Damian Franco

Meiling Traeger

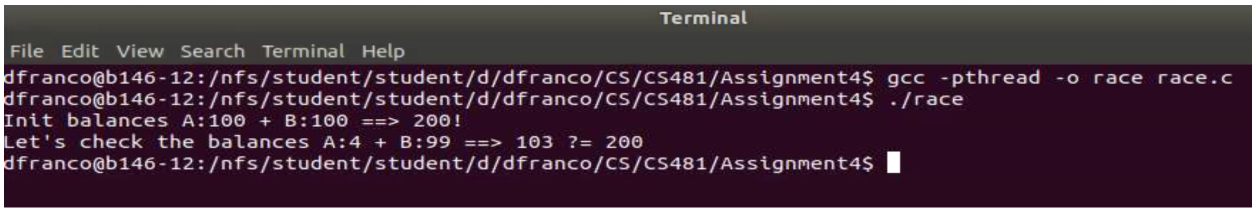
CS 481-003

Programming Assignment #4

1) For number 1, we were asked to run the C program (*race.c*) about 20-40 times and observe the output. We observed that the outputs are very random with no real consistency to them. Randomization of the balances occurs within the *MakeTransactions()* function call. The function call will assign the two bank values to a random integer by utilizing the C function *rand()* and by creating and manipulating temporary integers. Most of the output is not out of the ordinary, but it seems to be erroneous because the comparison statement within the last printf "*=? 200*" implies that both integers should have the sum of 200, but it does not. If any errors are found within this program, they mostly lie with the way the program operates. Possible race conditions may be occurring here which can cause errors in the scheduling and timing of the current threads and even possibly lead to errors within the program's behavior. We believe race conditions occur in this program because the shared data seems to be getting accessed at the same time and leads to some mysterious outcomes when assigning new values to the shared memory. To counteract that, a critical section must be reached, which is what we will be trying to achieve in question 2.

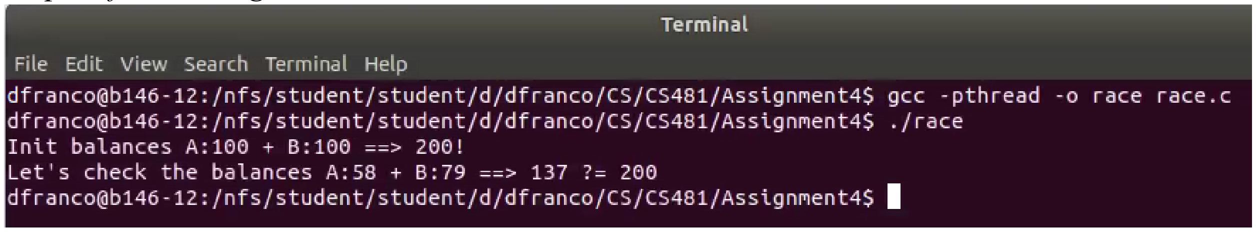
Output Screenshots:

Output after running **1** *time:*



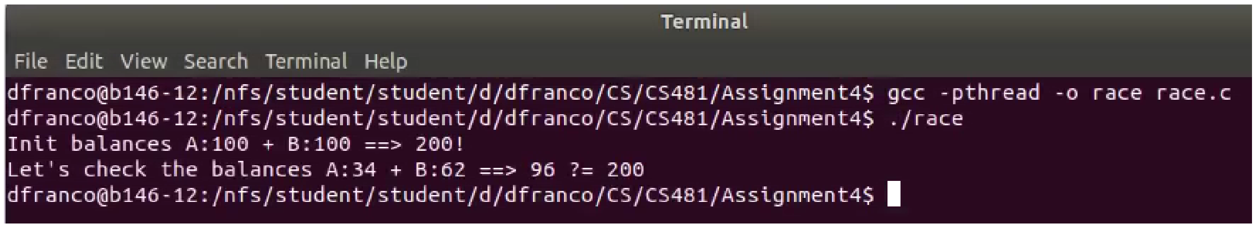
```
dfranco@b146-12:/nfs/student/student/d/dfranco/CS/CS481/Assignment4$ gcc -pthread -o race race.c
dfranco@b146-12:/nfs/student/student/d/dfranco/CS/CS481/Assignment4$ ./race
Init balances A:100 + B:100 ==> 200!
Let's check the balances A:4 + B:99 ==> 103 ?= 200
dfranco@b146-12:/nfs/student/student/d/dfranco/CS/CS481/Assignment4$
```

Output after running **5** *times:*



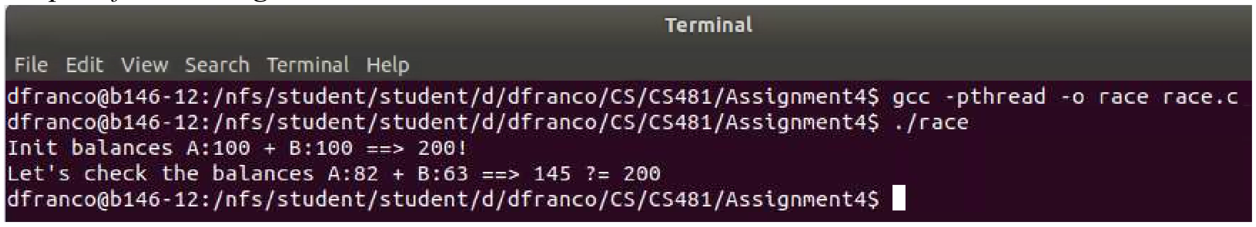
```
dfranco@b146-12:/nfs/student/student/d/dfranco/CS/CS481/Assignment4$ gcc -pthread -o race race.c
dfranco@b146-12:/nfs/student/student/d/dfranco/CS/CS481/Assignment4$ ./race
Init balances A:100 + B:100 ==> 200!
Let's check the balances A:58 + B:79 ==> 137 ?= 200
dfranco@b146-12:/nfs/student/student/d/dfranco/CS/CS481/Assignment4$
```

Output after running **10** *times:*



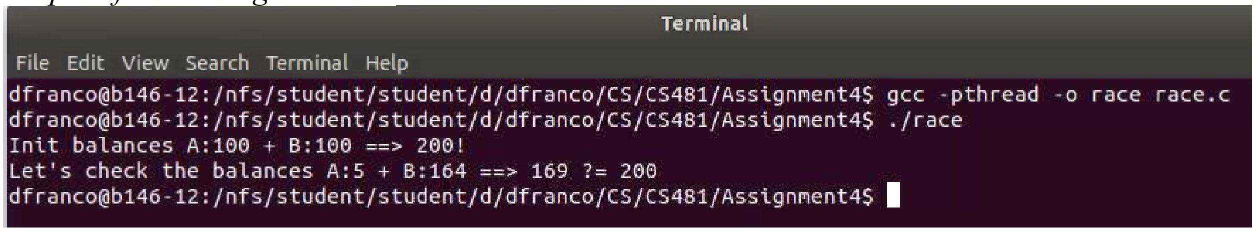
```
dfranco@b146-12:/nfs/student/student/d/dfranco/CS/CS481/Assignment4$ gcc -pthread -o race race.c
dfranco@b146-12:/nfs/student/student/d/dfranco/CS/CS481/Assignment4$ ./race
Init balances A:100 + B:100 ==> 200!
Let's check the balances A:34 + B:62 ==> 96 ?= 200
dfranco@b146-12:/nfs/student/student/d/dfranco/CS/CS481/Assignment4$
```

Output after running **20** *times:*



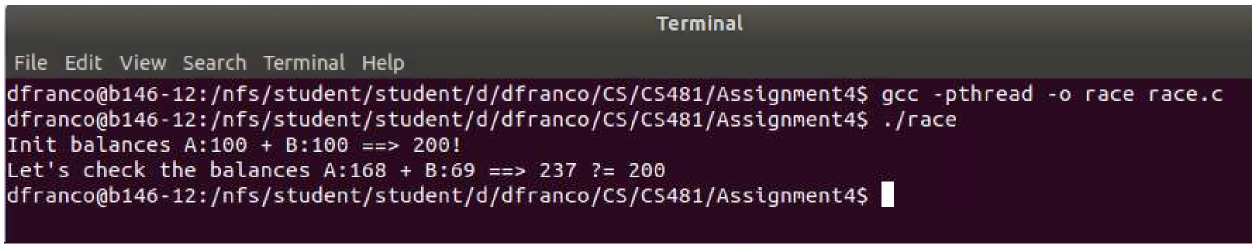
```
dfranco@b146-12:/nfs/student/student/d/dfranco/CS/CS481/Assignment4$ gcc -pthread -o race race.c
dfranco@b146-12:/nfs/student/student/d/dfranco/CS/CS481/Assignment4$ ./race
Init balances A:100 + B:100 ==> 200!
Let's check the balances A:82 + B:63 ==> 145 ?= 200
dfranco@b146-12:/nfs/student/student/d/dfranco/CS/CS481/Assignment4$
```

Output after running **30** *times:*



```
dfranco@b146-12:/nfs/student/student/d/dfranco/CS/CS481/Assignment4$ gcc -pthread -o race race.c
dfranco@b146-12:/nfs/student/student/d/dfranco/CS/CS481/Assignment4$ ./race
Init balances A:100 + B:100 ==> 200!
Let's check the balances A:5 + B:164 ==> 169 ?= 200
dfranco@b146-12:/nfs/student/student/d/dfranco/CS/CS481/Assignment4$
```

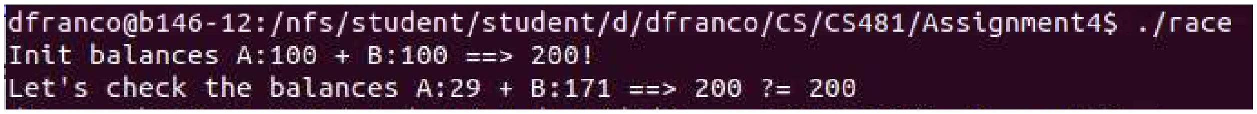
Output after running **40** *times:*



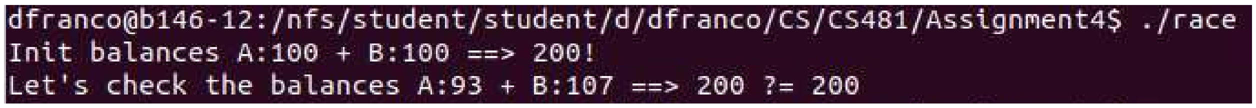
```
dfranco@b146-12:/nfs/student/student/d/dfranco/CS/CS481/Assignment4$ gcc -pthread -o race race.c
dfranco@b146-12:/nfs/student/student/d/dfranco/CS/CS481/Assignment4$ ./race
Init balances A:100 + B:100 ==> 200!
Let's check the balances A:168 + B:69 ==> 237 ?= 200
dfranco@b146-12:/nfs/student/student/d/dfranco/CS/CS481/Assignment4$
```

2) For number 2, we were asked to use the thread library calls (*mutex lock and unlock*) to modify the *race.c* code to remove any potential race conditions. First, we created a global mutex called *shared_mutex* on the top of the code to achieve mutual exclusion and remove any race conditions. Next, the mutex that was created must be initialized. This was achieved in the *main* function before creating the threads. The issue that was occurring in problem one was because of race conditions and by utilizing the mutex lock and unlock functions in the *pthread.h* library, we were able to achieve mutual exclusion. In the *MakeTransactions* function, the mutex that was created must be locked before the accessing and the assignment of the shared data (*Bank.balance[0]* & *Bank.balance[1]*). After, the mutex must be unlocked to allow the next thread to access the data. Lastly, the mutex that was created must be destroyed, which we did in the *main* function after the print out of the balances. The outputs we received were more reliable and showed signs of mutual exclusion and little to no race conditions. Most of the time, the output will be something in the range of 200-210. There were many times that we were able to get a 200 output, but that was only about 50% of the time. We are sure this is achieving mutual exclusion, but this program may have smaller race conditions present that could be related to guaranteed progress or bounded waiting. Overall, the code is reliable and the main race conditions were removed to achieve a reasonable output.

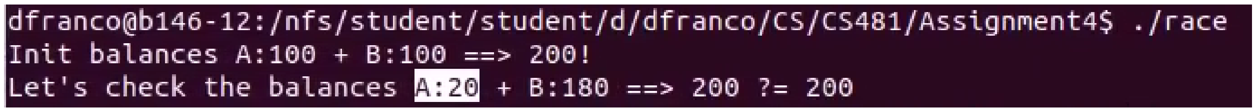
*Output equating to **200**:*



```
dfranco@b146-12:/nfs/student/student/d/dfranco/CS/CS481/Assignment4$ ./race
Init balances A:100 + B:100 ==> 200!
Let's check the balances A:29 + B:171 ==> 200 ?= 200
```

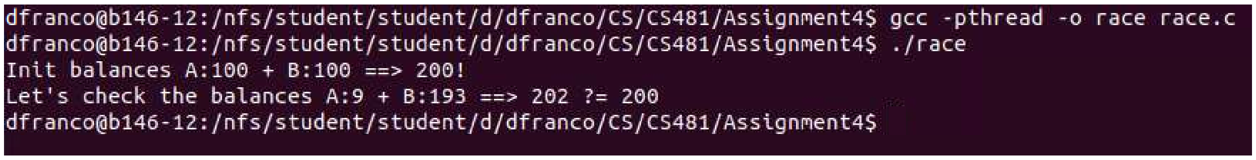


```
dfranco@b146-12:/nfs/student/student/d/dfranco/CS/CS481/Assignment4$ ./race
Init balances A:100 + B:100 ==> 200!
Let's check the balances A:93 + B:107 ==> 200 ?= 200
```

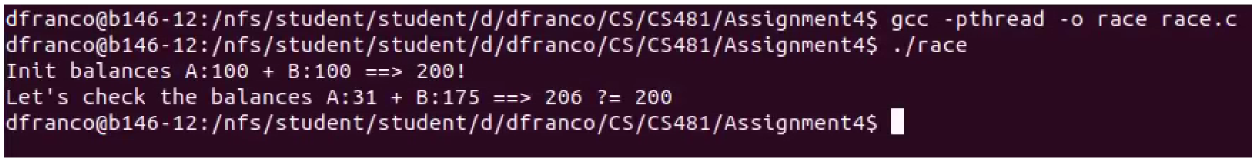


```
dfranco@b146-12:/nfs/student/student/d/dfranco/CS/CS481/Assignment4$ ./race
Init balances A:100 + B:100 ==> 200!
Let's check the balances A:20 + B:180 ==> 200 ?= 200
```

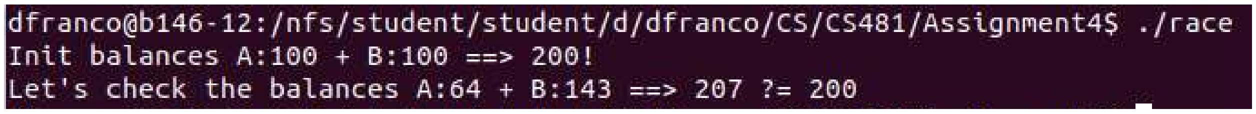
*Output showing **close** to 200:*



```
dfranco@b146-12:/nfs/student/student/d/dfranco/CS/CS481/Assignment4$ gcc -pthread -o race race.c
dfranco@b146-12:/nfs/student/student/d/dfranco/CS/CS481/Assignment4$ ./race
Init balances A:100 + B:100 ==> 200!
Let's check the balances A:9 + B:193 ==> 202 ?= 200
dfranco@b146-12:/nfs/student/student/d/dfranco/CS/CS481/Assignment4$
```



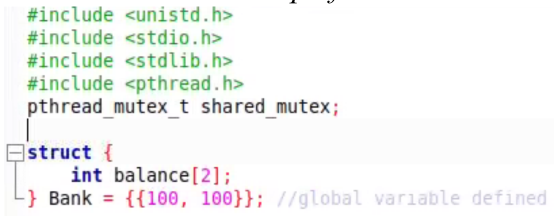
```
dfranco@b146-12:/nfs/student/student/d/dfranco/CS/CS481/Assignment4$ gcc -pthread -o race race.c
dfranco@b146-12:/nfs/student/student/d/dfranco/CS/CS481/Assignment4$ ./race
Init balances A:100 + B:100 ==> 200!
Let's check the balances A:31 + B:175 ==> 206 ?= 200
dfranco@b146-12:/nfs/student/student/d/dfranco/CS/CS481/Assignment4$
```



```
dfranco@b146-12:/nfs/student/student/d/dfranco/CS/CS481/Assignment4$ ./race
Init balances A:100 + B:100 ==> 200!
Let's check the balances A:64 + B:143 ==> 207 ?= 200
```

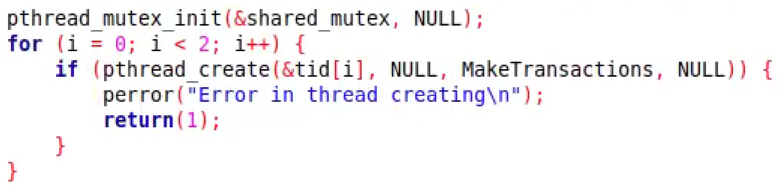
Code:

Global mutex created on top of the code:



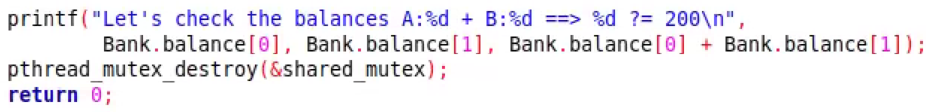
```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
pthread_mutex_t shared_mutex;

struct {
    int balance[2];
} Bank = {{100, 100}}; //global variable defined
```

Mutex initialization in the main function:



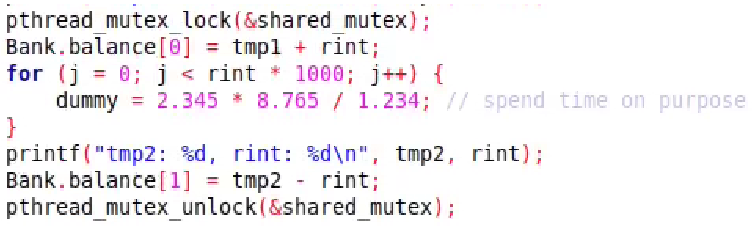
```
pthread_mutex_init(&shared_mutex, NULL);
for (i = 0; i < 2; i++) {
    if (pthread_create(&tid[i], NULL, MakeTransactions, NULL)) {
        perror("Error in thread creating\n");
        return(1);
    }
}
```

Mutex being destroyed at the end of the main function:



```
printf("Let's check the balances A:%d + B:%d ==> %d ?= 200\n",
        Bank.balance[0], Bank.balance[1], Bank.balance[0] + Bank.balance[1]);
pthread_mutex_destroy(&shared_mutex);
return 0;
```

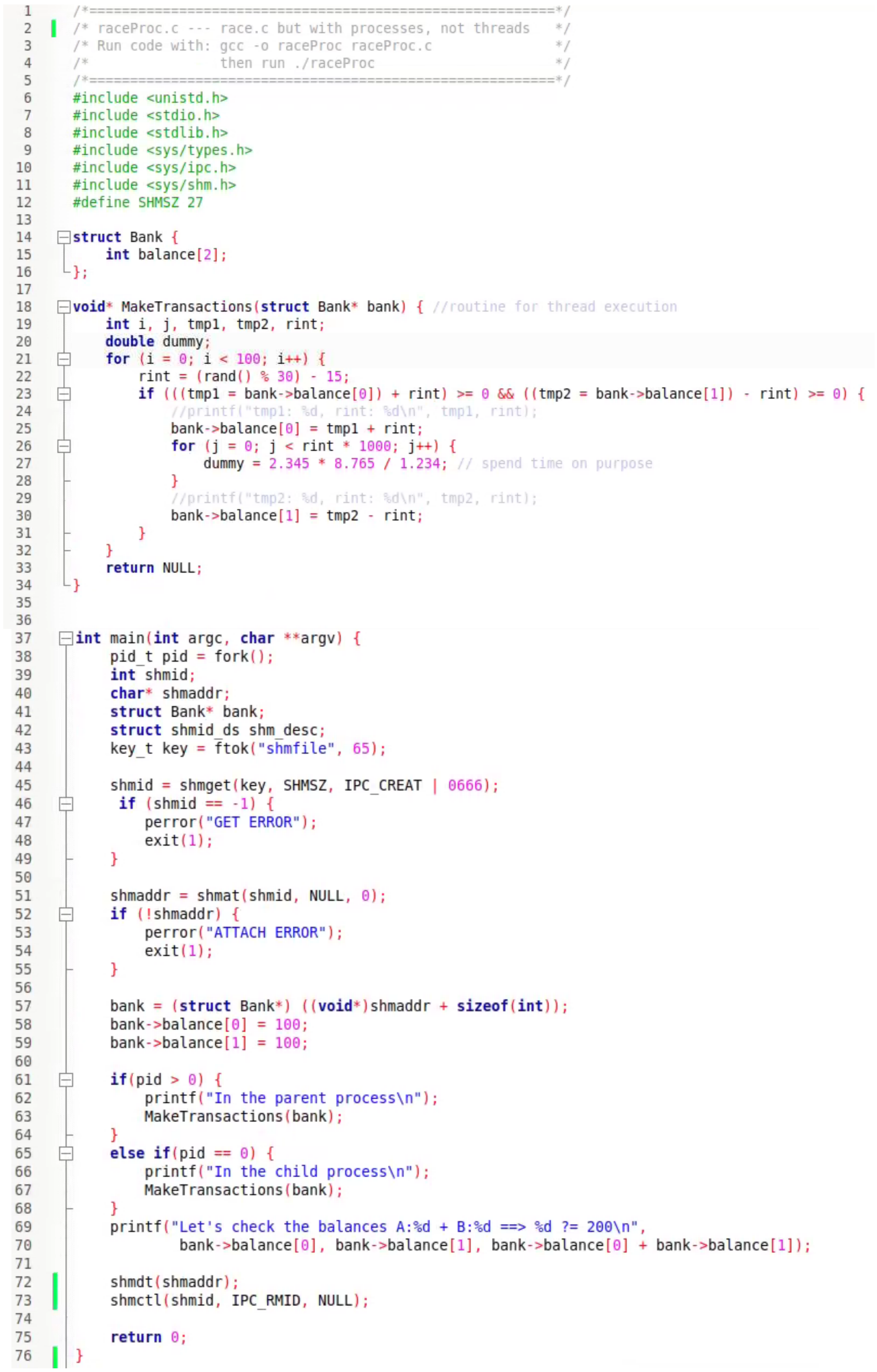
Mutex lock and unlock after accessing shared data in the MakeTransactions function:



```
pthread_mutex_lock(&shared_mutex);
Bank.balance[0] = tmp1 + rint;
for (j = 0; j < rint * 1000; j++) {
    dummy = 2.345 * 8.765 / 1.234; // spend time on purpose
}
printf("tmp2: %d, rint: %d\n", tmp2, rint);
Bank.balance[1] = tmp2 - rint;
pthread_mutex_unlock(&shared_mutex);
```

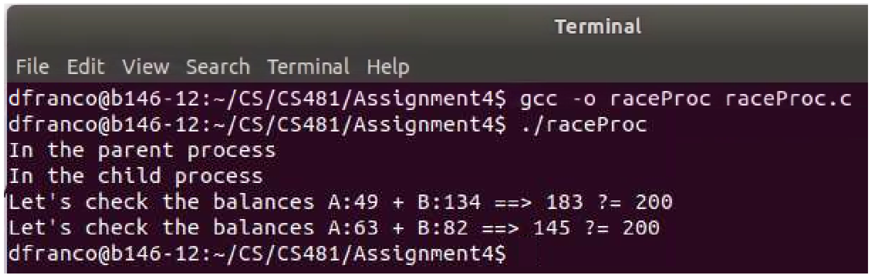
3) For number 3, we were asked to rewrite the code that was given to us (*race.c*), but utilize processes instead of threads. The transition from threads to processes requires the use of the *fork()* function call to create the processes, and the accessing and allocation of the memory that is shared between the processes (IPC). Shared memory was the approach that we took to achieve inter-process communication. Applying shared memory to the *Bank* struct was one difficulty we had to hurdle within this program. The system libraries were very helpful in being able to set, attach, detach, and destroy the shared memory. At the core, the basic procedure of the program remains the same. Both the child and parent processes would call the *MakeTransactions()* function. This function had no functional changes to the original function that was given to us, the only change is the way the *Bank* struct balances are accessed, which is why there is a *Bank* struct that is now being passed through the function. Overall, the behavior of this program was relatively the same as it was in the first problem for this assignment. The balance values that the output provided are erroneous. Race conditions are the cause of the erroneous outputs that the program produces. There is no critical section achieved because of the lack of a semaphore setup within this program, which we will provide in the next problem. Below are screenshots of the program we wrote *raceProc.c* and the output that shows the behavior of the program after running it multiple times.

Code *(raceProc.c):*



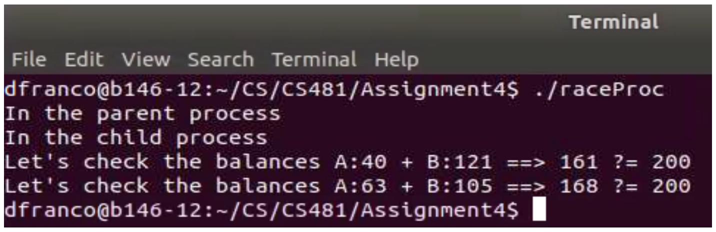
```c
/*=========================================================*/
/* raceProc.c --- race.c but with processes, not threads   */
/* Run code with: gcc -o raceProc raceProc.c               */
/*                then run ./raceProc                      */
/*=========================================================*/
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#define SHMSZ 27

struct Bank {
    int balance[2];
};

void* MakeTransactions(struct Bank* bank) { //routine for thread execution
    int i, j, tmp1, tmp2, rint;
    double dummy;
    for (i = 0; i < 100; i++) {
        rint = (rand() % 30) - 15;
        if (((tmp1 = bank->balance[0]) + rint) >= 0 && ((tmp2 = bank->balance[1]) - rint) >= 0) {
            //printf("tmp1: %d, rint: %d\n", tmp1, rint);
            bank->balance[0] = tmp1 + rint;
            for (j = 0; j < rint * 1000; j++) {
                dummy = 2.345 * 8.765 / 1.234; // spend time on purpose
            }
            //printf("tmp2: %d, rint: %d\n", tmp2, rint);
            bank->balance[1] = tmp2 - rint;
        }
    }
    return NULL;
}


int main(int argc, char **argv) {
    pid_t pid = fork();
    int shmid;
    char* shmaddr;
    struct Bank* bank;
    struct shmid_ds shm_desc;
    key_t key = ftok("shmfile", 65);

    shmid = shmget(key, SHMSZ, IPC_CREAT | 0666);
     if (shmid == -1) {
        perror("GET ERROR");
        exit(1);
    }

    shmaddr = shmat(shmid, NULL, 0);
    if (!shmaddr) {
        perror("ATTACH ERROR");
        exit(1);
    }

    bank = (struct Bank*) ((void*)shmaddr + sizeof(int));
    bank->balance[0] = 100;
    bank->balance[1] = 100;

    if(pid > 0) {
        printf("In the parent process\n");
        MakeTransactions(bank);
    }
    else if(pid == 0) {
        printf("In the child process\n");
        MakeTransactions(bank);
    }
    printf("Let's check the balances A:%d + B:%d ==> %d ?= 200\n",
            bank->balance[0], bank->balance[1], bank->balance[0] + bank->balance[1]);

    shmdt(shmaddr);
    shmctl(shmid, IPC_RMID, NULL);

    return 0;
}
```

Output after running **5** *times:*



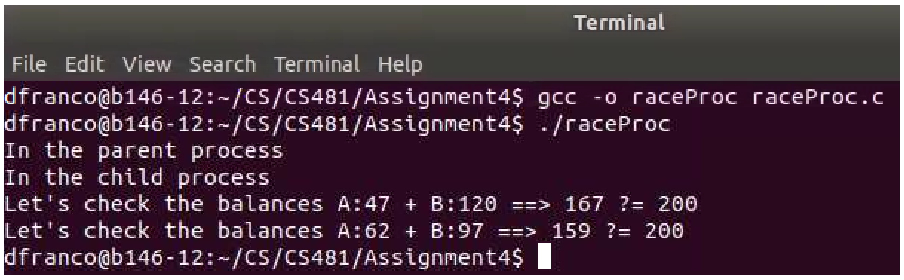
```
dfranco@b146-12:~/CS/CS481/Assignment4$ gcc -o raceProc raceProc.c
dfranco@b146-12:~/CS/CS481/Assignment4$ ./raceProc
In the parent process
In the child process
Let's check the balances A:49 + B:134 ==> 183 ?= 200
Let's check the balances A:63 + B:82 ==> 145 ?= 200
dfranco@b146-12:~/CS/CS481/Assignment4$
```

Output after running **10** *times:*



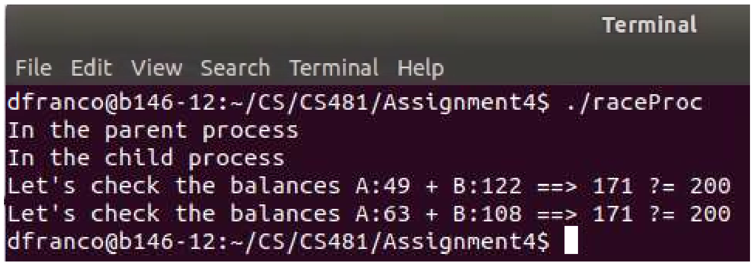
```
dfranco@b146-12:~/CS/CS481/Assignment4$ ./raceProc
In the parent process
In the child process
Let's check the balances A:40 + B:121 ==> 161 ?= 200
Let's check the balances A:63 + B:105 ==> 168 ?= 200
dfranco@b146-12:~/CS/CS481/Assignment4$
```

Output after running **20** *times:*



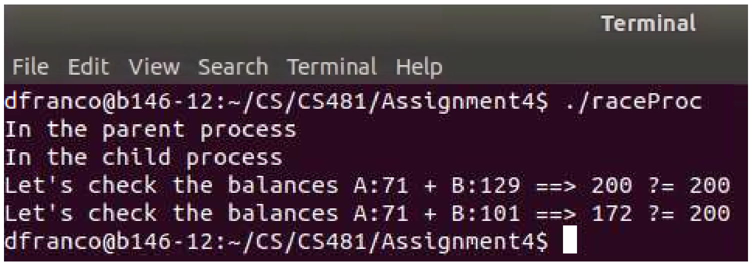
```
dfranco@b146-12:~/CS/CS481/Assignment4$ gcc -o raceProc raceProc.c
dfranco@b146-12:~/CS/CS481/Assignment4$ ./raceProc
In the parent process
In the child process
Let's check the balances A:47 + B:120 ==> 167 ?= 200
Let's check the balances A:62 + B:97 ==> 159 ?= 200
dfranco@b146-12:~/CS/CS481/Assignment4$
```

Output after running **30** *times:*



```
dfranco@b146-12:~/CS/CS481/Assignment4$ ./raceProc
In the parent process
In the child process
Let's check the balances A:49 + B:122 ==> 171 ?= 200
Let's check the balances A:63 + B:108 ==> 171 ?= 200
dfranco@b146-12:~/CS/CS481/Assignment4$
```

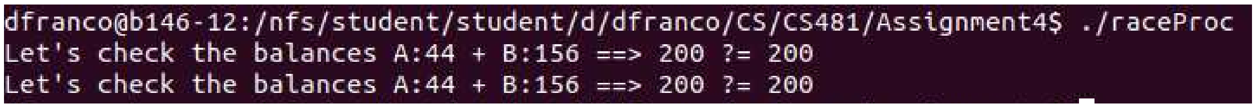
Output after running **40** *times:*



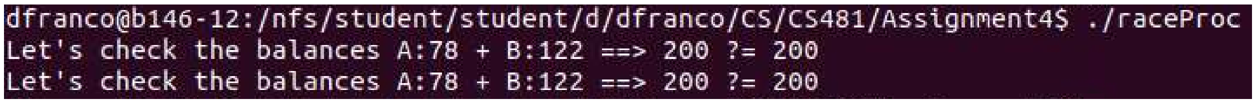
```
dfranco@b146-12:~/CS/CS481/Assignment4$ ./raceProc
In the parent process
In the child process
Let's check the balances A:71 + B:129 ==> 200 ?= 200
Let's check the balances A:71 + B:101 ==> 172 ?= 200
dfranco@b146-12:~/CS/CS481/Assignment4$
```

4) For number 4, we were asked to modify the code we wrote in the previous question to remove any race conditions to allow the program to function properly. With the addition of a semaphore setup, the inter-process communication was complete without any race conditions. By locking and unlocking the semaphore when accessing the shared memory, there was no corruption of the data. The way that we implemented the unnamed semaphore was by looking through various examples of how to set up a semaphore and how to lock and unlock a semaphore. In the *main()* method, the semaphore is initialized and checked for errors when calling the *semget()* and *semctl()* functions. The semaphore ID integer that was initialized must now be passed through to all function calls within the program because that is how we referenced which semaphore to lock and unlock. The *MakeTransactions()* function remains the same besides the addition of the function calls *sem_lock()* and *sem_unlock()*. These functions were written to lock and unlock the semaphore anytime they are called. Overall, the output of the program is very similar to the output from the program *race.c* when the race conditions were removed. Most of the time, the output will be approximately or exactly the value of 200. The output we are getting is expected, and it also shows how the semaphore is working between both the processes because of the consistent balances that are being printed out. One major takeaway from this project is that we must always check every box when it comes to scheduling, mutual exclusion and everything that involves the removal of race conditions to maintain a reliable program.

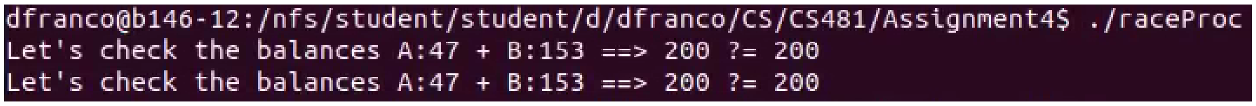
Output equating to ***200****:*



```
dfranco@b146-12:/nfs/student/student/d/dfranco/CS/CS481/Assignment4$ ./raceProc
Let's check the balances A:44 + B:156 ==> 200 ?= 200
Let's check the balances A:44 + B:156 ==> 200 ?= 200
```

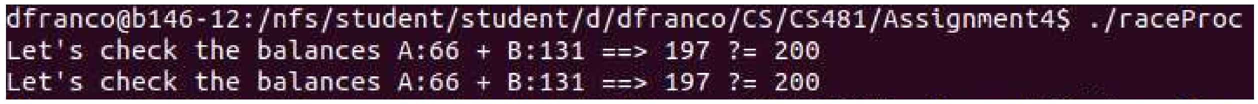


```
dfranco@b146-12:/nfs/student/student/d/dfranco/CS/CS481/Assignment4$ ./raceProc
Let's check the balances A:78 + B:122 ==> 200 ?= 200
Let's check the balances A:78 + B:122 ==> 200 ?= 200
```

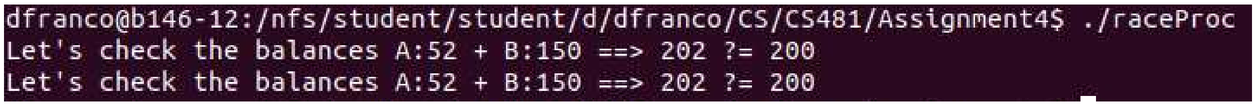


```
dfranco@b146-12:/nfs/student/student/d/dfranco/CS/CS481/Assignment4$ ./raceProc
Let's check the balances A:47 + B:153 ==> 200 ?= 200
Let's check the balances A:47 + B:153 ==> 200 ?= 200
```

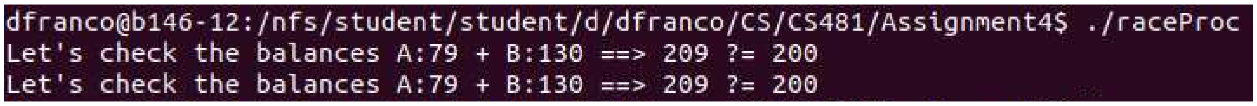
*Output showing **close** to 200:*



```
dfranco@b146-12:/nfs/student/student/d/dfranco/CS/CS481/Assignment4$ ./raceProc
Let's check the balances A:66 + B:131 ==> 197 ?= 200
Let's check the balances A:66 + B:131 ==> 197 ?= 200
```



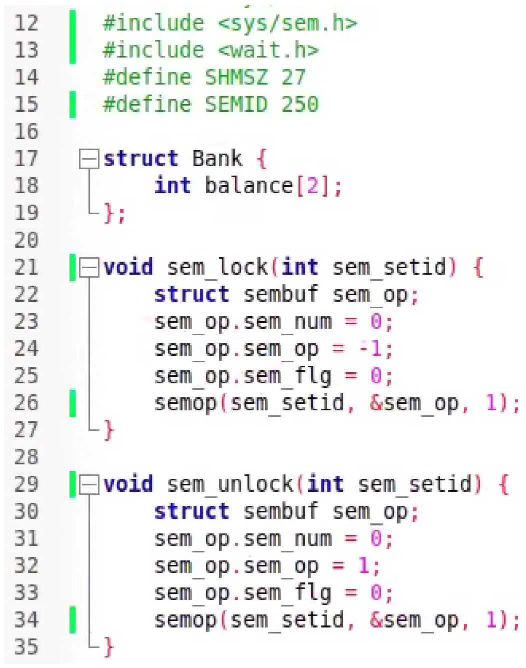
```
dfranco@b146-12:/nfs/student/student/d/dfranco/CS/CS481/Assignment4$ ./raceProc
Let's check the balances A:52 + B:150 ==> 202 ?= 200
Let's check the balances A:52 + B:150 ==> 202 ?= 200
```



```
dfranco@b146-12:/nfs/student/student/d/dfranco/CS/CS481/Assignment4$ ./raceProc
Let's check the balances A:79 + B:130 ==> 209 ?= 200
Let's check the balances A:79 + B:130 ==> 209 ?= 200
```

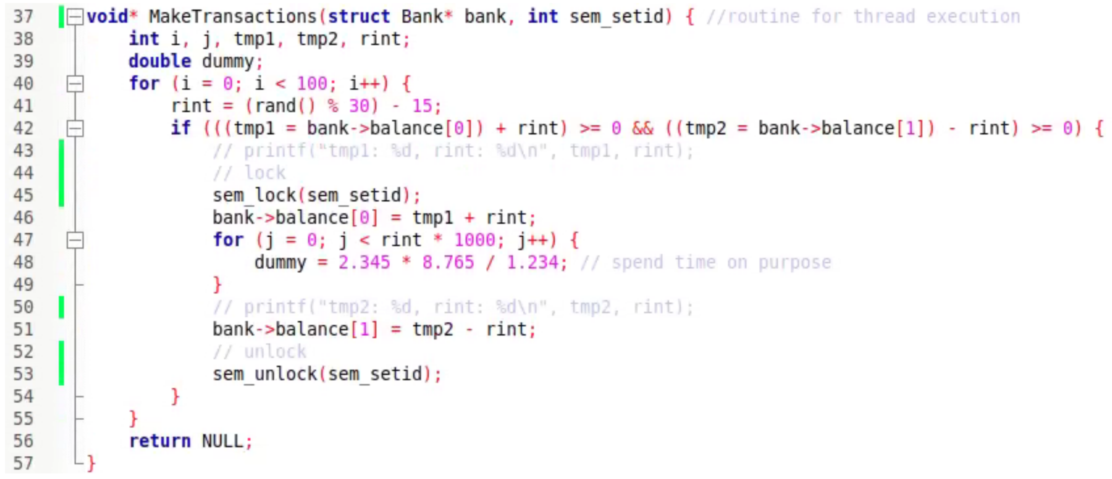
Code:

Semaphore lock and unlock functions:



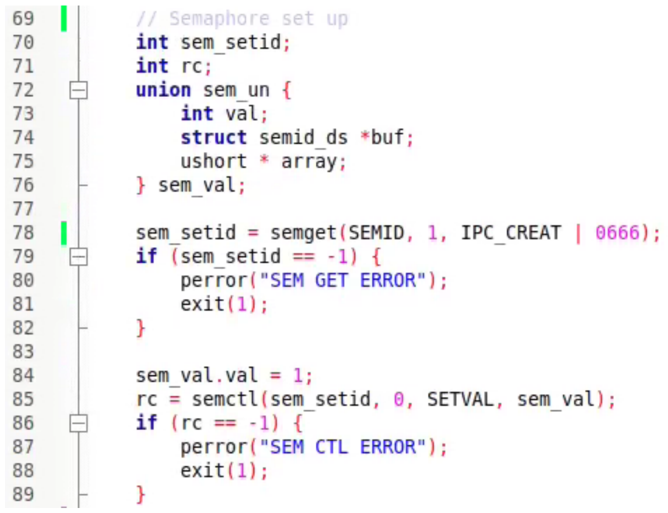
```c
#include <sys/sem.h>
#include <wait.h>
#define SHMSZ 27
#define SEMID 250

struct Bank {
    int balance[2];
};

void sem_lock(int sem_setid) {
    struct sembuf sem_op;
    sem_op.sem_num = 0;
    sem_op.sem_op = -1;
    sem_op.sem_flg = 0;
    semop(sem_setid, &sem_op, 1);
}

void sem_unlock(int sem_setid) {
    struct sembuf sem_op;
    sem_op.sem_num = 0;
    sem_op.sem_op = 1;
    sem_op.sem_flg = 0;
    semop(sem_setid, &sem_op, 1);
}
```

This shows where the unlock and lock function calls are used:



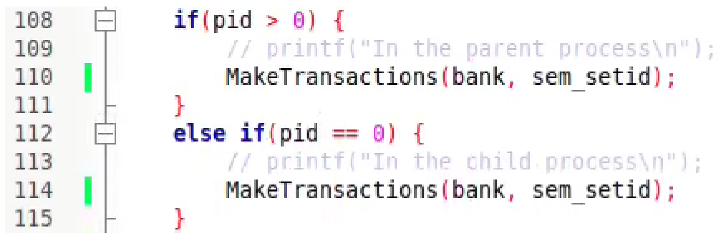
```c
void* MakeTransactions(struct Bank* bank, int sem_setid) { //routine for thread execution
    int i, j, tmp1, tmp2, rint;
    double dummy;
    for (i = 0; i < 100; i++) {
        rint = (rand() % 30) - 15;
        if (((tmp1 = bank->balance[0]) + rint) >= 0 && ((tmp2 = bank->balance[1]) - rint) >= 0) {
            // printf("tmp1: %d, rint: %d\n", tmp1, rint);
            // lock
            sem_lock(sem_setid);
            bank->balance[0] = tmp1 + rint;
            for (j = 0; j < rint * 1000; j++) {
                dummy = 2.345 * 8.765 / 1.234; // spend time on purpose
            }
            // printf("tmp2: %d, rint: %d\n", tmp2, rint);
            bank->balance[1] = tmp2 - rint;
            // unlock
            sem_unlock(sem_setid);
        }
    }
    return NULL;
}
```

Semaphore initialization and set up:



```c
    // Semaphore set up
    int sem_setid;
    int rc;
    union sem_un {
        int val;
        struct semid_ds *buf;
        ushort * array;
    } sem_val;

    sem_setid = semget(SEMID, 1, IPC_CREAT | 0666);
    if (sem_setid == -1) {
        perror("SEM GET ERROR");
        exit(1);
    }

    sem_val.val = 1;
    rc = semctl(sem_setid, 0, SETVAL, sem_val);
    if (rc == -1) {
        perror("SEM CTL ERROR");
        exit(1);
    }
```

Processes function calls, now passing through the semaphore ID:



```c
    if(pid > 0) {
        // printf("In the parent process\n");
        MakeTransactions(bank, sem_setid);
    }
    else if(pid == 0) {
        // printf("In the child process\n");
        MakeTransactions(bank, sem_setid);
    }
```