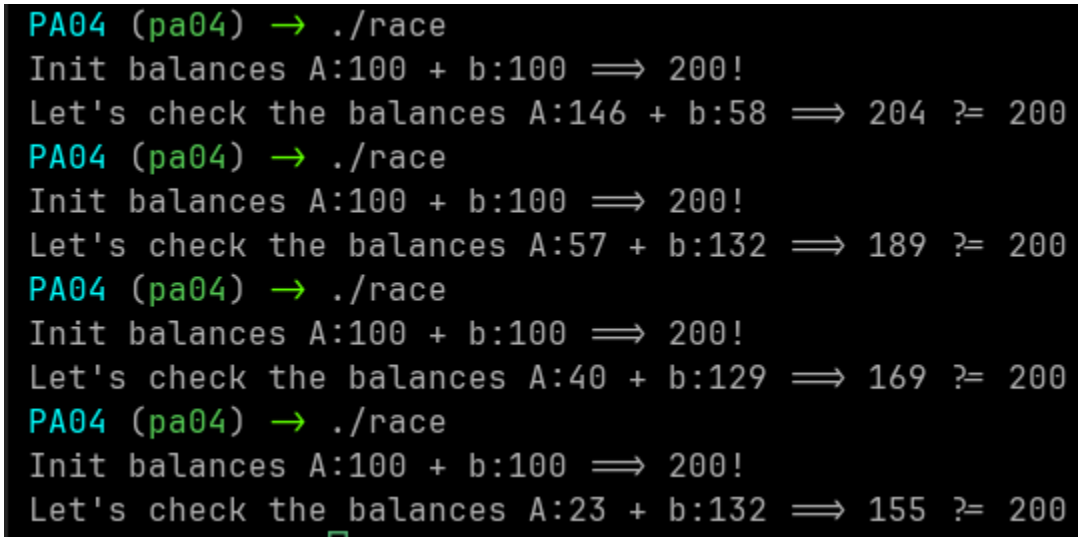
Programming Assignment #04

1. The code was run 40 times and its output is collected for analysis. The output of the program shows the occurrence of a race condition in a multithreaded environment. Although the initial balances of the two accounts are consistently set to 100 each, the final balances fluctuate significantly across runs. This variability is caused by the fact that both threads access and modify the shared Bank.balance array without proper synchronization mechanisms to control concurrent access. As a result, updates made by one thread can be interfered by another thread, leading to inconsistent state changes in the shared data and ultimately violating the expected invariant that the sum of the two balances should remain constant at 200.



2. You will find the code for this question in `thread_safe.c`. A mutex lock was added to the `Bank` struct and used to guard the critical section. Before entering the critical section, the thread will acquire the lock (or wait for it to be available) and after the critical section releases the lock. The snippet below highlights the changes made to achieve the critical section.

```
typedef struct {
        int balance[2];
        pthread_mutex_t mutex;
} Bank;

/* Routine for thread execution */
void* make_transactions() {
        int i, j, tmp1, tmp2, rint;
```

```
        double dummy;

        for (i = 0; i < 100; i++) {
                rint = (rand() % 30) - 15;

                pthread_mutex_lock(&bank.mutex); // Aquire lock
                if (((tmp1 = bank.balance[0]) + rint) >= 0 &&
                                ((tmp2 = bank.balance[1]) - rint) >= 0) {
                        bank.balance[0] = tmp1 + rint;
                        for (j = 0; j < rint * 1000; j++) {
                                dummy = 2.345 * 8.765 / 1.234;
                        }  // spend time on purpose
                        bank.balance[1] = tmp2 - rint;
                }
                pthread_mutex_unlock(&bank.mutex); // Release lock
        }

        return NULL;
}

int main(int argc, char** argv) {
        pthread_mutex_init(&bank.mutex, NULL);
        ...
}
```

By making these modifications, the program now correctly outputs the account values where they always sum to the initial total value of 200.

```
PA04 (pa04) → ./thread_safe
Init balances A:100 + b:100 ⟹ 200!
Let's check the balances A:197 + b:3 ⟹ 200 ?= 200
PA04 (pa04) → ./thread_safe
Init balances A:100 + b:100 ⟹ 200!
Let's check the balances A:13 + b:187 ⟹ 200 ?= 200
PA04 (pa04) → ./thread_safe
Init balances A:100 + b:100 ⟹ 200!
Let's check the balances A:69 + b:131 ⟹ 200 ?= 200
PA04 (pa04) → ./thread_safe
Init balances A:100 + b:100 ⟹ 200!
Let's check the balances A:23 + b:177 ⟹ 200 ?= 200
```

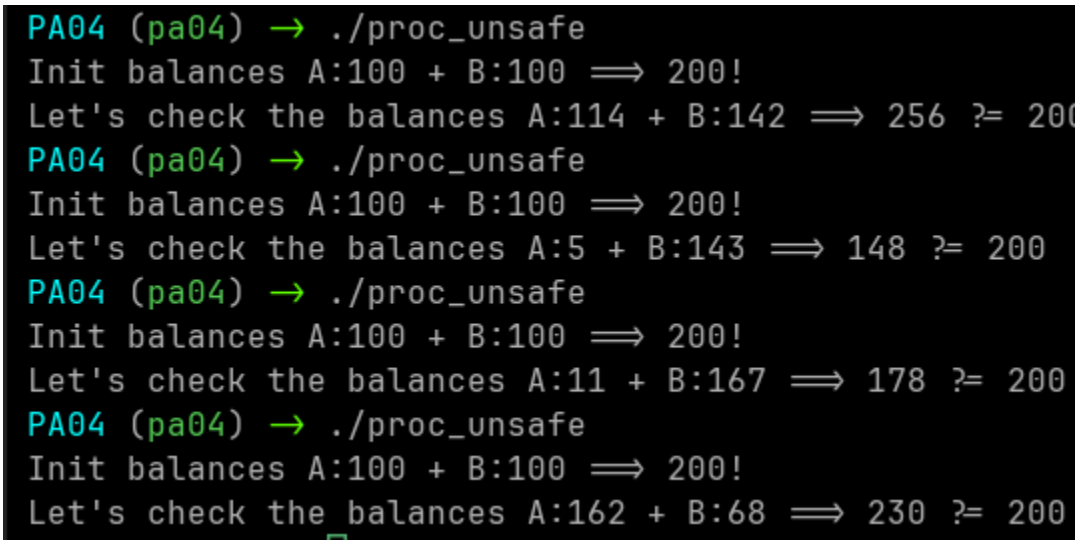
3. You will find the source code for this question in `proc_unsafe.c`. To allow both processes to modify the same bank, we used a shared memory segment. The modifications to the code are listed below.

```
// Define global variables
Bank *bank;

...

int main(int argc, char** argv) {
        int shmid;
        char *shm;
        srand(getpid());

        // Create shared memory segment
        key_t key = ftok("proc_unsafe.c", 65);
        if ((shmid = shmget(key, sizeof(Bank), IPC_CREAT | 0666)) < 0) {
                perror("Failed to get shared memory segment");
                exit(EXIT_FAILURE);
        }

        // Attach the shared memory segment
        if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
                perror("Failed to attach shared memory segment");
```

```
                exit(EXIT_FAILURE);
        }

        // Initialize the bank
        bank = (Bank *) shm;
        bank->balance[0] = 100;
        bank->balance[1] = 100;

        // Print the initial state
        printf("Init balances A:%d + B:%d ==> %d!\n",
        bank->balance[0], bank->balance[1], bank->balance[0] + bank->balance[1]);

        pid_t pid = fork();
        if (pid > 0) {
                // Parent process
                make_transactions();
        } else if (pid == 0) {
                // Child process
                make_transactions();
                shmdt(shm);
                exit(EXIT_SUCCESS);
        } else {
                // Fork failed
                perror("Failed to fork");
                exit(EXIT_FAILURE);
        }

        // Wait for child to finish
        wait(NULL);

        // Print the results
        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]);

        // Cleanup
        shmdt(shm);
        shmctl(shmid, IPC_RMID, NULL);  // Remove shared memory segment
```

```
    return 0;
}
```

Yes, the race condition still exists in this version using processes. The critical section in make_transactions() where we read, modify, and write the balances is still not protected by any synchronization mechanism and produces output where the sum of the account balances does not add up to the original sum.



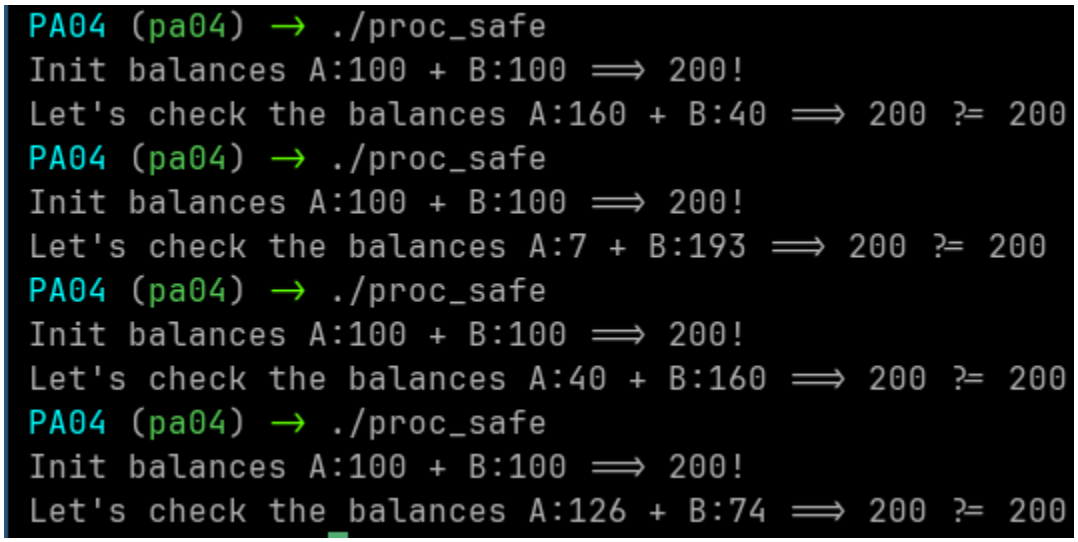
4. You will find the code for this question in `proc_safe.c`. The achieve the critical section with multiple processes, a semaphore was added to the Bank struct which both processes will share. Before entering the critical section, the process will wait if the other process is currently in the critical section and enter when the other process signals that it has exited. The code below shows the modifications made.

```
typedef struct {
    int balance[2];
    sem_t sem;
} Bank;

// Define global variables
Bank *bank;

/* Routine for thread execution */
void* make_transactions() {
```

```
        int i, j, tmp1, tmp2, rint;
        double dummy;

        for (i = 0; i < 100; i++) {
                rint = (rand() % 30) - 15;

                sem_wait(&bank->sem);
                if (((tmp1 = bank->balance[0]) + rint) >= 0 &&
                                ((tmp2 = bank->balance[1]) - rint) >= 0) {
                        bank->balance[0] = tmp1 + rint;
                        for (j = 0; j < rint * 1000; j++) {
                                dummy = 2.345 * 8.765 / 1.234;
                        }  // spend time on purpose
                        bank->balance[1] = tmp2 - rint;
                }
                sem_post(&bank->sem);
        }

        return NULL;
}


int main(int argc, char** argv) {
        ...
        // Initialize the bank
        bank = (Bank *) shm;
        bank->balance[0] = 100;
        bank->balance[1] = 100;
        sem_init(&bank->sem, 1, 1);

        ...

        // Cleanup
        sem_destroy(&bank->sem);
        shmdt(shm);
        shmctl(shmid, IPC_RMID, NULL);  // Remove shared memory segment
```

```
        return 0;
}
```

By making these modifications, the race condition is no longer present. The sum of the account values after running the program now equal the sum of the account values at initialization as desired.



Tasks

Both Matt and Leo developed and experimented with the code for all questions related to this assignment. After completing our implementations, we observed our code to be structured in similar manners and merged our code into the final product. Leo wrote the responses for questions 1 and 3 while Matt authored questions 2 and 4.