LAB Assignment 9

Operating Systems (UCS-303)

Instructions: The instructor is required to discuss the concept of Multithreading with the students and the students have to implement following.

1. Write a C programs to implement multithreading where first thread calculates the sum of the elements of shared data (int data [10]), another thread finds the maximum value, and the third thread finds the minimum value. The main thread waits for these threads to finish and prints their results.
2. Two threads thread1 and thread2 are updating the common variable inside a critical section. Write a program using semaphore to ensure that only one thread can access the critical section at a time, to prevent the race condition.
3. Write a program in C to create a child process using fork() system call, parent and child shares a variable int VAR = 10; parent and child can execute concurrently, parent increments the value of VAR by 2 and child decrements the value by 2. Synchronize both the process by using semaphore such that child should always execute before parent.
4. Create a program that simulates a simple bank with multiple accounts and multiple clients making deposits (thread1) and withdrawals (thread2) concurrently. The goal is to ensure that account balances remain consistent even with concurrent operations. Use mutex locks to implement this.

Note: Home Assignment

1. Write a solution using semaphore for Producer Consumer Problem.

Solution 1.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define NUM_THREADS 4
int data[]={45, 56, 78, 32, 9, 9, 5};
// Thread function to calculate the sum of data
void* calculate_sum(void* arg) {
                int* thread_id = (int*)arg;
                int sum = 0;
                int i;
                for ( i = 0; i < 6; i++) {
                        sum += data[i];
                }
                sleep(2);
                printf("Thread %d: Sum of data is %d\n", *thread_id, sum);
                pthread_exit(NULL);
}

// Thread function to find the maximum value in data
void* find_max(void* arg) {
                int* thread_id = (int*)arg;
                int max = data[0];
                int i;
                for ( i = 1; i < 6; i++) {
                        if (data[i] > max) {
```

```
                    max = data[i];
                }
            }
            printf("Thread %d: Maximum value in data is %d\n", *thread_id, max);
            pthread_exit(NULL);
}

// Thread function to find the minimum value in data
void* find_min(void* arg) {
                int* thread_id = (int*)arg;
                int min = data[0];
                int i;
                for ( i = 1; i < 6; i++) {
                    if (data[i] < min) {
                        min = data[i];
                    }
                }
                printf("Thread %d: Minimum value in data is %d\n", *thread_id, min);
                pthread_exit(NULL);
}

int main() {
    pthread_t threads[NUM_THREADS];
    int thread_ids[NUM_THREADS], i;
        // Initialize the data with random values
        //data[6] = {45, 56, 78, 32, 9, 5};
       for (i = 0; i < NUM_THREADS; i++) {
         thread_ids[i] = i;
       }
    // Create threads to perform different tasks
    pthread_create(&threads[0], NULL, calculate_sum, &thread_ids[0]);
    pthread_create(&threads[1], NULL, find_max, &thread_ids[1]);
    pthread_create(&threads[2], NULL, find_min, &thread_ids[2]);
    // Main thread waits for these threads to finish
    for ( i = 0; i < 3; i++) {
        pthread_join(threads[i], NULL);
    }
    return 0;
}
```

Solution 2.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
// Define a global variable to be shared by two threads
int common_variable = 10;
// Define a semaphore
sem_t semaphore;
void *Incre(void *arg) {
    int thread_id = *((int *)arg);
        // Wait on the semaphore (Increment it)
        sem_wait(&semaphore);
        int i= common_variable;
        // Critical section: update the common variable
        i += 1;
        sleep(2);
        common_variable =i;
        printf("Thread %d updated common_variable to %d\n", thread_id, common_variable);
       // Signal that we're done with the critical section (increment the semaphore)
       sem_post(&semaphore);
       pthread_exit(NULL);
}

void *Decr(void *arg) {
    int thread_id = *((int *)arg);
        // Wait on the semaphore (decrement it)
        sem_wait(&semaphore);
        // Critical section: update the common variable
        int i= common_variable;
        // Critical section: update the common variable
        i -= 1;
        common_variable =i;
        printf("Thread %d decremented common_variable to %d\n", thread_id, common_variable);
        // Signal that we're done with the critical section (increment the semaphore)
        sem_post(&semaphore);
    pthread_exit(NULL);
}

int main() {
    // Initialize the semaphore with a value of 1
    sem_init(&semaphore, 0, 1);
    pthread_t thread1, thread2;
    int thread_id1 = 1;
    int thread_id2 = 2;

    // Create two threads
    pthread_create(&thread1, NULL, Incre, &thread_id1);
    pthread_create(&thread2, NULL, Decr, &thread_id2);
    // Wait for the threads to finish
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
    // Destroy the semaphore
    sem_destroy(&semaphore);
    printf("Final common_variable value: %d\n", common_variable);
    return 0;
}
```

Solution 3

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#define MSGSIZE 16
#include <semaphore.h>
int VAR = 10;
sem_t semaphore;
int main() {
    char inbuf[MSGSIZE];
    int fd[2];
    sem_init(&semaphore, 0, 0);
    pipe(fd);
    pid_t pid = fork();
    if (pid == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    } else if (pid == 0) {
        // Child process
sleep(2);
        VAR=VAR-2;
        printf("Child Process: Shared Variable = %d\n", VAR);
        sem_post(&semaphore);
        write(fd[1], "1" , 2);
    } else {
        // Parent process
        read(fd[0], inbuf, MSGSIZE);
        sem_init(&semaphore, 0, atoi(inbuf));
        sem_wait(&semaphore);
        VAR = VAR+2;
        printf("Parent Process: Shared Variable = %d\n", VAR);
       }
    sem_destroy(&semaphore);
    return 0;
}
```

Solution 4

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define NUM_Client 5
int account = 500;
pthread_mutex_t lock;
// Thread function to calculate the sum of data
void* Deposit(void* arg) {
    int* client_id = (int*)arg;
    int amount;
    int x;
     pthread_mutex_lock(&lock);
     printf("\n Enter amount to deposit");
     scanf("%d",&amount);
     x=account;
     x = x + amount;
     sleep(3);
     account=x;
     pthread_mutex_unlock(&lock);
              printf("Client %d deposited: %d, Amount after deposit is %d\n", *client_id, amount, account);

              pthread_exit(NULL);
}

void* Withdraw(void* arg) {
    int* client_id = (int*)arg;
    int amount,x;
    pthread_mutex_lock(&lock);
     printf("\n Enter amount to withdraw");
     scanf("%d",&amount);
     x=account;
     x = x - amount;
     account=x;
     pthread_mutex_unlock(&lock);
              printf("Client %d withdraw: %d, Amount after withdrawl is %d\n", *client_id, amount,
account);
     pthread_exit(NULL);
}

int main() {
    pthread_t threads[NUM_Client];
    pthread_mutex_init(&lock, NULL);
    int client_ids[NUM_Client], i;
        // Initialize the data with random values
        //data[6] = {45, 56, 78, 32, 9, 5};
       for (i = 0; i < NUM_Client; i++) {
         client_ids[i] = i;
       }
    // Create threads to perform different tasks
    for(i=0;i<NUM_Client;i++){
             if(i%2==0)
             pthread_create(&threads[0], NULL, Deposit, &client_ids[i]);
             else{
             pthread_create(&threads[1], NULL, Withdraw, &client_ids[1]);
                          }
    }
    // Main thread waits for these threads to finish
    for ( i = 0; i < 3; i++) {
        pthread_join(threads[i], NULL);
    }


    return 0;
}
```

Solution4:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <stdlib.h>
#define NUM_Client 5
int account[9] = {550, 450, 300, 700, 500, 600, 400, 800, 350};
pthread_mutex_t lock;
// Thread function to deposit
void* Deposit(void* arg) {
    int* client_id = (int*)arg;
    int amount, x, k;
    k=rand() % (10);
    printf("\n Enter amount to deposit");
    scanf("%d",&amount);
    pthread_mutex_lock(&lock);
     x=account[k];
     x = x + amount;
     //sleep(3);
     account[k]=x;
    pthread_mutex_unlock(&lock);
            printf("Client %d deposited: %d, Amount after deposit is %d\n", *client_id, amount, account[k]);
    pthread_exit(NULL);
}


//Method to withdraw the money
void* Withdraw(void* arg) {
    int* client_id = (int*)arg;
    int amount,x,k;
    k=rand() % (10);
    printf("\n Enter amount to withdraw");
    scanf("%d",&amount);
    pthread_mutex_lock(&lock);
     x=account[k];
     x = x - amount;
     account[k]=x;
    pthread_mutex_unlock(&lock);
        printf("Client %d withdraw: %d, Amount after withdrawl is %d\n", *client_id, amount, account[k]);
    pthread_exit(NULL);
}

int main() {
    pthread_t threads[NUM_Client];
    pthread_mutex_init(&lock, NULL);
    int client_ids[NUM_Client], i;
    for (i = 0; i < NUM_Client; i++) {
       client_ids[i] = i;
    }
    for(i=0;i<NUM_Client;i++){
        if(i%2==0){
                pthread_create(&threads[i], NULL, Deposit, &client_ids[i]);}
        else{
                pthread_create(&threads[i], NULL, Withdraw, &client_ids[i]);
                }
    }
    // Main thread waits for these threads to finish
    for ( i = 0; i < NUM_Client; i++) {
        pthread_join(threads[i], NULL);
    }
    return 0;
```

```
}
```