Distributed System Lab
Assignment 2

Ashish Verma
20204041
CS - A

Solution: 1)

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

// Global variables
FILE* file;
pthread_mutex_t mutex;

// Function to read from the file
void* read_file(void* arg) {
    pthread_mutex_lock(&mutex);

    // Read from the file
    char buffer[100];
    fseek(file, 0, SEEK_SET);
    fread(buffer, sizeof(char), 100, file);
    printf("Read from file: %s\n", buffer);

    pthread_mutex_unlock(&mutex);
    return NULL;
}

// Function to write to the file
void* write_file(void* arg) {
    pthread_mutex_lock(&mutex);
```

```
    // Write to the file
    char data[] = "Hello, World!";
    fseek(file, 0, SEEK_SET);
    fwrite(data, sizeof(char), sizeof(data), file);

    pthread_mutex_unlock(&mutex);
    return NULL;
}

// Function to update the file
void* update_file(void* arg) {
    pthread_mutex_lock(&mutex);

    // Update the file
    char data[] = "Updated content!";
    fseek(file, 0, SEEK_SET);
    fwrite(data, sizeof(char), sizeof(data), file);

    pthread_mutex_unlock(&mutex);
    return NULL;
}

int main() {
    // Initialize the mutex
    if (pthread_mutex_init(&mutex, NULL) != 0) {
        printf("Mutex initialization failed\n");
        return 1;
    }

    // Open the file
    file = fopen("data.txt", "r+");
```

```
    if (!file) {
        printf("File open failed\n");
        return 1;
    }

    // Create threads for read, write, and update operations
    pthread_t readThread, writeThread, updateThread;

    pthread_create(&readThread, NULL, read_file, NULL);
    pthread_create(&writeThread, NULL, write_file, NULL);
    pthread_create(&updateThread, NULL, update_file, NULL);

    // Wait for threads to finish
    pthread_join(readThread, NULL);
    pthread_join(writeThread, NULL);
    pthread_join(updateThread, NULL);

    // Close the file
    fclose(file);

    // Destroy the mutex
    pthread_mutex_destroy(&mutex);

    return 0;
}
```

Solution: 2)

```
#include <stdio.h>
#include <pthread.h>

// Define two resources
pthread_mutex_t resource1 = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t resource2 = PTHREAD_MUTEX_INITIALIZER;

// Thread function for the first thread
void* thread1_function(void* arg) {
    printf("Thread 1: Attempting to lock resource 1...\n");
    pthread_mutex_lock(&resource1);
    printf("Thread 1: Resource 1 locked.\n");

    // Introduce a delay to increase the chances of deadlock
    sleep(1);

    printf("Thread 1: Attempting to lock resource 2...\n");
    pthread_mutex_lock(&resource2);
    printf("Thread 1: Resource 2 locked.\n");

    // Critical section for Thread 1

    pthread_mutex_unlock(&resource2);
    printf("Thread 1: Resource 2 unlocked.\n");

    pthread_mutex_unlock(&resource1);
```

```
    printf("Thread 1: Resource 1 unlocked.\n");

    return NULL;
}

// Thread function for the second thread
void* thread2_function(void* arg) {
    printf("Thread 2: Attempting to lock resource 2...\n");
    pthread_mutex_lock(&resource2);
    printf("Thread 2: Resource 2 locked.\n");

    // Introduce a delay to increase the chances of deadlock
    sleep(1);

    printf("Thread 2: Attempting to lock resource 1...\n");
    pthread_mutex_lock(&resource1);
    printf("Thread 2: Resource 1 locked.\n");

    // Critical section for Thread 2

    pthread_mutex_unlock(&resource1);
    printf("Thread 2: Resource 1 unlocked.\n");

    pthread_mutex_unlock(&resource2);
    printf("Thread 2: Resource 2 unlocked.\n");

    return NULL;
}

int main() {
    pthread_t thread1, thread2;
```

```
    // Create two threads
    pthread_create(&thread1, NULL, thread1_function, NULL);
    pthread_create(&thread2, NULL, thread2_function, NULL);

    // Wait for threads to finish
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    return 0;
}
```

Solution: 3)

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

// Define two resources
pthread_mutex_t resource1 = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t resource2 = PTHREAD_MUTEX_INITIALIZER;

// Function for the first thread
void* thread1_function(void* arg) {
    int retry = 1;

    while (retry) {
        printf("Thread 1: Attempting to lock resource 1...\n");
```

```
        if (pthread_mutex_trylock(&resource1) == 0) {
            printf("Thread 1: Resource 1 locked.\n");

            // Sleep to introduce a delay (simulating work)
            sleep(1);

            printf("Thread 1: Attempting to lock resource 2...\n");

            if (pthread_mutex_trylock(&resource2) == 0) {
                printf("Thread 1: Resource 2 locked.\n");
                retry = 0; // Both resources acquired, exit loop
            } else {
                pthread_mutex_unlock(&resource1); // Release resource 1
                printf("Thread 1: Resource 1 unlocked.\n");
            }
        }
    }

    // Critical section for Thread 1

    pthread_mutex_unlock(&resource2);
    printf("Thread 1: Resource 2 unlocked.\n");

    pthread_mutex_unlock(&resource1);
    printf("Thread 1: Resource 1 unlocked.\n");

    return NULL;
}

// Function for the second thread
void* thread2_function(void* arg) {
```

```
int retry = 1;

while (retry) {
    printf("Thread 2: Attempting to lock resource 2...\n");

    if (pthread_mutex_trylock(&resource2) == 0) {
        printf("Thread 2: Resource 2 locked.\n");

        // Sleep to introduce a delay (simulating work)
        sleep(1);

        printf("Thread 2: Attempting to lock resource 1...\n");

        if (pthread_mutex_trylock(&resource1) == 0) {
            printf("Thread 2: Resource 1 locked.\n");
            retry = 0; // Both resources acquired, exit loop
        } else {
            pthread_mutex_unlock(&resource2); // Release resource 2
            printf("Thread 2: Resource 2 unlocked.\n");
        }
    }
}

// Critical section for Thread 2

pthread_mutex_unlock(&resource1);
printf("Thread 2: Resource 1 unlocked.\n");

pthread_mutex_unlock(&resource2);
printf("Thread 2: Resource 2 unlocked.\n");

return NULL;
```

```
}

int main() {
    pthread_t thread1, thread2;
    // Create two threads
    pthread_create(&thread1, NULL, thread1_function, NULL);
    pthread_create(&thread2, NULL, thread2_function, NULL);
    // Wait for threads to finish
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
    return 0;
}
```