Anthony Galczak
CS481 – PA03
Semaphores and Mutex's

NOTE: I wrote a Python script (run_race.py) that compiles an implementation of race.c and extracts the output and sees how many we got wrong (not 200) and how much we were off by average.
I have attached this with the assignment if you're interesting in running some analysis.
NOTE: I worked off and on with Tristin Glunt on this, however we wrote separate reports and code.

Q1.) This code is supposed to add a random int from -15 to 15 to balance 0 and remove a random int -15 to 15 from balance 1. This means that overall the balances between the two should always add up to 200 as it is some version of a "zero-sum" game. However, due to the threads executing non-deterministically we almost never get A + B to equal to 200. Basically, the second thread comes in and messes with the balance before the first thread was done.

For regular race.c, I got: 48 correct out of 1000 (4.8%), 52.62 off by average.

Q2.)
I added a global called "shared_mutex". I also call pthread_mutex_lock before any of the declarations inside the for loop and then call pthread_mutex_unlock after all the operations are done in the for loop. This is still a multi-threaded program as the iterations can execute out of order, but we are making it so that each iteration of the for loop is one big atomic operation.

For using mutex's on race.c, I got: 1000 correct out of 1000.
My code changes are shown below. Note that the code is slightly re-written for readability. I am only showing the code for MakeTransactions() and the global shared_mutex. The code in main didn't change at all.

```
pthread_mutex_t shared_mutex;

void* MakeTransactions()
{
    //routine for thread execution
    int i, j, tmp1, tmp2, random_int, tempPlusRandInt, tempMinusRandInt;
    double dummy;

    for (i=0; i < 100; i++)
    {
        pthread_mutex_lock(&shared_mutex);
        random_int = (rand()%30)-15;
        tmp1 = Bank.balance[0];
        tmp2 = Bank.balance[1];
        tempPlusRandInt = tmp1 + random_int;
        tempMinusRandInt = tmp2 - random_int;

        if (tempPlusRandInt >= 0 && tempMinusRandInt >=0)
        {
            Bank.balance[0] = tempPlusRandInt;

            // spend time on purpose
            for (j=0; j < random_int*1000; j++)
            {
                dummy=2.345*8.765/1.234;
            }

            Bank.balance[1] = tempMinusRandInt;
        }
        pthread_mutex_unlock(&shared_mutex);
    }

    return NULL;
}
```

Q3.) By making two separate processes that used the shared Bank struct we have now caused a race condition again in our code. When running my fancy Python script, I get 12 correct out of 1000 (1.2%) and average of 30.46 off by average.Since the memory is shared, the processes are accessing the same balances and thus the adds and subtracts are non-deterministic.

Code is copied below. The struct now has a name so that we can do things like sizeof(bank_t) and initialize the Bank as a bank_t struct. Additionally since we are using a pointer all the Bank calls to balance have to be "->" instead of ".".

```
typedef struct
{
    int balance[2];
} bank_t;
bank_t* Bank;


int main(int argc, char **argv)
{
    // Making the shared memory segment
    key_t key = ftok("race_process.c", 0);
    int shared_memory_id = shmget(key, sizeof(Bank), IPC_CREAT | 0666);
    Bank = (bank_t *)shmat(shared_memory_id, NULL, 0);

    Bank->balance[0] = 100;
    Bank->balance[1] = 100;

    srand(getpid());
    printf("Init balances A:%d + B:%d ==> %d!\n", Bank->balance[0],Bank-
>balance[1],Bank->balance[0]+Bank->balance[1]);

    if(fork() == 0)
    {
      MakeTransactions();
    }
    else
    {
      MakeTransactions();
    }

    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]);

    return 0;
}
```

Q4.) In order to fix the race condition for the two processes, I used a named semaphore. This consisted of declaring a global semaphore and then initializing it in main() with sem_open(). One critical thing was using the parameter O_CREAT, but not O_EXCL since this semaphore is not exclusive to a single process.

There is also the issue of whether we are putting sem_wait and sem_post inside or outside the for loop. If I put sem_wait directly before the for loop in MakeTransactions() and sem_post directly after the for loop in MakeTransactions() then it 100% fixes the race condition. However, I understand it is in the spirit of the assignment to put it inside the for loop for actual “multi-process” execution of the iterations.

There is some interesting behavior between different types of machines that I have noticed on using semaphores as well. I am using an 8th gen i7 running Red Hat 7 for my personal machine and when I put sem_wait and sem_post inside the for loop (just like in part 2, using mutex's) then my accuracy over 1000 iterations is between 88-92%. Meaning we are still encountering a race condition. When I instead use the CS machines which are slightly slower and a different OS (5th gen Xeon, Ubuntu) it is reliably near 100% (getting between 99-99.5% accuracy, i.e. 5-10 wrong out of 1000 iterations). After talking to Professor Sun he verified that running this script on his machine with my race_semaphore.c got 100% accuracy.

I'm not totally sure why I'm seeing a race condition specifically on my machine, but one thing to note is that checking by hand on my machine or on the CS machines likely would not show any race condition.

Here is the code only excluding the includes.

```
typedef struct
{
    int balance[2];
} bank_t;
bank_t* Bank;

sem_t* shared_semaphore;

void* MakeTransactions()
{
    //routine for thread execution
    int i, j, tmp1, tmp2, random_int, tempPlusRandInt, tempMinusRandInt;
    double dummy;

    for (i=0; i < 100; i++)
    {
        // decrementing counter, "locking"
        sem_wait(shared_semaphore);
        random_int = (rand()%30)-15;
        tmp1 = Bank->balance[0];
        tmp2 = Bank->balance[1];
        tempPlusRandInt = tmp1 + random_int;
        tempMinusRandInt = tmp2 - random_int;

        if (tempPlusRandInt >= 0 && tempMinusRandInt >=0)
        {
            Bank->balance[0] = tempPlusRandInt;

            // spend time on purpose
            for (j=0; j < random_int*1000; j++)
            {
                dummy=2.345*8.765/1.234;
            }

            Bank->balance[1] = tempMinusRandInt;
        }
                // Incrementing counter, "unlocking"
        sem_post(shared_semaphore);
    }
    return NULL;
}

int main(int argc, char **argv)
{
    // Making the shared memory segment
    key_t key = ftok("race_process.c", 0);
    int shared_memory_id = shmget(key, sizeof(Bank), IPC_CREAT | 0666);
    Bank = (bank_t *)shmat(shared_memory_id, NULL, 0);

    Bank->balance[0] = 100;
    Bank->balance[1] = 100;

    // Initializing semaphore, O_CREAT either creates or uses already
existing
    // semaphore. Additionally, last parameter 1 so that we have a counter
    // already incremented so we can call wait right away.
    shared_semaphore = sem_open("semaphore", O_CREAT, 0666, 1);

    srand(getpid());
    printf("Init balances A:%d + B:%d ==> %d!\n", Bank->balance[0],Bank-
>balance[1],Bank->balance[0]+Bank->balance[1]);

    if(fork() == 0)
    {
      MakeTransactions();
    }
    else
    {
      MakeTransactions();
    }

    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]);

    return 0;
}
```

Lastly, I have decided to submit all of the code in a .zip file if you're interested in doing any testing with the Python script I made or the code I wrote for parts 1, 2, 3, or 4. I'm especially interested to see if sem_wait and sem_post inside the for loop on #4 works at 100% accuracy on other machines. Professor Sun said that indeed it works correctly on his hardware.