Name: Ashmit Thawait

Roll No: 102203790

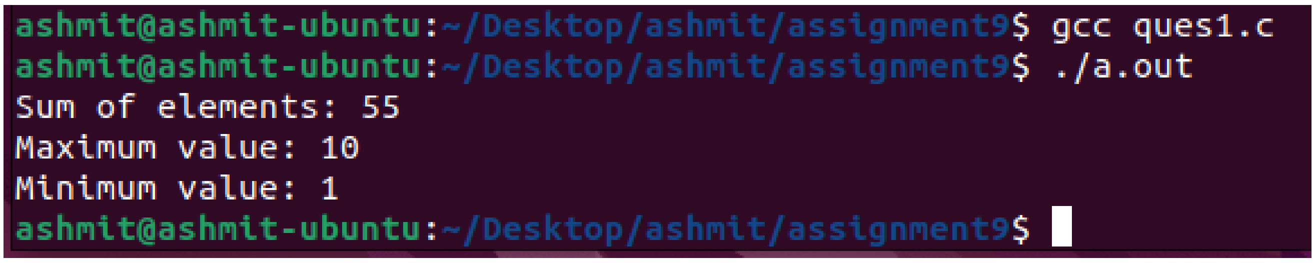
Group: 2CO-17

Lab Assignment 9

Operating Systems (UCS303)

Ques1) 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.

OUTPUT:



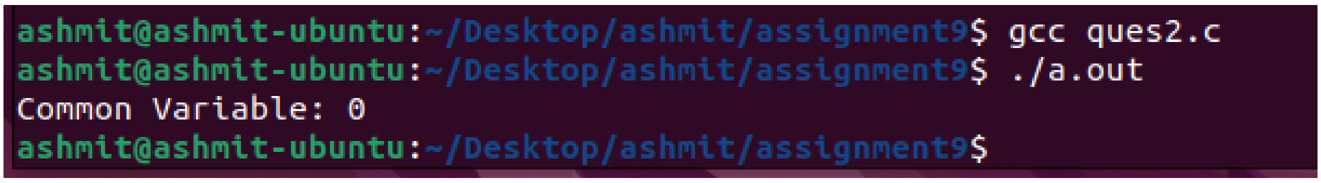
```
ashmit@ashmit-ubuntu:~/Desktop/ashmit/assignment9$ gcc ques1.c
ashmit@ashmit-ubuntu:~/Desktop/ashmit/assignment9$ ./a.out
Sum of elements: 55
Maximum value: 10
Minimum value: 1
ashmit@ashmit-ubuntu:~/Desktop/ashmit/assignment9$ 
```

CODE:

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

#define NUM_THREADS 3
#define ARRAY_SIZE 10

int data[ARRAY_SIZE] = {2, 5, 9, 1, 7, 3, 8, 6, 4, 10}; // Sample data

int sum_result = 0;
int max_result = 0;
int min_result = 0;

// Function to calculate the sum of elements
void *calculate_sum(void *arg) {
    int sum = 0;
    for (int i = 0; i < ARRAY_SIZE; i++) {
        sum += data[i];
    }
    sum_result = sum;
    pthread_exit(NULL);
}

// Function to find the maximum value
void *find_max(void *arg) {
    int max = data[0];
    for (int i = 1; i < ARRAY_SIZE; i++) {
        if (data[i] > max) {
            max = data[i];
        }
    }
    max_result = max;
    pthread_exit(NULL);
}

// Function to find the minimum value
void *find_min(void *arg) {
    int min = data[0];
    for (int i = 1; i < ARRAY_SIZE; i++) {
        if (data[i] < min) {
            min = data[i];
        }
    }
    min_result = min;
    pthread_exit(NULL);
}

int main() {
    pthread_t threads[NUM_THREADS];

    // Create threads for sum, max, and min
    pthread_create(&threads[0], NULL, calculate_sum, NULL);
    pthread_create(&threads[1], NULL, find_max, NULL);
    pthread_create(&threads[2], NULL, find_min, NULL);

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

    // Print results
    printf("Sum of elements: %d\n", sum_result);
    printf("Maximum value: %d\n", max_result);
    printf("Minimum value: %d\n", min_result);

    return 0;
}
```

Ques2) 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.

Output:



```
ashmit@ashmit-ubuntu:~/Desktop/ashmit/assignment9$ gcc ques2.c
ashmit@ashmit-ubuntu:~/Desktop/ashmit/assignment9$ ./a.out
Common Variable: 0
ashmit@ashmit-ubuntu:~/Desktop/ashmit/assignment9$
```

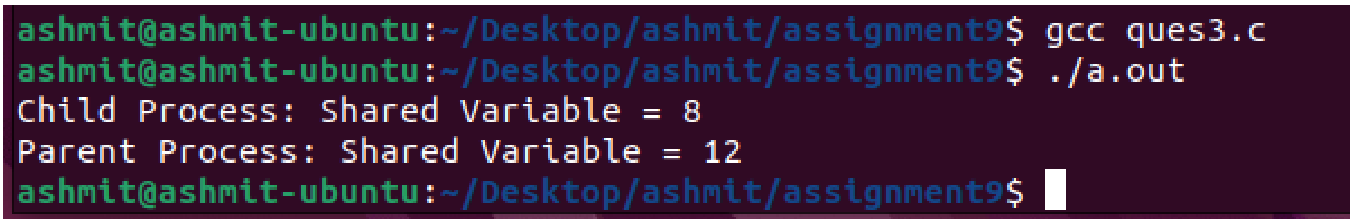
CODE:

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

// Define a global variable and a semaphore to protect it
int commonVariable = 0;
sem_t mutex;

// Function to simulate thread1
void *thread1(void *arg) {
   for (int i = 0; i < 10000; i++) {
      sem_wait(&mutex); // Wait for access to the critical section
      commonVariable++;
      sem_post(&mutex); // Release access to the critical section
   }
   pthread_exit(NULL);
}

// Function to simulate thread2
void *thread2(void *arg) {
   for (int i = 0; i < 10000; i++) {
      sem_wait(&mutex); // Wait for access to the critical section
      commonVariable--;
      sem_post(&mutex); // Release access to the critical section
   }
   pthread_exit(NULL);
}

int main() {
   pthread_t t1, t2;
```

```
    // Initialize the semaphore
    sem_init(&mutex, 0, 1);

    // Create thread1 and thread2
    pthread_create(&t1, NULL, thread1, NULL);
    pthread_create(&t2, NULL, thread2, NULL);

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

    // Print the final value of the common variable
    printf("Common Variable: %d\n", commonVariable);

    // Destroy the semaphore
    sem_destroy(&mutex);

    return 0;
}
```

Ques3) 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.

Output:



```
ashmit@ashmit-ubuntu:~/Desktop/ashmit/assignment9$ gcc ques3.c
ashmit@ashmit-ubuntu:~/Desktop/ashmit/assignment9$ ./a.out
Child Process: Shared Variable = 8
Parent Process: Shared Variable = 12
ashmit@ashmit-ubuntu:~/Desktop/ashmit/assignment9$ 
```

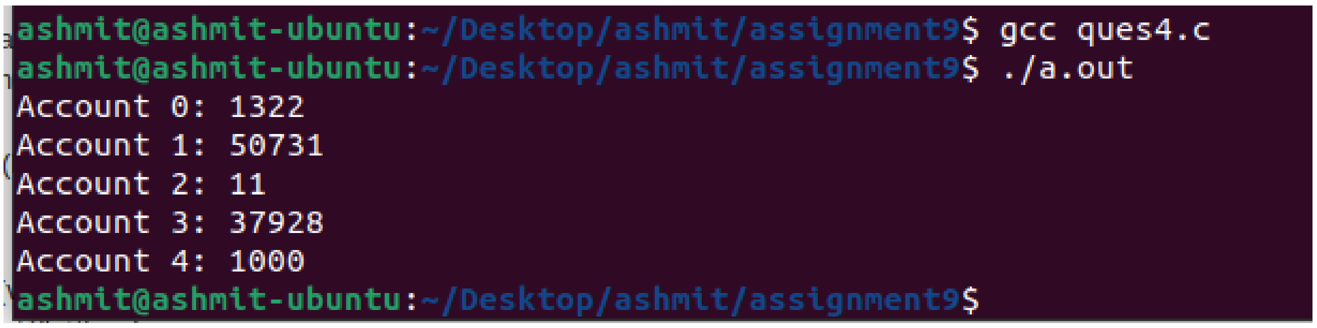
CODE:

```
#include <stdio.h>

#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <semaphore.h>

#define MSGSIZE 16


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
            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;
        }

}
```

Ques4) 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.

Output:



```
ashmit@ashmit-ubuntu:~/Desktop/ashmit/assignment9$ gcc ques4.c
ashmit@ashmit-ubuntu:~/Desktop/ashmit/assignment9$ ./a.out
Account 0: 1322
Account 1: 50731
Account 2: 11
Account 3: 37928
Account 4: 1000
ashmit@ashmit-ubuntu:~/Desktop/ashmit/assignment9$
```

Code:

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

#define NUM_ACCOUNTS 5
#define NUM_CLIENTS 4

int accounts[NUM_ACCOUNTS] = {1000, 1000, 1000, 1000, 1000};
pthread_mutex_t account_mutex[NUM_ACCOUNTS];

void* deposit(void* arg) {
   int account = *((int*)arg);
   for (int i = 0; i < 1000; i++) {
      int amount = rand() % 100;
      pthread_mutex_lock(&account_mutex[account]);
      accounts[account] += amount;
      pthread_mutex_unlock(&account_mutex[account]);
   }
   pthread_exit(NULL);
}

void* withdraw(void* arg) {
   int account = *((int*)arg);
   for (int i = 0; i < 1000; i++) {
      int amount = rand() % 100;
      pthread_mutex_lock(&account_mutex[account]);
      if (accounts[account] >= amount) {
         accounts[account] -= amount;
      }
      pthread_mutex_unlock(&account_mutex[account]);
   }
   pthread_exit(NULL);
}

int main() {
   pthread_t threads[NUM_CLIENTS * 2];
   int client_args[NUM_CLIENTS * 2];

   // Initialize mutexes for each account
   for (int i = 0; i < NUM_ACCOUNTS; i++) {
      pthread_mutex_init(&account_mutex[i], NULL);
   }

   // Create threads for deposits and withdrawals
   for (int i = 0; i < NUM_CLIENTS; i++) {
      client_args[i] = i;
```

```
        pthread_create(&threads[i], NULL, deposit, &client_args[i]);
        client_args[i + NUM_CLIENTS] = i;
        pthread_create(&threads[i + NUM_CLIENTS], NULL, withdraw, &client_args[i + NUM_CLIENTS]);
    }

    // Wait for all threads to finish
    for (int i = 0; i < NUM_CLIENTS * 2; i++) {
        pthread_join(threads[i], NULL);
    }

    // Print account balances
    for (int i = 0; i < NUM_ACCOUNTS; i++) {
        printf("Account %d: %d\n", i, accounts[i]);
    }

    // Destroy mutexes
    for (int i = 0; i < NUM_ACCOUNTS; i++) {
        pthread_mutex_destroy(&account_mutex[i]);
    }

    return 0;
}
```