PA05

Jayson Grace

Tuesday 20$^{\text{th}}$ October, 2015

Jayson Grace

Problem 1a

Upon running the code, I was able to determine that the balance was not what I was expecting it to be. Instead of staying at the initial balance of 200, it would either be too big or too small. After inspecting the code, I saw that there were a number of floating point calculations happening between the calculation of balances A and B:

```
25        Bank.balance[0] = tmp1 + rint;
26        for (j = 0; j < rint * 100; j++)
27        {
28          dummy = 2.345 * 8.765 / 1.234;
29        }
30        Bank.balance[1] = tmp2 - rint;
```

To determine if this was causing an issue, I moved the Balance calculation for B above the loop:

```
25        Bank.balance[0] = tmp1 + rint;
26        Bank.balance[1] = tmp2 - rint;
27        for (j = 0; j < rint * 100; j++)
28        {
29          dummy = 2.345 * 8.765 / 1.234;
30        }
```

This almost completely eliminated any issues that I was seeing before, with the exception of a few discrepancies.

After looking at the code a bit more, I came to realize that the variable **balance** is shared between the two threads that we create while running the program. As such, a race condition can occur when the two threads are trying to modify that variable.

To confirm this might be causing an issue, I used valgrind:

```
valgrind --tool=helgrind ./race -v
```

The threads accessing the variable without any protection appear to be the catalyst of the sporadic behavior I saw when executing the program. The behavior became less prevalent when I took the latency being introduced by all of the floating point calculations out of the equation, but it did not completely resolve the issue. In fact, this issue will persist until a form of protection is introduced into the program.

Problem 1b

```
#include <pthread.h>

typedef struct
{
    int S;
    pthread_mutex_t mut;
    pthread_cond_t cond;
} Sem437;

void Sem437Init(Sem437*, int);
void Sem437P(Sem437*);
void Sem437V(Sem437*);

/**
* @brief Used to initialize the mutex and condition variable
* @param sem semaphore to initialize
* @param i value to associate with the semaphore
*/
void Sem437Init(Sem437* sem, int i)
{
    sem->S = i;
    pthread_mutex_init(&sem->mut, NULL);
    pthread_cond_init(&sem->cond, NULL);
}

/**
* @brief Used for decrementing
* @param sem semaphore to decrement
*/
void Sem437P(Sem437* sem)
{
    pthread_mutex_lock(&sem->mut);
    // If semaphore is negative, block
    while(sem->S <= 0)
        pthread_cond_wait(&sem->cond, &sem->mut);
    sem->S -= 1;
    pthread_mutex_unlock(&sem->mut);
}
```

```
/**
* @bried Used for incrementing
* @param sem semaphore to increment
*/
void Sem437V(Sem437* sem)
{
    pthread_mutex_lock(&sem->mut);
    sem->S += 1;
    pthread_cond_signal(&sem->cond);
    pthread_mutex_unlock(&sem->mut);
}
```

Problem 1c

After modifying the code, race.c now looks like this:

```
/*==========================================================*/
/* race.c -- for playing with ECE437 */
/*==========================================================*/
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <pthread.h>

typedef int bool;
enum { false, true };

typedef struct
{
    int S;
    pthread_mutex_t mut;
    pthread_cond_t cond;
} Sem437;

void Sem437Init(Sem437*, int);
void Sem437P(Sem437*);
void Sem437V(Sem437*);

/**
 * @brief Used to initialize the mutex and condition variable
 * @param sem semaphore to initialize
```

```
 * @param i value to associate with the semaphore
 */
void Sem437Init(Sem437* sem, int i)
{
    sem->S = i;
    pthread_mutex_init(&sem->mut, NULL);
    pthread_cond_init(&sem->cond, NULL);
}

/**
 * @brief Used for decrementing
 * @param sem semaphore to decrement
 */
void Sem437P(Sem437* sem)
{
    pthread_mutex_lock(&sem->mut);
    // If semaphore is negative, block
    while(sem->S <= 0)
        pthread_cond_wait(&sem->cond, &sem->mut);
    sem->S -= 1;
    pthread_mutex_unlock(&sem->mut);
}

/**
 * @bried Used for incrementing
 * @param sem semaphore to increment
 */
void Sem437V(Sem437* sem)
{
    pthread_mutex_lock(&sem->mut);
    sem->S += 1;
    pthread_cond_signal(&sem->cond);
    pthread_mutex_unlock(&sem->mut);
}

struct
{
    int balance[2];
}

Bank = {{100, 100}}; //global variable defined
```

```
// Semaphore lock we'll be using
Sem437 lock;

// Specify whether or not debug mode should be run
bool DEBUG = false;

void *MakeTransactions()
{ //routine for thread execution
    int i, j, tmp1, tmp2, rint;
    double dummy;
    if (DEBUG)
        printf("\nSemaphore value: %i\n",lock.S);
    Sem437P(&lock);
    for (i = 0; i < 100; i++)
    {
        rint = (rand() % 30) - 15;
        if (((tmp1 = Bank.balance[0]) + rint) >= 0 && ((tmp2 = Bank.ba
        {
            Bank.balance[0] = tmp1 + rint;
            for (j = 0; j < rint * 100; j++)
            {
                dummy = 2.345 * 8.765 / 1.234;
            }
            Bank.balance[1] = tmp2 - rint;
        }
    }
    Sem437V(&lock);
    return NULL;
}

int main(int argc, char **argv)
{
    int i;
    void *voidptr = NULL;
    pthread_t tid[2];
    Sem437Init(&lock, 1);
    srand(getpid());
    printf("Init balances A:%d + B:%d ==> %d!\n", Bank.balance[0], Ban
    for (i = 0; i < 2; i++)
        if (pthread_create(&tid[i], NULL, MakeTransactions, NULL))
```

```
        {
            perror("Error in thread creating\n");
            return (1);
        }
    for (i = 0; i < 2; i++)
        if (pthread_join(tid[i], (void *) &voidptr))
        {
            perror("Error in thread joining\n");
            return (1);
        }
    printf("Let's check the balances A:%d + B:%d ==> %d ?= 200\n",
            Bank.balance[0], Bank.balance[1], Bank.balance[0] + Bank.b
    return 0;
}
```

This adds in the semaphore created for 1b to create a lock for the block of code in which a race condition is occurring. As a result, we are able to eliminate any discrepancies introduced by the formerly unprotected balance variable and the balance is as it should be.