PA05

Jayson Grace, Dominic Salas

Thursday 5th November, 2015

Jayson Grace, Dominic Salas

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.

Jayson Grace, Dominic Salas

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.

Problem 2a

Make your own version of simrw.c based on simrwsimple.c provided.

- We were able to successfully modify our code to take in all of the options as described.
- We were a little confused about protecting the global data. Global locks were already implemented in the sample code given. It was unclear if additional work was required for 2a.
- Writers are not being starved because the basic implementation is alternating and only allowing one item in at a time.
- Readers are not sharing the room with the basic implementation. Only one reader is being allowed in at a time because the current code is not distinguishing between a reader and a writer and is not setup to accommodate additional members in the room at once.
- ANSWER - Is finishCLK >> arrvCLK? Can you estimate finishCLK based on arrvCLK, arrival rate,time to read/write?

Jayson Grace, Dominic Salas

Problem 2b

Case A: a naive scheme in favor of readers, as long as one or more readers in the room, all writers have to wait.

- Synchronization primitives used for reader/writer coordination has been successfully implemented
- Writers are being starved since we're giving priority to the readers. Readers are always able to enter the room with other readers but we're also preventing writers from entering while there are readers waiting.
- Readers are now sharing the room.
- There is a substantial increase in the average waiting time between X=5 and X=10, which is expected since this is increasing the length of time spent in the room.

Problem 2c

Case B: same as Case A except that the room has a capacity limit. If an additional reader comes, it has to compete with any waiting writers.

- Synchronization primitives used for reader/writer coordination has been successfully implemented
- From 2b or Case A you'll notice from the table that there is a large increase in the wait times and denies. The increase in wait times is expected since here we're limiting the number of readers that can be in a room all at once. The capacity limit was unspecified in this problem, but we used C=5 since that was listed in the last problem.

Problem 2d

Case C:.a scheme with relative equal priority. For a newly arrived reader, it cannot join the room unless there is no writer being waiting.

- Synchronization primitives used for reader/writer coordination has been successfully implemented

- You see a number of changes from A Case C. There is a big distinction between the wait times. In A we are giving priority to the readers and as a result there is a huge wait time for writers especially when we increase our time to simulate to 10. In case C, we are giving more priority to the writers so that it's more balanced out. Also, with case C, are numbers are more balanced out slightly favoring are writers. But discrepancies are far less than what is seen in case A.

- With respect to the maxRoom in cases B & A, we are passing in the max room for our testing. However, by modifying the size of the room in case B and case C, you're naturally increasing the wait times. This is expected, since you're essentially forcing my readers to wait and naturally will cause writers to wait longer as well.

P	finishCLK	numR	numW	numDeny	avgRwait	avgWwait	maxRwait	maxWwait	maxRroom
=0, X=5	5002	824	329	3047	1054.6	2508.3	1402	3940	1
=1, X=5	3604	3484	717	0	0.7	11.3	10	59	14
=1, X=10	3873	3486	209	508	3.0	3138.9	19	3615	14
=2, X=5	4024	3486	203	508	109	3393	183	3776	5
=2, X=10	4312	1938	203	2061	386.7	3672.1	523	4035	5
=3, X=5	3611	3486	717	0	5.3	5.1	25	15	14
=3, X=10	3760	3450	717	36	152.5	6.6	230	32	14