Samuel Villarreal

Sid: 861021551

Cs160 Concurrent Programming and Parallel Systems

CS 160: Lab Assignment 2

Due at 11:59PM on Feb 24, 2014

1)

File is included you can run test by calling: make test1

```
/*
 * hello world program
 *  arg: a number for threads
 *  output: each thread will print "hello world"
*/

#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

void *thread(void *vargp)
{
      printf("hello world\n");
      return NULL;
}

int main()
{
      int nthreads ;

      printf("number of threads wanted? :");
```

```
    scanf("%d", &nthreads);

    pthread_t tid[nthreads];
    for(int i = 0; i < nthreads; i++)
    {
        pthread_create(&tid[i] , NULL, thread, NULL);
    }
    for( int j = 0 ; j < nthreads; j++)
    {
        pthread_join(tid[j],NULL);
    }

    return 0;
}
```

2)

File is included you can run test by calling: make test2

/*

* what was happening was that the main was exiting before the thread woke up

* fix: waited for the thread created to finish before exiting main

*/

```
/*
* what was happening was that the main was exiting before the thread woke up
* fix: waited for the thread created to finish before exiting main
*/

#include <pthread.h>
#include <stdio.h>
```

```
#include <stdlib.h>
#include <unistd.h>

void *thread(void *vargp);

int main()
{
      pthread_t tid;

      pthread_create(&tid, NULL, thread, NULL);
//
      pthread_join(tid, NULL);
//
      exit(0);
}

void *thread(void *vargp)
{
      sleep(1);
      printf("Hello, world!\n");
      return NULL;
}
```

3)

File is included you can run test by calling: make test3

Problem was that the main and new thread race to look at t

```
#include <pthread.h>
```

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#define NUM_THREADS 6

/*
* The bug was that where passing t and having a race conditon on it
*/
void *PrintHello(void *threadid)
{
      long taskid = *(long*)threadid;
      free(threadid);
      sleep(1);
      printf("Hello from thread %ld\n", taskid);
      pthread_exit(NULL);

}

int main()
{
      pthread_t pthreads[NUM_THREADS];
      int rc;
      long t;

      for(t = 0; t < NUM_THREADS; t++)
      {
            printf("creating thread %ld\n",t);
            long* temp = (long*)malloc(sizeof(long));
            *temp = t;
            rc = pthread_create(&pthreads[t],NULL, PrintHello, (void *) 
temp);
            if(rc)
            {
                  printf("err");
                  exit(-1);
            }
      }

      pthread_exit(NULL);
}
```

4)

A)

V(t)

P(t)

V(s)

P(s)

P(s) V(s) P(t) V(t)

Red Squares denote : critical section

Green lines denote : possible paths

Purple X denote: dead lock

Initially: s =1 , t= 0

B) Yes, It always deadlocks because since t = 0 from the start the sema does not let anything in.

C) If we were to initialize t = 1 then that would solve the problem.

D)

V(t)

P(t)

V(s)

P(s)

P(s) V(s) P(t) V(t)

5)

It cannot dead lock, first we notice that (a) is only used in thread 1 so we can look beyond that:

Thread1	Thread2
P(b)	P(c)
V(b)	P(b)
P(c)	V(b)
V(c)	V(c)

We can see that if Thread1 grabs (b) it will promptly release it, allowing Thread2 to continue and release (c) for Thread1. If Thread2 grabs b first it also releases right after allowing Thread1 to start and complete after Thread2 also releases (c).

6)

Initially a=1 b=1 c=1

A)

Thread1: (ab) and (ac)

Thread2: (cb)

Thread3: (ab)

B) Thread2 and Thread3 violate the ordering rule

C) Thead2: Thread3:

```
P(a) ;          P(a);
P(b);           V(a);
V(b);           P(b);
V(a);           P(c);
P(c);           V(c);
V(c);           V(b);
```

7)

A)they are necessary

B)not necessary

C)not necessary

8)

Mutex lock= 1, Mutex readers =n , Mutex hold = 1

Read:

```
P(hold)
P(readers)
if (readers == n-1 ) P(lock)
V(hold)
//read
V(readers)
if(readers == n) V(lock)
```

Write:

```
P(hold)
P(lock)
//write
V(lock)
```

V(hold)

9)

Files provided you can run it calling: make test

matrix.cpp

```
#include <pthread.h>

#include <stdio.h>

#include <stdlib.h>

#include "mat.h"


using namespace std;

const int arr1Col = 1000;

const int arr1Row = 1000;

const int arr2Col = 1000;

const int arr2Row = 1000;

// check dimensions of matrices to check for size validity

int validmult()

{

      if( arr1Col == arr2Row) return 1;

      return 0;

}


/*

*     this function is the math behind finding one elem in the answer

*/

void calc(float* ans,const int row,const int col)

{
  float total = 0;
  for(int i = 1; i <= arr1Row ; i++)
  {
   total += arr1[(arr1Row *(row-1))+(i-1)] * arr2[(col-1)+(arr1Row*(i-
1))];
  }
  ans[((row-1)*arr1Row)+(col-1)] = total;
}

/*
*     prints out matrix
*/
void printar(const float* ans)
{
  for(int i = 1; i <= arr1Row*arr1Row; i++)
  {
    printf(" %f", ans[i-1]);
    if(i % arr1Row == 0) printf("\n");
  }
}

int main()
{

float Ans[arr1Row*arr1Row];

      if( validmult()== 0 )
      {
            printf("not valid size matrices");
      }
      int i;
      int j;
      for(i =1 ; i<=arr1Row; i++)
      {
            for(j = 1; j<=arr1Row; j++)
            {
              calc(Ans, i, j);
            }
      }
      printar(Ans);

      exit(0);
}
```

mt2.cpp

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <semaphore.h>
#include "mat.h"

using namespace std;
const int arr1Col = 1000;
const int arr1Row = 1000;
const int arr2Col = 1000;
const int arr2Row = 1000;
const int td = 4;
```

```
// allows only one writer at a time
sem_t mutex;

float ans[1000000];
// check dimensions of matrices to check for size validity
int validmult()
{
      if( arr1Col == arr2Row) return 1;
      return 0;
}

void calc(const int row,const int col)
{
  float total = 0;
  for(int i = 1; i <= arr1Row ; i++)
  {
   total += arr1[(arr1Row *(row-1))+(i-1)] * arr2[(col-1)+(arr1Row*(i-1))];
  }
  sem_wait(&mutex);
  ans[((row-1)*arr1Row)+(col-1)] = total;
  sem_post(&mutex);
}

/*
*     this function is the math behind finding one elem in the answer
*/
void *split(void *vargp)
```

```
{
  int i= *((int*)vargp);
  int j= *((int*)(vargp)+1);
  free(vargp);

  for(; ( (i<=j)  || ( (i + (arr1Row/4) > arr1Row) && (i > (arr1Row -
(arr1Row/4)))))&& (i <= arr1Row); i++)
      {
            int q= 1;
            for(; q<=arr1Row; q++)
            {
              calc(i, q);
            }
      }
  return NULL;
}



/*
*     prints out matrix
*/
void printar(const float* ans)
{
  for(int i = 1; i <= arr1Row* arr1Row; i++)
  {
    printf(" %f", ans[i-1]);
    if(i % arr1Row == 0) printf("\n");
  }
}
```

```
int main()
{
      pthread_t tid[td];
      int i;
      int q;

      //init sema
      sem_init(&mutex, 0, 1);

      // check if this mult is allowable
      if( validmult()== 0 )
      {
            printf("not valid size matrices");
      }
      for(i = 1, q= 0 ; i<= td; i++, q++)
      {
              int *serv = (int *)malloc(sizeof(int) * 2);
              *serv = 1 +((i-1)* (arr1Row/4)) ;
              *(serv+1) = (i * (arr1Row/4));
              pthread_create(&tid[q],NULL,split,serv);
      }
      for(int z = 0; z < td ; z++)
      {
            pthread_join(tid[z],NULL);
      }
      printar(ans);

      exit(0);
}
```

Note: you need mat.h which holds two 1000000 float arrays filled with values