&semaphore);

return NULL;

}

void \*thread2(void \*arg) {

int multiplied\_data;

// Wait for the semaphore

sem\_wait(&semaphore);

// Read the shared memory and perform the operation

multiplied\_data = shared\_data \* 5;

// Print the result

printf("Thread 2: Multiplied data = %d\n", multiplied\_data);

// Release the semaphore

sem\_post(&semaphore);

return NULL;

}

int main() {

pthread\_t t1, t2;

// Initialize the semaphore

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

// Read the integer data from the user

printf("Enter an integer value: ");

scanf("%d", &shared\_data);

// Create thread 1

if (pthread\_create(&t1, NULL, thread1, NULL) != 0) {

fprintf(stderr, "Error creating thread 1\n");

return 1;

}

// Create thread 2

if (pthread\_create(&t2, NULL, thread2, NULL) != 0) {

fprintf(stderr, "Error creating thread 2\n");

return 1;

}

// Wait for the threads to finish

if (pthread\_join(t1, NULL) != 0) {

fprintf(stderr, "Error joining thread 1\n");

return 1;

}

if (pthread\_join(t2, NULL) != 0) {

fprintf(stderr, "Error joining thread 2\n");

return 1;

}

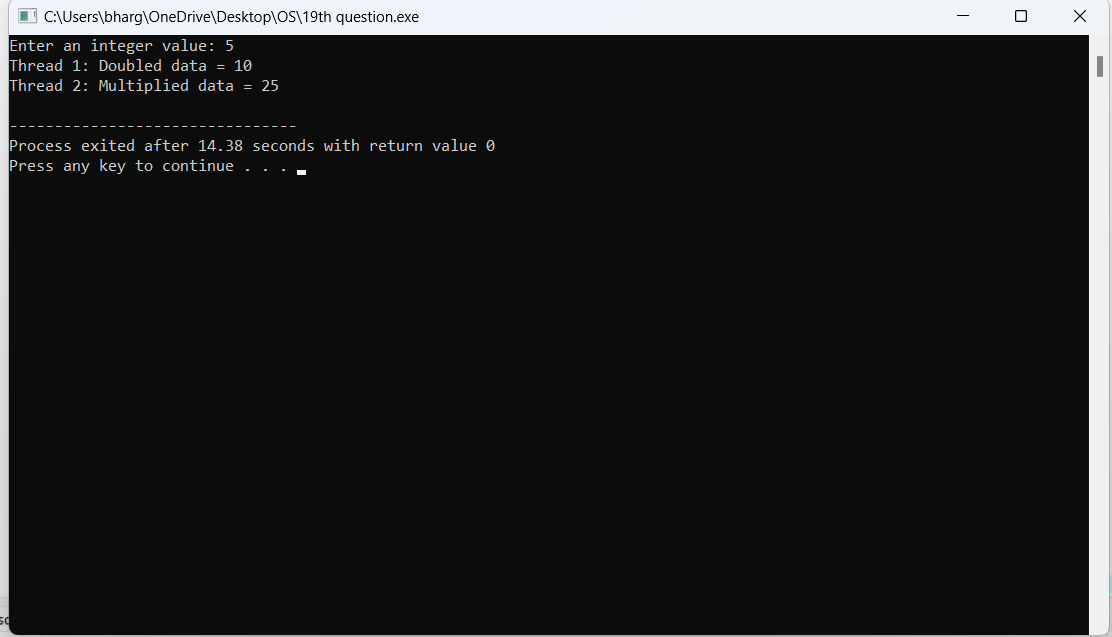
// Destroy the semaphore

sem\_destroy(&semaphore);

return 0;

}

Output



20 #include <stdio.h>

#define MAX\_PARTITIONS 6

#define MAX\_PROCESSES 5

struct MemoryPartition {

int size;

int allocated;

};

struct Process {

int size;

int allocated\_partition;

};

void worst\_fit(struct MemoryPartition memory[], int num\_partitions, struct Process processes[], int num\_processes) {

int i, j;

for (i = 0; i < num\_processes; i++) {

int worst\_index = -1;

for (j = 0; j < num\_partitions; j++) {

if (memory[j].allocated == 0 && memory[j].size >= processes[i].size) {

if (worst\_index == -1 || memory[j].size > memory[worst\_index].size) {

worst\_index = j;

}

}

}

if (worst\_index != -1) {

memory[worst\_index].allocated = 1;

memory[worst\_index].size -= processes[i].size;

processes[i].allocated\_partition = worst\_index;

}

}

}

int main() {

struct MemoryPartition memory[MAX\_PARTITIONS] = {

{300, 0},

{600, 0},

{350, 0},

{200, 0},

{750, 0},

{125, 0}

};

struct Process processes[MAX\_PROCESSES] = {

{115, -1},

{500, -1},

{358, -1},

{200, -1},

{375, -1}

};

int num\_partitions = sizeof(memory) / sizeof(memory[0]);

int num\_processes = sizeof(processes) / sizeof(processes[0]);

worst\_fit(memory, num\_partitions, processes, num\_processes);

printf("Process\t\tSize\tPartition\n");

for (int i = 0; i < num\_processes; i++) {

printf("P%d\t\t%d KB\t", i+1, processes[i].size);

if (processes[i].allocated\_partition != -1) {

printf("%d\n", processes[i].allocated\_partition + 1);

} else {

printf("Not allocated\n");

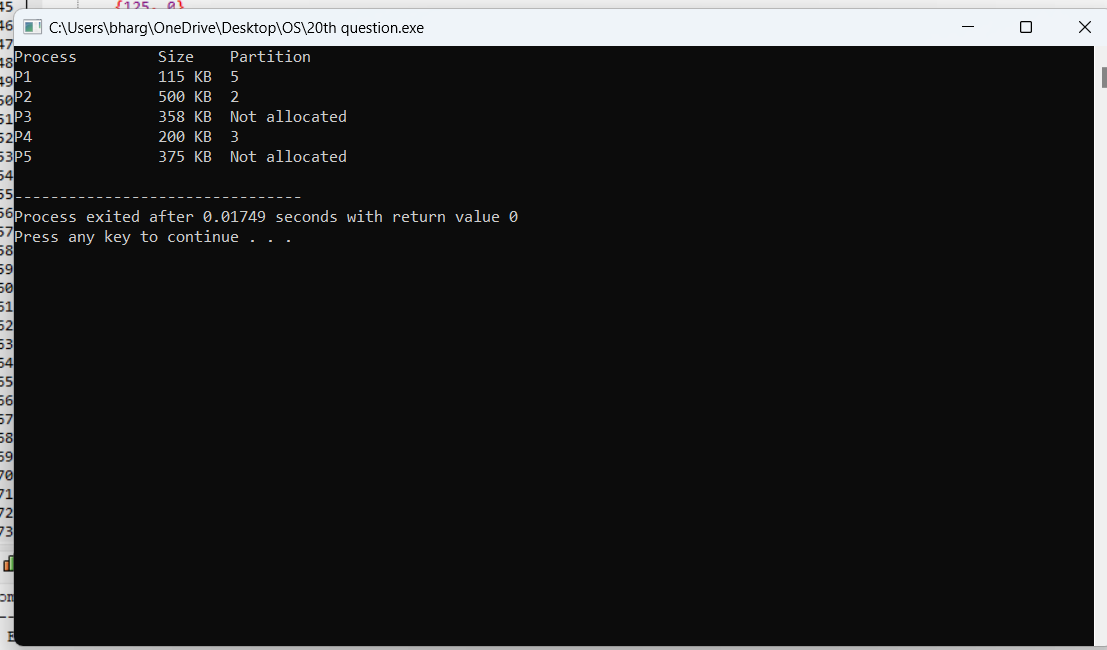
}

}

return 0;

}

Output



21 #include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>

#define NUM\_PHILOSOPHERS 5

enum { THINKING, HUNGRY, EATING } state[NUM\_PHILOSOPHERS];

pthread\_mutex\_t mutex;

sem\_t semaphores[NUM\_PHILOSOPHERS];

void test(int i) {

if (state[i] == HUNGRY

&& state[(i + NUM\_PHILOSOPHERS - 1) % NUM\_PHILOSOPHERS] != EATING

&& state[(i + 1) % NUM\_PHILOSOPHERS] != EATING) {

state[i] = EATING;

sem\_post(&semaphores[i]);

}

}

void pickup\_forks(int i) {

pthread\_mutex\_lock(&mutex);

state[i] = HUNGRY;

test(i);

pthread\_mutex\_unlock(&mutex);

sem\_wait(&semaphores[i]);

}

void return\_forks(int i) {

pthread\_mutex\_lock(&mutex);

state[i] = THINKING;

test((i + NUM\_PHILOSOPHERS - 1) % NUM\_PHILOSOPHERS);

test((i + 1) % NUM\_PHILOSOPHERS);

pthread\_mutex\_unlock(&mutex);

}

void \*philosopher(void \*arg) {

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

while (1) {

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

sleep(rand() % 3);

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

pickup\_forks(id);

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

sleep(rand() % 3);

printf("Philosopher %d is returning forks\n", id);

return\_forks(id);

}

return NULL;

}

int main() {

pthread\_t threads[NUM\_PHILOSOPHERS];

int ids[NUM\_PHILOSOPHERS];

int i;

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

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

state[i] = THINKING;

sem\_init(&semaphores[i], 0, 0);

ids[i] = i;

}

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

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

}

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

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

}

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

sem\_destroy(&semaphores[i]);

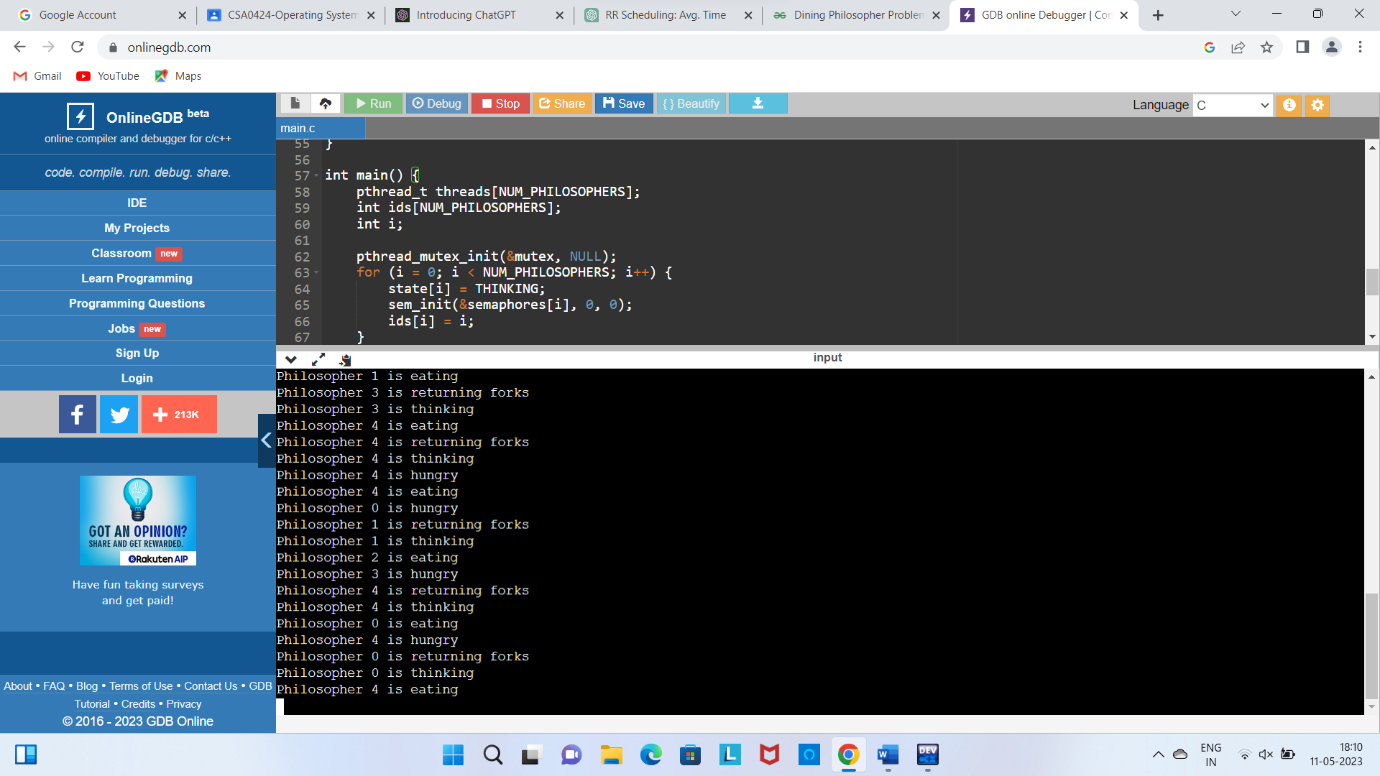
}

pthread\_mutex\_destroy(&mutex);

return 0;

}

Output



22. #include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define MAX\_DIRS 10

#define MAX\_FILES 10

#define MAX\_NAME\_LEN 20

struct File {

char name[MAX\_NAME\_LEN];

int size;

};

struct Directory {

char name[MAX\_NAME\_LEN];

struct File files[MAX\_FILES];

int num\_files;

};

struct User {

char name[MAX\_NAME\_LEN];

struct Directory dir;

};

struct User users[MAX\_DIRS];

int num\_users = 0;

void create\_user(char \*name) {

if (num\_users >= MAX\_DIRS) {

printf("Error: maximum number of users reached\n");

return;

}

struct User user;

strcpy(user.name, name);

strcpy(user.dir.name, name);

user.dir.num\_files = 0;

users[num\_users++] = user;

printf("User %s created\n", name);

}

struct User \*find\_user(char \*name) {

for (int i = 0; i < num\_users; i++) {

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

return &users[i];

}

}

return NULL;

}

void create\_file(char \*username, char \*filename, int size) {

struct User \*user = find\_user(username);

if (user == NULL) {

printf("Error: user %s not found\n", username);

return;

}

if (user->dir.num\_files >= MAX\_FILES) {

printf("Error: maximum number of files reached for user %s\n", username);

return;

}

struct File file;

strcpy(file.name, filename);

file.size = size;

user->dir.files[user->dir.num\_files++] = file;

printf("File %s created for user %s\n", filename, username);

}

void print\_directory(struct Directory dir) {

printf("Directory %s:\n", dir.name);

for (int i = 0; i < dir.num\_files; i++) {

printf("- %s (%d KB)\n", dir.files[i].name, dir.files[i].size);

}

}

void print\_users() {

printf("Users:\n");

for (int i = 0; i < num\_users; i++) {

print\_directory(users[i].dir);

}

}

int main() {

create\_user("user1");

create\_user("user2");

create\_user("user3");

create\_file("user1", "file1.txt", 10);

create\_file("user1", "file2.txt", 20);

create\_file("user1", "file3.txt", 30);

create\_file("user2", "file1.txt", 15);

create\_file("user2", "file2.txt", 25);

create\_file("user3", "file1.txt", 12);

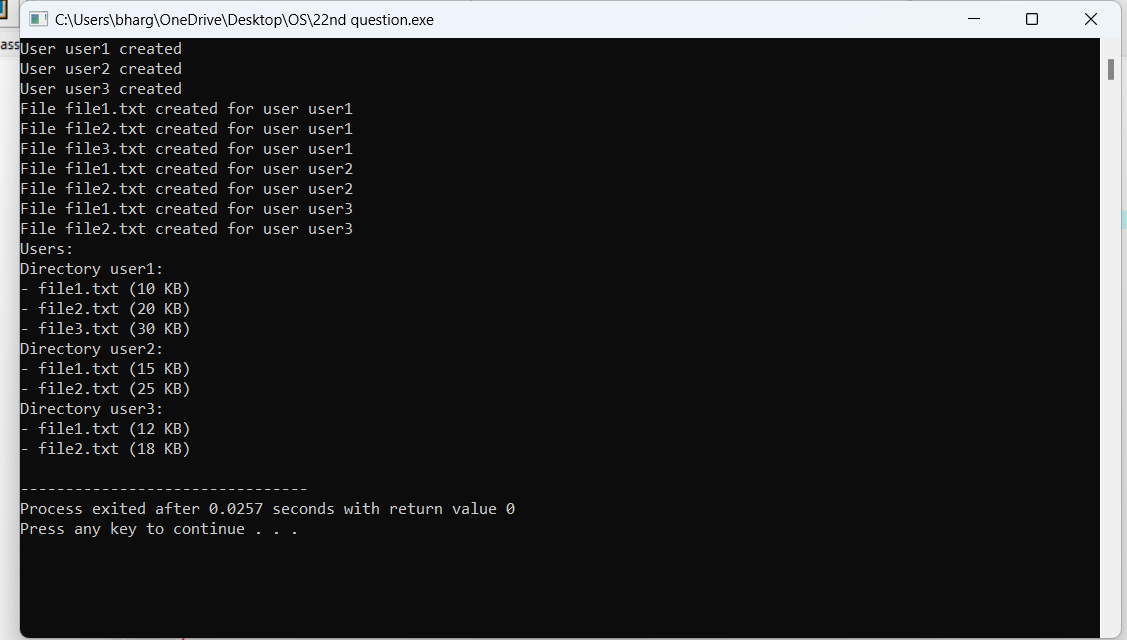
create\_file("user3", "file2.txt", 18);

print\_users();

return 0;

}

Output



23 #include <stdio.h>

#include <stdlib.h>

#include <math.h>

#define MAX\_TRACKS 100

int compare(const void \*a, const void \*b) {

return (\*(int \*)a - \*(int \*)b);

}

int scan(int tracks[], int num\_tracks, int head\_position) {

int i, head\_index = -1;

// Find the index of the head position

for (i = 0; i < num\_tracks; i++) {

if (tracks[i] == head\_position) {

head\_index = i;

break;

}

}

if (head\_index == -1) {

printf("Error: Head position not found in the track list.\n");

return -1;

}

// Sort the tracks in ascending order

qsort(tracks, num\_tracks, sizeof(int), compare);

int total\_movement = 0;

// Start scanning from the head position towards the higher track numbers

for (i = head\_index; i < num\_tracks; i++) {

total\_movement += abs(head\_position - tracks[i]);

head\_position = tracks[i];

}

// Move the head to the maximum track

total\_movement += abs(head\_position - MAX\_TRACKS);

head\_position = MAX\_TRACKS;

// Continue scanning from the maximum track towards the lower track numbers

for (i = num\_tracks - 1; i >= 0; i--) {

total\_movement += abs(head\_position - tracks[i]);

head\_position = tracks[i];

}

return total\_movement;

}

int main() {

int tracks[MAX\_TRACKS] = {55, 58, 60, 70, 18};

int num\_tracks = sizeof(tracks) / sizeof(tracks[0]);

int head\_position = 50; // Initial head position

int average\_movement = scan(tracks, num\_tracks, head\_position) / num\_tracks;

printf("Average head movement: %d\n", average\_movement);

return 0;

}

Output



24 #include <stdio.h>

#define BLOCK\_SIZE 8192 // 8 KB

#define POINTER\_SIZE 4

#define DIRECT\_BLOCKS 12

#define SINGLE\_INDIRECT\_BLOCKS (BLOCK\_SIZE / POINTER\_SIZE)

#define DOUBLE\_INDIRECT\_BLOCKS (BLOCK\_SIZE / POINTER\_SIZE) \* (BLOCK\_SIZE / POINTER\_SIZE)

#define TRIPLE\_INDIRECT\_BLOCKS (BLOCK\_SIZE / POINTER\_SIZE) \* (BLOCK\_SIZE / POINTER\_SIZE) \* (BLOCK\_SIZE / POINTER\_SIZE)

#define MAX\_FILE\_SIZE (DIRECT\_BLOCKS + \

SINGLE\_INDIRECT\_BLOCKS + \

DOUBLE\_INDIRECT\_BLOCKS + \

TRIPLE\_INDIRECT\_BLOCKS) \* BLOCK\_SIZE

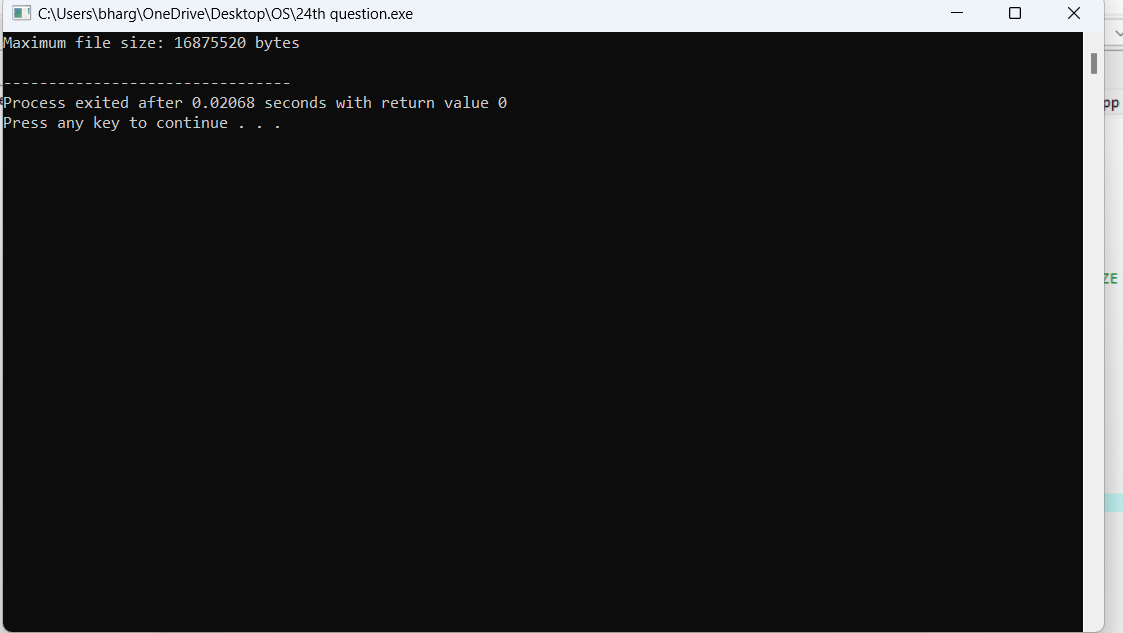
int main() {

printf("Maximum file size: %lu bytes\n", MAX\_FILE\_SIZE);

return 0;

}

Output



25. #include <stdio.h>

#define FILE\_BLOCKS 100

// Contiguous Allocation

int contiguousAllocation(int blockPosition) {

return 1; // Only one disk I/O operation is required for contiguous allocation

}

// Linked Allocation

int linkedAllocation(int blockPosition) {

return 1; // Only one disk I/O operation is required for linked allocation

}

// Indexed Allocation

int indexedAllocation(int blockPosition) {

if (blockPosition == 0 || blockPosition == FILE\_BLOCKS - 1)

return 2; // Two disk I/O operations are required for indexed allocation at the beginning or end

else

return 3; // Three disk I/O operations are required for indexed allocation in the middle

}

int main() {

int blockPosition;

printf("Enter the position of the block (0-%d): ", FILE\_BLOCKS - 1);

scanf("%d", &blockPosition);

printf("Contiguous Allocation: %d disk I/O operations\n", contiguousAllocation(blockPosition));

printf("Linked Allocation: %d disk I/O operations\n", linkedAllocation(blockPosition));

printf("Indexed Allocation: %d disk I/O operations\n", indexedAllocation(blockPosition));

return 0;

}

Output

