Programming Assignment № 5

Steven Seppala 24 October 2014

Programming assignment five

1. Race Condition

   The unchanged code has metastability issues because of the numerous floating point calculations that take place before and after variables are being assigned values. This will cause a context switch to occur fairly often in-between the floating point calculations that will assign a value to a variable before the previous thread can complete its task. This in turn, due to the instruction latency of floating point calculations, will cause the metastability race condition to give incorrect values.

```
        {

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

   The expression was moved to below the dummy floating point instructions. This improves performance because the floating point calculations take much longer per instruction cycle to execute and go through an ALU than simply assigning a value to a variable does. Moving the instructions so that they are concurrent will solve 90% of the race conditions.I chose to move them both below the floating point calculations simply because I was scrolling down.

2. Semaphore monitor implementation

   Using a semaphor monitor implementation that locks and waits the threads depending on a record variable makes all race conditions disappear. Using the sem437.c class as shown below, I use a binary record that locks when the first thread enters, and puts the next thread into a waiting state that blocks. Once the first thread calls V(); it signals the next thread to leave the blocking mode and begin executing code again.

```c
#include<pthread.h>


typedef struct
{
    int record; //semaphore count
    pthread_mutex_t mux;
    pthread_cond_t cond;
}Sem437;

void Sem437Init(Sem437 *s, int a)
{
    s->record = a;
    pthread_mutex_init(&s->mux, NULL); //initalize mutex
    pthread_cond_init(&s->cond, NULL); //initalize condition

}

void Sem437P(Sem437 *s)//Decrimentor
{
    pthread_mutex_trylock(&s->mux);
    while (s->record == 0)
        pthread_cond_wait(&s->cond,&s->mux);
    s->record = 0;
    pthread_mutex_unlock(&s->mux);
}

void Sem437V(Sem437 *s) //Incrimentor
{
    pthread_mutex_lock(&s->mux);
    int whatis = s->record;
    s->record = 1;
    pthread_mutex_unlock(&s->mux);
    if(whatis == 0)
        pthread_cond_signal(&s->cond);
}
```

This method of blocking and waiting for the thread to continue before the next thread can run will ensure that no context switching between different threads happens and lets only one thread modify data at any given time.

3. Traffic Light

   The traffic light program was implemented using a consumer producer scheme as illustrated by the sample code given. This method is augmented by the fact that each product is given its own thread as well. The main thread will signal threads to allow them to be processed as going through the intersection. All information is displayed as requested exactly in the assignment. The assignment is also implemented with semaphores and mutex's as requested in the assignment; there is no mutex monitor implemented as it is filler code and not necessary.