```c
/***************************************************************************
*  FILE: pmms.c
*  AUTHOR: Connor Beardsmore - 15504319
*  UNIT: OS200 Assignment S1 - 2016
*  PURPOSE: Matrix multiplication using multithreading and POSIX mutexs
*  LAST MOD: 07/05/16
*  REQUIRES: pmms.h
***************************************************************************/

#include "pmms.h"

//---------------------------------------------------------------------------

int main(int argc, char* argv[])
{
    // ENSURE ONLY 6 COMMAND LINE ARGUMENTS ENTERED
    if ( argc != 6 )
    {
        printf( "Usage: ./pmms[Matrix A File] [Matrix B File] [M] [N] [K]\n" );
        printf( "Please see README for detailed steps on how to run!\n" );
        return -1;
    }

    // RENAME COMMAND LINE ARGUMENTS FOR CODE READABILITY
    char* fileA = argv[1];
    char* fileB = argv[2];
    M = atoi( argv[3] );
    N = atoi( argv[4] );
    K = atoi( argv[5] );
    status = 0;

    // VALIDATE THAT M,N,K ARE ALL 1 OR MORE
    if ( ( M < 1 ) || ( N < 1 ) ||  ( K < 1 ) )
    {
        printf( "ERROR - Matrix dimensions must bee positive value.\n" );
        return -1;
    }

    // MAP MATRICES STRUCT TO ADDRESS SPACE, ASSIGN TO POINTERS
    first = (int*)malloc( M * N * sizeof(int) );
    second = (int*)malloc( N * K * sizeof(int) );
    product = (int*)malloc( M * K * sizeof(int) );

    // READ DATA FROM FILE INTO MATRIX SHARED MEMORY
    // ERROR CHECK TO CONFIRM THAT BOTH WORKED AS EXPECTED
    status = readFile( fileA, first, M, N );
    if ( status != 0 )
    {
        freeMatrices( first, second, product );
        return -1;
    }
    status = readFile( fileB, second, N, K );
    if ( status != 0 )
    {
        freeMatrices( first, second, product );
        return -1;
    }

    // INITIAL SUBTOTAL FIELDS TO "EMPTY"
    subtotal.value = SUBTOTAL_EMPTY;
    subtotal.threadID = SUBTOTAL_EMPTY;
    subtotal.rowNumber = SUBTOTAL_EMPTY;

    // CREATE M THREADS IN A MALLOC'D ARRAY
    pthread_t* producers = (pthread_t*)malloc( sizeof(pthread_t) * M );

    // INITIALISE THE SEMAPHORES
    status = createLocks(locks);
    if ( status != 0 )
    {
        fprintf( stderr, "ERROR - creating POSIX mutex + conditions\n");
        freeMatrices( first, second, product );
        free( producers );
        return -1;
    }

    // THE 'M' CREATED THREADS EXECUTE PRODUCER FUNCTION
    // NO THREAD SPECIFIC DATA IS REQUIRED
    for ( int ii = 0; ii < M; ii++ )
    {
        pthread_create( &producers[ii], NULL, producer, NULL );
        // AUTOMATICALLY RELEASE SYSTEM RESOURCES UPON THREAD EXITING
        pthread_detach( producers[ii] );
    }

    // PARENT THREAD EXECUTES CONSUMER FUNCTION
    consumer(NULL);

    // PARENT DESTORYS ALL SEMAPHORES
    status = destroyLocks(locks);
    if ( status != 0 )
    {
        fprintf( stderr, "ERROR - destroying POSIX mutex + conditions\n");
        freeMatrices( first, second, product );
        free( producers );
        return -1;
    }

    // OUTPUT FINAL TOTAL
    printf( "Total: %d\n", grandTotal );

    // FREE ALL MALLOC'D MEMORY
    freeMatrices( first, second, product );
    free( producers );

    return 0;
}
//------------------------------------------------------------------------------
// FUNCTION: producer
// PURPOSE: Parent process consumes the subtotal + childPID create by children.

void* producer()
{
    int rowNumber = 0;
    int total = 0;
    int value;

    // THREAD DETERMINES WHICH ROW TO CALCULATE
    // MUTEX REQUIRED TO ACCESS rowNumber, SO EACH THREAD HAS DISTINCT VALUE
    pthread_mutex_lock( &locks.mutex );
        rowNumber = subtotal.rowNumber;
        subtotal.rowNumber = subtotal.rowNumber + 1;
    pthread_mutex_unlock( &locks.mutex );

    // CALCULATE OFFSETS TO CONVERT 1D ARRAYS TO VIRTUAL 2D
    int offsetA = rowNumber * N;
    int offsetC = rowNumber * K;

    for ( int ii = 0; ii < K; ii++ )
    {
        value = 0;

        // CALCULATE ROW DATA
        for ( int jj = 0; jj < N; jj++ )
            value += first[offsetA + jj] * second[jj * K + ii];

        product[offsetC + ii] = value;
    }

    // CALCULATE TOTAL OF ALL ELEMENTS IN ROW
    for ( int kk = 0; kk < K; kk++ )
        total += product[offsetC + kk];

    // WAIT FOR LOCK BEFORE ACCESSING SHARED DATA
    pthread_mutex_lock( &locks.mutex );
    while ( subtotal.value != 0 )
        // GIVE UP MUTEX LOCK WHILE WAITING FOR CONDITION
        pthread_cond_wait( &locks.empty, &locks.mutex );

        subtotal.value = total;
        subtotal.threadID = pthread_self();

    pthread_cond_signal( &locks.full );
    pthread_mutex_unlock( &locks.mutex );

    // THREAD FINISHES ONCE ROW CALCULATED
    pthread_exit(0);
}

//----------------------------------------------------------------------------
// FUNCTION: consumer
// PURPOSE: Parent process consumes the subtotal + threadID create by thread.

void* consumer()
{
    grandTotal = 0;

    for ( int ii = 0; ii < M; ii++ )
    {
    // WAIT FOR LOCK BEFORE ACCESSING SHARED DATA
        pthread_mutex_lock( &locks.mutex );
        while ( subtotal.value == 0 )
            // GIVE UP MUTEX LOCK WHILE WAITING FOR CONDITION
            pthread_cond_wait( &locks.full, &locks.mutex );

            // OUTPUT ROW TOTAL AND RESET SUBTOTAL VALUES
            printf( "Subtotal produced by thread with ID " );
            printf( "%ld: %d\n", subtotal.threadID, subtotal.value );
            grandTotal += subtotal.value;
            subtotal.value = SUBTOTAL_EMPTY;
            subtotal.threadID = SUBTOTAL_EMPTY;

            pthread_cond_signal( &locks.empty );
        pthread_mutex_unlock( &locks.mutex );
    }

    return NULL;
}

//----------------------------------------------------------------------------
// FUNCTION: createLocks
// EXPORT: status (int)
// PURPOSE: Initialise the Mutex and Conditions used for locks

int createLocks()
{
    // IF ANY METHOD FAILS, STATUS WILL BE NON-ZERO
    int status = 0;
    status += pthread_mutex_init( &locks.mutex, NULL );
    status += pthread_cond_init( &locks.full, NULL );
    status += pthread_cond_init( &locks.empty, NULL );
    return status;
}

//----------------------------------------------------------------------------
// FUNCTION: destroyLocks
// EXPORT: status (int)
// PURPOSE: Destroy the Mutex and Conditions used for locks

int destroyLocks()
{
    // IF ANY METHOD FAILS, STATUS WILL BE NON-ZERO
    int status = 0;
    status += pthread_mutex_destroy( &locks.mutex );
    status += pthread_cond_destroy( &locks.full );
    status += pthread_cond_destroy( &locks.empty );
    return status;
}

//---------------------------------------------------------------------------
// FUNCTION freeMatrices
// IMPORT: first (int*), second (int*), third (int*)
// PURPOSE: Free's the malloc'd member associated with the matrices imported

void freeMatrices(int* first, int* second, int* product)
{
    free(first);
    free(second);
    free(product);
}

//---------------------------------------------------------------------------
// FUNCTION: printMatrix()
// IMPORT: newMatrix (Matrix*)
// PURPOSE: Print matrix contents to std out for debugging purposes

void printMatrix(int* matrix, int rows, int cols)
{
    // OFFSET TO CALCULATE "ROWS" OF THE 1D ELEMENT ARRAY
    int offset = 0;
    printf("\n");

    // ITERATE OVER ENTIRE MATRIX AND PRINT EACH ELEMENT
    for ( int ii = 0; ii < rows; ii++ )
    {
        offset = ii * cols;
        for ( int jj = 0; jj < cols; jj++ )
        {
            printf("%d ", matrix[ offset + jj ] );
        }
        printf("\n");
    }
}

//--------------------------------------------------------------------------
// FUNCTION: printMatrices
// IMPORT: first (int*), second (int*), product (int*)
// PURPOSE: Prints the contents of three different Matrices to std out

void printMatrices(int* first, int* second, int* third, int M, int N, int K)
{
        printMatrix(first, M, N);
        printMatrix(second, N, K);
        printMatrix(third, M, K);
}

//--------------------------------------------------------------------------
```