```c
/*****************************************************************************
Authors  : Albert Owusu-Asare , Box 4497, <owusuasa@grinnell.edu>
         : Ezra Edgerton      , Box 3503, <edgerton@grinnell.edu>

This document contains answers to questions from CS213 Lab: Threads


Questions:
http://www.cs.grinnell.edu/~weinman/courses/CSC213/2014F/labs/threads.html

 modified code has the MODIFIED keyword
Original author is cited below :
 * Jerod Weinman
 * 21 May 2008
*****************************************************************************/

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

#include "matrix.h"
#include "matrixop.h"

#define SUB2IND(row,col,numcols) (row) * (numcols) + (col)

/* Check the dimensions of three matrices for the sum C=A+B
 *
 * Returns -1 if there is a dimension mismatch, and zero otherwise. */

int mtxCheckAddDim(const struct matrix_t *a, const struct matrix_t *b,
                        const struct matrix_t *c)
{

     if (a->rows != b->rows || a->rows != c->rows  ||
         a->cols != b->cols || a->cols != c->cols ) {
          return -1;
     }

     return 0;

}


/* Matrix addition C = A + B
 *
 * Preconditions:
 *   Matrix parameters a,b and c all have the same dimension
 *
 * Postconditions:
 *   The resulting matrix sum is stored in parameter c
 *   Return value of 0 indicates successful completion of the addition
 */
int mtxAdd( const struct matrix_t *a, const struct matrix_t *b,
                  struct matrix_t *c) {
     /* Make sure these matrices are "addable" */
     int res = mtxCheckAddDim(a,b,c);

     if (res<0) {
          fprintf(stderr,"Dimension mismatch for matrix add.\n");
          return res;
     }

     int m = a->rows;        /* Dimensions of input matrices */
     int n = a->cols;

     MTX_TYPE *pa, *pb, *pc; /* Pointers to matrix data */

     pa = a->data;           /* Initialize pointers to beginning of data */
     pb = b->data;
     pc = c->data;

     /* Iterate over every entry in the matrices, adding and storing the result */
     int i,j;
     for (i=0 ; i<m ; i++)
          for (j=0 ; j<n ; j++, pa++,pb++,pc++)
               *pc = *pa + *pb;

     return 0; /* Indicate successful completion */
}



/* MODIFIED : This is an additional piece to the original
 * Check the dimensions of three matrices for the sum C=AB
 *
 * Returns -1 if there is a dimension mismatch, and zero otherwise. */

int mtxCheckMultiplyDim(const struct matrix_t *a, const struct matrix_t *b,
                        const struct matrix_t *c)
{

     if (a->cols != b->rows || a->rows != c->rows
         || b->cols != c->cols ) {
          return -1;
     }

     return 0;

}



/*
 * MODIFIED
 * Matrix multiplication C = AB
 *
 * Preconditions:
 *   Matrix parameters a,b,c have the required dimension.
    mtxCheckMultiplyDim(a,b,c) returns >0
 *
 * Postconditions:
 *   The resulting matrix sum is stored in parameter c
 *   Return value of 0 indicates successful completion of the multiplication
 */

int mtxMultiplyMin( const struct matrix_t *a, const struct matrix_t *b,
                    struct matrix_t *c)
{
     /* Make sure these matrices are "multiply-able" */
     int res = mtxCheckMultiplyDim(a,b,c);

     if (res<0) {
          fprintf(stderr,"Dimension mismatch for matrix add.\n");
          return res;
     }

     int m = a->rows;        /* Dimensions of input matrices */
     int n = a->cols;
     int p = b->cols;


     MTX_TYPE *pa, *pb, *pc; /* Pointers to matrix data */

     pa = a->data;           /* Initialize pointers to beginning of data */
     pb = b->data;
```

```
    pc = c->data;



    /* Iterate over every entry in the matrices, adding and storing the result */
    int i,j,k, l;

    /* initialise elements of matrix c to 0*/

    for (l = 0; l < (m*p); l++, pc++)
      {
        *pc = 0;
      }
    pc = c->data; // point back to start of c matrix


    for (i = 0; i < m; i++,  pc += SUB2IND(1,0,p), pb = b->data)
      //i loops through the rows of a and the rows of c
      {
        for (k = 0; k < n; k++, pa++, pc = pc - p)
          //k loops through the columns of a and the rows of b
          {
            for (j = 0; j < p; j++,pb++,pc++)
              //the columns of b and the columns of c
              {
                *pc += *pa * *pb;

              }
          }
      }

    return 0; /* Indicate successful completion */
}
/*
 * MODIFIED
 * Does the matrix multiplication work for one thread
 * preconditions:  none
 * postconditions : matrix c is modified such that each of its values agrees with m
atrix
 * multiplication of c = a * b.
 *
 * much of the code involving the creation and running of threads was taken from th
e POSIX
 * Threads Programming tutorial:
 * https://chttps://computing.llnl.gov/tutorials/pthreads/#Joining
 * computing.llnl.gov/tutorials/pthreads/#PassingArguments
 */

void* threadMtxMultiplyMin(void *multParam)
{
  struct matrixThreadParam_t  *data;
  data =  ( struct matrixThreadParam_t *) multParam ;

  /* Dimensions of input matrices */
  int m = data->a->rows;
  int n = data->a->cols;
  int p = data->b->cols;


  int stride = data->numThreads;
  int ID = data->threadId;

/* Pointers to matrix data */
  MTX_TYPE *pa, *pb, *pc;

/* Initialize pointers to beginning of data */
  pa = data->a->data;
  pb = data->b->data;
  pc = data->c->data;

  /* Iterate over every entry in the matrices, adding and storing the result */
  int i,j,k;

  pc += SUB2IND(ID, 0, p);
  pa += SUB2IND(ID, 0, n);


  for (i = ID; i < m;
       i= i + stride, pc += SUB2IND(stride, 0, p), pb = data->b->data,
         pa += SUB2IND(stride-1, 0 , n))
    //i loops through the rows of a and the rows of c
    {

      for (k = 0; k < n; k++, pa++, pc = pc - p)
        //k loops through the columns of a and the rows of b
        {
          for (j = 0; j < p; j++,pb++,pc++)
            //the columns of b and the columns of c
            {
              *pc += *pa * *pb;

            }
        }
    }

  return NULL; /* Indicate successful completion */
}

/*
 * MODIFIED
 * Computes the threaded concurrent matrix product.
 * preconditions:  none
 * postconditions : matrix c is modified such that each of its values agrees with m
atrix
 * multiplication of c = a * b.
 *
 * much of the code involving the creation and running of threads was taken from th
e POSIX
 * Threads Programming tutorial:
 * https://chttps://computing.llnl.gov/tutorials/pthreads/#Joining
 * computing.llnl.gov/tutorials/pthreads/#PassingArguments
 */

int parMtxMultiply( const struct matrix_t *a,
                    const struct matrix_t *b,
                    struct matrix_t *c,
                    int numThreads)
{
  //checks to see if matrixes are multipliable
 int res = mtxCheckMultiplyDim(a,b,c);

     if (res<0) {
          fprintf(stderr,"Dimension mismatch for matrix add.\n");
          return res;
     }

     /* initialise elements of matrix c to 0*/
     int l;
     MTX_TYPE *pc;
     pc = c->data;

     for (l = 0; l < (a->rows * b->cols); l++, pc++)
       {
         *pc = 0;
       }
     pc = c->data; // point back to start of c matrix
```

```
    //creates threads according to numThreads
    int t;
    int rc; //gets return value when thread is created and when thread is joined
    struct matrixThreadParam_t thread_matrix_array[numThreads];
    pthread_t threads[numThreads];

    pthread_attr_t attr;
    void *status;

    //initialization of attr to make threads joinable
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

    for (t = 0; t < numThreads; t++)
      {
        //set elements of thread_array[t]
        thread_matrix_array[t].a = a;
        thread_matrix_array[t].b = b;
        thread_matrix_array[t].c = c;
        thread_matrix_array[t].numThreads = numThreads;
        thread_matrix_array[t].threadId = t;
        rc = pthread_create(&threads[t], &attr, threadMtxMultiplyMin,
                            (void *) &thread_matrix_array[t]);
        if (rc) {
          fprintf (stderr, "Could not create thread %d\n", t);
          return EXIT_FAILURE;
        }
      }
    //join threads
    pthread_attr_destroy(&attr);
    for (t = 0; t < numThreads; t++)
      {
        rc =  pthread_join(threads[t], &status);
        if (rc)
          {
            fprintf(stderr, "Thread %d could not join\n", t);
            return EXIT_FAILURE;
          }
      }
    return 0;
}
```