INDEPENDENT UNIVERSITY BANGLADESH

Design Of Operating System

(CSE315)

Programming Project

Sec: 02

Date: 18th December 2023

Submitted to

Mohammad Noor Nobi

Senior Lecturer

Department of Computer Science & Engineering

Independent University Bangladesh

Submitted by

Ishtiaq Hossen

Student ID: 1921532

Department of Computer Science & Engineering

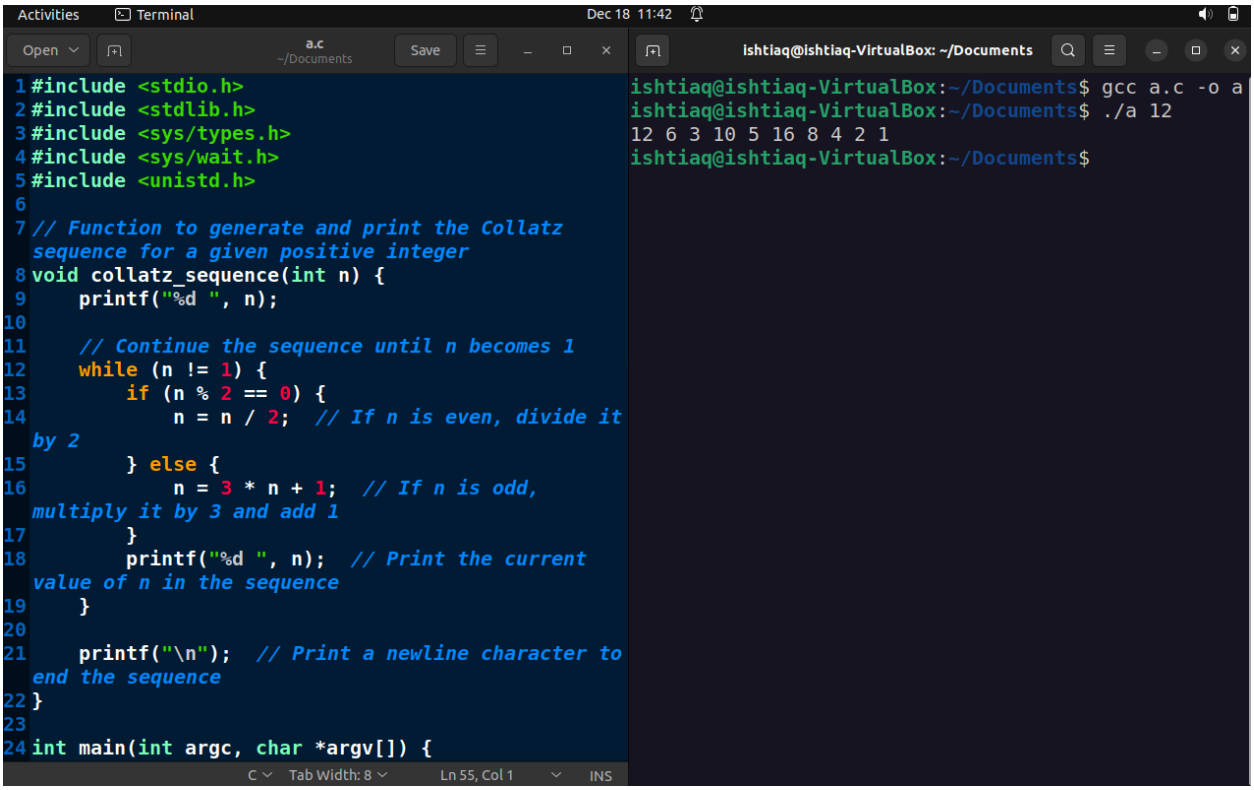
Independent University Bangladesh

Problem A.

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

// Function to generate and print the Collatz sequence for a given positive integer
void collatz_sequence(int n) {
    printf("%d ", n);

    // Continue the sequence until n becomes 1
    while (n != 1) {
        if (n % 2 == 0) {
            n = n / 2;  // If n is even, divide it by 2
        } else {
            n = 3 * n + 1;  // If n is odd, multiply it by 3 and add 1
        }
        printf("%d ", n);  // Print the current value of n in the sequence
    }

    printf("\n");  // Print a newline character to end the sequence
}

int main(int argc, char *argv[]) {
    // Check if the correct number of command-line arguments is provided
    if (argc != 2) {
        printf("Usage: %s <positive_integer>\n", argv[0]);
        return 1;
    }

    // Convert the command-line argument to an integer
    int n = atoi(argv[1]);

    // Check if the provided integer is positive
    if (n <= 0) {
        printf("Please provide a positive integer.\n");
        return 1;
    }

    // Create a child process using fork()
    pid_t pid = fork();
```

```c
    // Check if fork() failed
    if (pid < 0) {
        fprintf(stderr, "Fork failed\n");
        return 1;
    } else if (pid == 0) {  // Child process
        collatz_sequence(n);  // Call the collatz_sequence function in the child
process
    } else {  // Parent process
        wait(NULL);  // Wait for the child process to complete before continuing
    }

    return 0;  // Return 0 to indicate successful execution
}
```

Output From Console(a) :

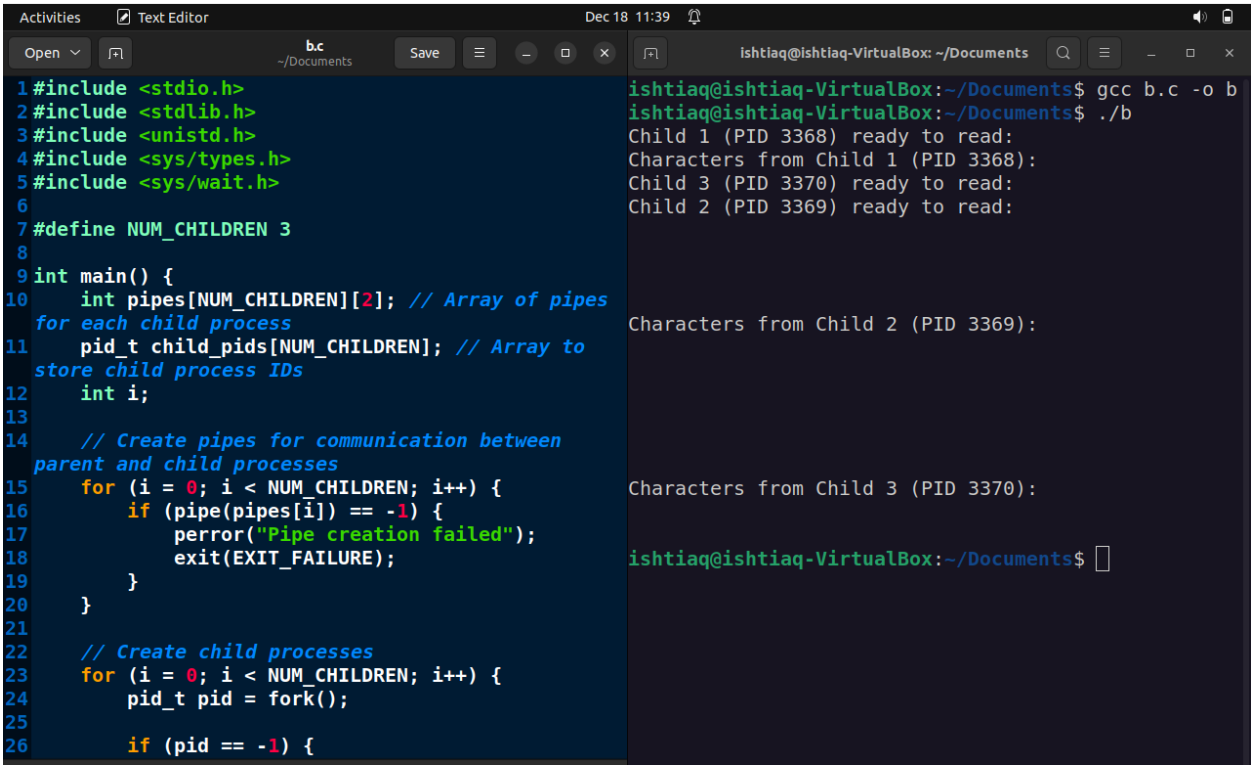


Problem B:

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

#define NUM_CHILDREN 3

int main() {


    int pipes[NUM_CHILDREN][2]; // Array of pipes for each child process
    pid_t child_pids[NUM_CHILDREN]; // Array to store child process IDs
    int i;

    // Create pipes for communication between parent and child processes
    for (i = 0; i < NUM_CHILDREN; i++) {
        if (pipe(pipes[i]) == -1) {
            perror("Pipe creation failed");
            exit(EXIT_FAILURE);
        }
    }


    // Create child processes
    for (i = 0; i < NUM_CHILDREN; i++) {
        pid_t pid = fork();

        if (pid == -1) {
            perror("Fork failed");
            exit(EXIT_FAILURE);
        } else if (pid == 0) { // Child process
            close(pipes[i][0]); // Close reading end in child
            printf("Child %d (PID %d) ready to read:\n", i + 1, getpid());

            char c;
            char prev = '\0';
```

```
                while (read(STDIN_FILENO, &c, 1) > 0) {
                if (c == '\n' && prev == '\n') {
                    break; // Terminate if two consecutive newlines are received
                }
                write(pipes[i][1], &c, 1); // Send character to parent through pipe
                prev = c;
            }

            close(pipes[i][1]); // Close writing end in child
            exit(EXIT_SUCCESS);
        } else { // Parent process
            child_pids[i] = pid;
            close(pipes[i][1]); // Close writing end in parent
        }
    }

    // Parent process reads characters from pipes and outputs them
    for (i = 0; i < NUM_CHILDREN; i++) {
        char buffer;
        printf("Characters from Child %d (PID %d):\n", i + 1, child_pids[i]);
        while (read(pipes[i][0], &buffer, 1) > 0) {
            printf("%c", buffer);
        }
        printf("\n");
        close(pipes[i][0]); // Close reading end in parent
    }

    // Wait for all child processes to terminate
    for (i = 0; i < NUM_CHILDREN; i++) {
        waitpid(child_pids[i], NULL, 0);
    }

    return 0;
}
```

Output From Console(b):



Problem C:

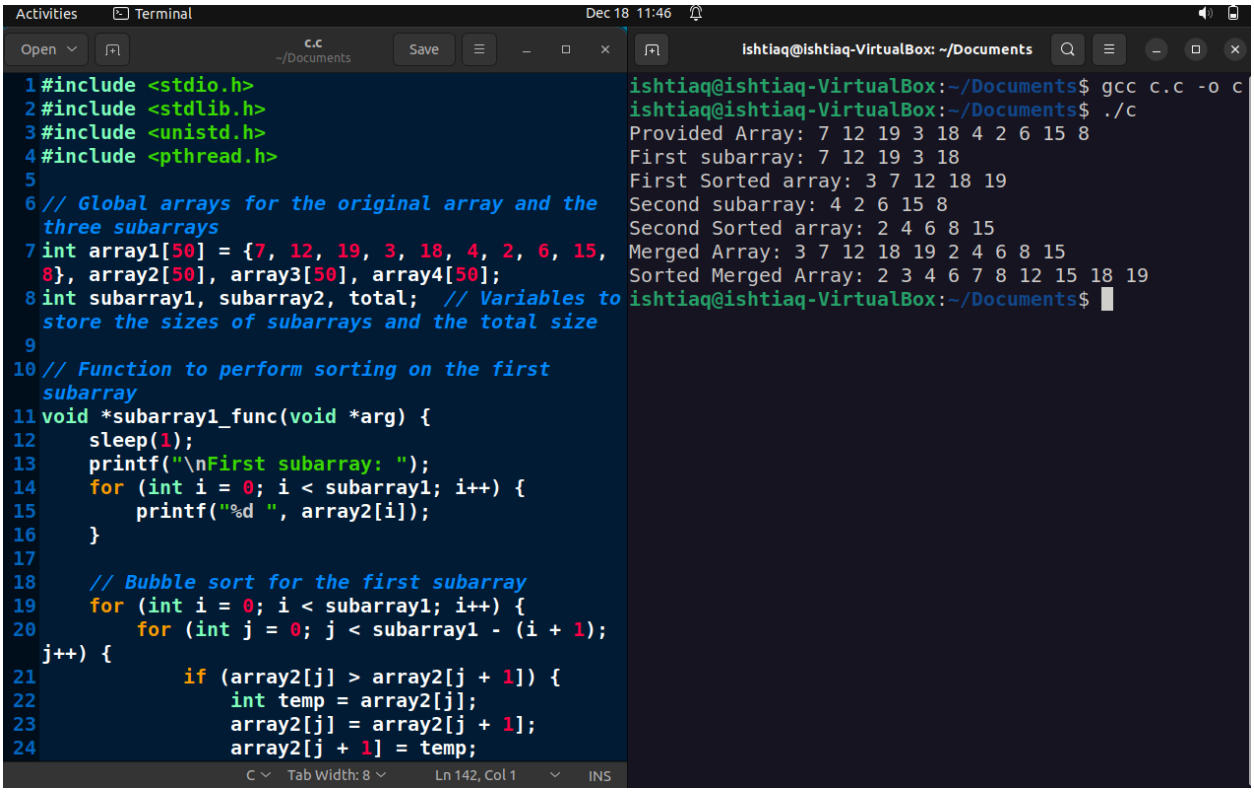
```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

// Global arrays for the original array and the three subarrays
int array1[50] = {7, 12, 19, 3, 18, 4, 2, 6, 15, 8}, array2[50], array3[50],
array4[50];
int subarray1, subarray2, total;  // Variables to store the sizes of subarrays and
the total size

// Function to perform sorting on the first subarray
void *subarray1_func(void *arg) {
    sleep(1);
    printf("\nFirst subarray: ");
    for (int i = 0; i < subarray1; i++) {
        printf("%d ", array2[i]);
    }
```

```c
    // Bubble sort for the first subarray
    for (int i = 0; i < subarray1; i++) {
        for (int j = 0; j < subarray1 - (i + 1); j++) {
            if (array2[j] > array2[j + 1]) {
                int temp = array2[j];
                array2[j] = array2[j + 1];
                array2[j + 1] = temp;
            }
        }
    }

    printf("\nFirst Sorted array: ");
    for (int i = 0; i < subarray1; i++) {
        printf("%d ", array2[i]);
    }
}

// Function to perform sorting on the second subarray
void *subarray2_func(void *arg) {
    sleep(2);
    printf("\nSecond subarray: ");
    for (int i = 0; i < subarray2; i++) {
        printf("%d ", array3[i]);
    }

    // Bubble sort for the second subarray
    for (int i = 0; i < subarray2; i++) {
        for (int j = 0; j < subarray2 - (i + 1); j++) {
            if (array3[j] > array3[j + 1]) {
                int temp = array3[j];
                array3[j] = array3[j + 1];
                array3[j + 1] = temp;
            }
        }
    }

    printf("\nSecond Sorted array: ");
    for (int i = 0; i < subarray2; i++) {
        printf("%d ", array3[i]);
    }
}

// Function to merge and sort the two subarrays
void *merge_func(void *arg) {
    sleep(3);
```

```
    total = subarray1 + subarray2;

    // Copy the first subarray to the merged array
    for (int i = 0; i < subarray1; i++) {
        array4[i] = array2[i];
    }

    int tempsubarray1 = subarray1;

    // Append the second subarray to the merged array
    for (int i = 0; i < subarray2; i++) {
        array4[tempsubarray1] = array3[i];
        tempsubarray1++;
    }

    printf("\nMerged Array: ");
    for (int i = 0; i < total; i++) {
        printf("%d ", array4[i]);
    }

    // Bubble sort for the merged array
    for (int i = 0; i < total; i++) {
        for (int j = 0; j < total - i - 1; j++) {
            if (array4[j + 1] < array4[j]) {
                int temp = array4[j];
                array4[j] = array4[j + 1];
                array4[j + 1] = temp;
            }
        }
    }
}

// Main function
int main(int argc, char const *argv[]) {
    int n = 10;
    pthread_t t1, t2, t3;

    printf("Provided Array: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", array1[i]);
    }

    int j = 0;

    // Divide the original array into two subarrays
```

```
    for (int i = 0; i < n / 2; i++) {
        array2[j] = array1[i];
        j++;
    }

    subarray1 = j;

    int k = 0;

    for (int i = n / 2; i < n; i++) {
        array3[k] = array1[i];
        k++;
    }

    subarray2 = k;

     // Create threads to perform operations on subarrays
    pthread_create(&t1, NULL, subarray1_func, NULL);
    pthread_create(&t2, NULL, subarray2_func, NULL);
    pthread_create(&t3, NULL, merge_func, NULL);

    // Wait for threads to finish execution
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    pthread_join(t3, NULL);

    printf("\nSorted Merged Array: ");
    for (int i = 0; i < total; i++) {
        printf("%d ", array4[i]);
    }

    printf("\n");
    return 0;
}
```

Output From Console(c) :



Problem D:

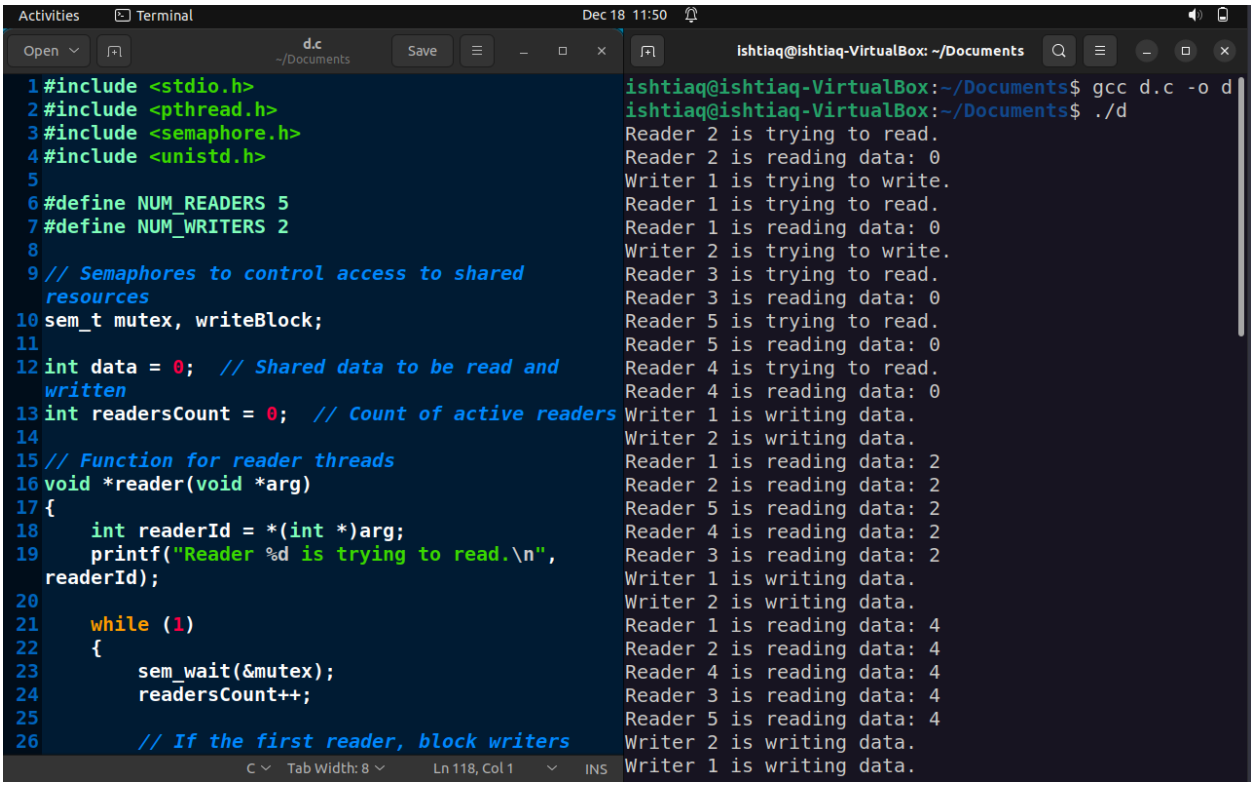
```c
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

#define NUM_READERS 5
#define NUM_WRITERS 2

// Semaphores to control access to shared resources
sem_t mutex, writeBlock;

int data = 0;  // Shared data to be read and written
int readersCount = 0;  // Count of active readers

// Function for reader threads
void *reader(void *arg)
{
    int readerId = *(int *)arg;
```

```
    printf("Reader %d is trying to read.\n", readerId);

    while (1)
    {
        sem_wait(&mutex);
        readersCount++;

        // If the first reader, block writers
        if (readersCount == 1)
        {
            sem_wait(&writeBlock);
        }

        sem_post(&mutex);

        printf("Reader %d is reading data: %d\n", readerId, data);
        usleep(1000000); // Simulate reading by sleeping for 1 second

        sem_wait(&mutex);
        readersCount--;

        // If the last reader, release the writeBlock semaphore
        if (readersCount == 0)
        {
            sem_post(&writeBlock);
        }

        sem_post(&mutex);

        usleep(1000000); // Sleep for 1 second before reading again
    }

    return NULL;
}

// Function for writer threads
void *writer(void *arg)
{
    int writerId = *(int *)arg;
    printf("Writer %d is trying to write.\n", writerId);

    while (1)
    {
        sem_wait(&writeBlock);
```

```
        printf("Writer %d is writing data.\n", writerId);
        data++; // Incrementing data to simulate writing

        sem_post(&writeBlock);

        usleep(2000000); // Sleep for 2 seconds before writing again
    }

    return NULL;
}

int main()
{
    pthread_t readers[NUM_READERS], writers[NUM_WRITERS];
    int readerIds[NUM_READERS], writerIds[NUM_WRITERS];

    // Initialize semaphores
    sem_init(&mutex, 0, 1);
    sem_init(&writeBlock, 0, 1);

    int i;

    // Create reader threads
    for (i = 0; i < NUM_READERS; i++)
    {
        readerIds[i] = i + 1;
        pthread_create(&readers[i], NULL, reader, &readerIds[i]);
    }

    // Create writer threads
    for (i = 0; i < NUM_WRITERS; i++)
    {
        writerIds[i] = i + 1;
        pthread_create(&writers[i], NULL, writer, &writerIds[i]);
    }

    // Wait for reader threads to finish
    for (i = 0; i < NUM_READERS; i++)
    {
        pthread_join(readers[i], NULL);
    }

    // Wait for writer threads to finish
    for (i = 0; i < NUM_WRITERS; i++)
    {
```

```
        pthread_join(writers[i], NULL);
    }

    // Destroy semaphores
    sem_destroy(&mutex);
    sem_destroy(&writeBlock);

    return 0;
}
```

Output From Console(d):



Problem E:

Client.java

```java
import java.io.*;
import java.net.*;
import java.util.*;

// Client class
class Client {

    // Driver code
    public static void main(String[] args) {
        // Establish a connection by providing host and port number
        try (Socket socket = new Socket("localhost", 1234)) {

            // Writing to the server
            PrintWriter out = new PrintWriter(socket.getOutputStream(), true);

            // Reading from the server
            BufferedReader in = new BufferedReader(new
InputStreamReader(socket.getInputStream()));

            // Object of Scanner class for user input
            Scanner sc = new Scanner(System.in);
            String line = null;

            // Loop until the user inputs "exit"
            while (!"exit".equalsIgnoreCase(line)) {
                // Reading input from the user
                line = sc.nextLine();

                // Sending the user input to the server
                out.println(line);
                out.flush();

                // Displaying the server's reply
                System.out.println("Server replied: " + in.readLine());
            }

            // Closing the scanner object
            sc.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
```

```
    }
}
/*Explanation:

The client establishes a connection to a server at the specified host ("localhost") and port number
(1234).

It creates a PrintWriter for writing to the server and a BufferedReader for reading from the server.

It uses a Scanner object (sc) to read input from the user.
The client enters a loop where it continuously reads input from the user, sends it to the server, and
prints the server's reply.
The loop continues until the user inputs "exit."
The Scanner object is closed, and any IOExceptions are caught and printed.

*/
```

Server.java

```
import java.io.*;
import java.net.*;

// Server class
class Server {
    public static void main(String[] args) {
        ServerSocket server = null;

        try {
            // Server is listening on port 1234
            server = new ServerSocket(1234);

            // Running an infinite loop for handling client requests
            while (true) {
                // Socket object to receive incoming client requests
                Socket clientSocket = server.accept();

                // Displaying that a new client is connected to the server
                System.out.println("New client connected " +
clientSocket.getInetAddress().getHostAddress());

                // Create a new thread (ClientHandler) to handle the client
separately
                ClientHandler clientHandler = new ClientHandler(clientSocket);

                // Start the thread to handle the client
```

```
                new Thread(clientHandler).start();
            }
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            if (server != null) {
                try {
                    server.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }

    // ClientHandler class (Handles each client in a separate thread)
    private static class ClientHandler implements Runnable {
        private final Socket clientSocket;

        // Constructor
        public ClientHandler(Socket socket) {
            this.clientSocket = socket;
        }

        public void run() {
            PrintWriter out = null;
            BufferedReader in = null;

            try {
                // Get the output stream of the client
                out = new PrintWriter(clientSocket.getOutputStream(), true);

                // Get the input stream of the client
                in = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()));

                String line;
                while ((line = in.readLine()) != null) {
                    // Print the received message from the client
                    System.out.printf(" Sent from the client: %s\n", line);

                    // Check if the client wants to disconnect
                    if ("exit".equals(line)) {
                        System.out.println("Client Disconnected \n");
                        out.println("You are disconnected \n");
```

```
                    break; // Break the loop if the client wants to disconnect
                } else {
                    // Echo the message back to the client
                    out.println(line);
                }
            }
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            try {
                // Close resources (output stream, input stream, and the client
socket)
                if (out != null) {
                    out.close();
                }
                if (in != null) {
                    in.close();
                }
                clientSocket.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}



/*
Explanation:

    The server listens for client connections on port 1234 in an infinite loop.
    When a new client connects, a new thread (ClientHandler) is created to handle
that client separately.
    The ClientHandler reads messages from the client, prints them, and echoes them
back to the client.
    If the client sends "exit," the server acknowledges the disconnection and breaks
out of the loop to disconnect the client.
    The server continues listening for new client connections.
*/
```

Problem F:

```c
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
#include "buffer.h"

pthread_mutex_t mutex;
sem_t full, empty;
buffer_product buffer[BUFFER_SIZE];
int counter;
pthread_t tid;
pthread_attr_t attr;
void *producer(void *param);
void *consumer(void *param);
int insert_product(buffer_product);
int remove_product(buffer_product*) ;


void initializeData() {
    pthread_mutex_init(&mutex, NULL);
    sem_init(&full, 0, 0);
    sem_init(&empty, 0, BUFFER_SIZE);
    pthread_attr_init(&attr);
    counter = 0;
}


void *producer(void *param) {
    buffer_product product;

    while (1) {
        int rNum = rand() / 100000000;
        sleep(rNum);
        product = rand()%100;
        sem_wait(&empty);
        pthread_mutex_lock(&mutex);
        if (insert_product(product)) {
            fprintf(stderr, " Error occurs in producer report\n");
        }
        else {
            printf("producer produced: %d\n", product);
        }
```

```
        pthread_mutex_unlock(&mutex);
        sem_post(&full);
    }
}


void *consumer(void *param) {
    buffer_product product;
    while (1) {
        int rNum = rand() / 1000000000;
        sleep(rNum);
        sem_wait(&full);
        pthread_mutex_lock(&mutex);
        if (remove_product(&product)) {
            fprintf(stderr, "Error occurs in consumer report\n");
        }
        else {
            printf("consumer consumed: %d\n", product);
        }
        pthread_mutex_unlock(&mutex);
        sem_post(&empty);
    }
}


int insert_product(buffer_product product) {
    if (counter < BUFFER_SIZE) {
        buffer[counter] = product;
        counter++;
        return 0;
    }
    else {
        return -1;
    }
}


int remove_product(buffer_product *product) {
    if (counter > 0) {
        *product = buffer[(counter - 1)];
        counter--;
        return 0;
    }
    else {
        return -1;
```

```
    }
}

int main(int argc, char *argv[]) {
    int i;
     if(argc != 4) {
      fprintf(stderr, "USAGE:./F <INT> <INT> <INT>\n");
      printf("Program is leaving\n");
      exit(0);
   }
    int sleeptime = atoi(argv[1]);
    int numProd = atoi(argv[2]);
    int numCons = atoi(argv[3]);

    initializeData();

    for (i = 0; i < numProd; i++) {
        pthread_create(&tid, &attr, producer, NULL);
    }
    for (i = 0; i < numCons; i++) {
        pthread_create(&tid, &attr, consumer, NULL);
    }
    sleep(sleeptime);
    printf("This program is existing.\n");
    exit(0);
}
```

DiningServer Problem:

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <time.h>

#define NUM_PHILOSOPHERS 5

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;  // Mutex for critical sections
pthread_cond_t conditions[NUM_PHILOSOPHERS];        // Condition variables for
signaling
int forks[NUM_PHILOSOPHERS];                         // Array to represent the state
of forks

// Function to initialize resources
void initialize() {
    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        pthread_cond_init(&conditions[i], NULL);    // Initialize condition variables
        forks[i] = 1;                               // Initially, all forks are
available
    }
}

// Function for a philosopher to pick up forks
void takeForks(int philosopherNumber) {
    pthread_mutex_lock(&mutex);  // Acquire the mutex lock to enter the critical
section

    // While either of the forks is not available, wait
    while (!forks[philosopherNumber] || !forks[(philosopherNumber + 1) %
NUM_PHILOSOPHERS]) {
        pthread_cond_wait(&conditions[philosopherNumber], &mutex);
    }

    forks[philosopherNumber] = 0;                                    // Mark the
left fork as unavailable
    forks[(philosopherNumber + 1) % NUM_PHILOSOPHERS] = 0;           // Mark the
right fork as unavailable
    pthread_mutex_unlock(&mutex);                                     // Release
the mutex lock
}

// Function for a philosopher to return forks
```

```c
void returnForks(int philosopherNumber) {
    pthread_mutex_lock(&mutex);  // Acquire the mutex lock to enter the critical section

    forks[philosopherNumber] = 1;                                    // Mark the left fork as available
    forks[(philosopherNumber + 1) % NUM_PHILOSOPHERS] = 1;           // Mark the right fork as available

    pthread_cond_signal(&conditions[philosopherNumber]);             // Signal the left neighbor
    pthread_cond_signal(&conditions[(philosopherNumber + 1) % NUM_PHILOSOPHERS]);  // Signal the right neighbor
    pthread_mutex_unlock(&mutex);                                     // Release the mutex lock
}

// Function for a philosopher thread
void *philosopher(void *arg) {
    int philosopherNumber = *(int *)arg;  // Get the philosopher's number

    while (1) {
        // Simulate philosopher thinking
        printf("Philosopher %d is thinking\n", philosopherNumber);
        usleep(rand() % 300 + 100);  // Sleep for a random period between 100 and 400 milliseconds

        // Simulate philosopher being hungry and trying to eat
        takeForks(philosopherNumber);

        // Simulate philosopher eating
        printf("Philosopher %d is eating\n", philosopherNumber);
        usleep(rand() % 300 + 100);  // Sleep for a random period between 100 and 400 milliseconds

        // Simulate philosopher finishing eating and returning forks
        returnForks(philosopherNumber);
    }

    return NULL;
}

// Main function
int main() {
    srand(time(NULL));  // Seed for random number generation
```

```
    initialize();  // Initialize resources

    pthread_t philosophers[NUM_PHILOSOPHERS];  // Thread IDs for philosophers
    int philosopherNumbers[NUM_PHILOSOPHERS];

    // Create philosopher threads
    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        philosopherNumbers[i] = i;
        pthread_create(&philosophers[i], NULL, philosopher, &philosopherNumbers[i]);
    }

    // Note: This program runs indefinitely and needs to be terminated manually

    // Wait for philosopher threads to finish
    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        pthread_join(philosophers[i], NULL);
    }

    return 0;
}
```

Output From Console: