```
# Makefile For Matrix Multiplication w. Multithreading
# OS200 Assignment
# Last Modified: 07/05/16
# Connor Beardsmore - 15504319

# MAKE VARIABLES
EXEC1 = pmms
OBJ1 = pmms.o fileIO.o
CFLAGS = -Wall -Wextra -std=c99 -lrt -pthread -D _XOPEN_SOURCE=500
CC = gcc


# RULES + DEPENDENCIES
$(EXEC1) : $(OBJ1)
	$(CC) $(OBJ1) -o $(EXEC1) $(CFLAGS)

pmms.o : pmms.c pmms.h fileIO.h
	$(CC) -c pmms.c $(CFLAGS)

fileIO.o : fileIO.c fileIO.h
	$(CC) -c fileIO.c $(CFLAGS)

clean:
	rm -f $(EXEC1) $(OBJ1)

```

```c
/****************************************************************************
*  FILE: pmms.h
*  AUTHOR: Connor Beardsmore - 15504319
*  UNIT: OS200 Assignment S1 - 2016
*  PURPOSE: Header file for pmms.c
*  LAST MOD: 07/05/16
*  REQUIRES: stdlib.h, pthread.h, fileIO.h
***************************************************************************/

#pragma once

#include <stdlib.h>
#include <pthread.h>
#include "fileIO.h"

//----------------------------------------------------------------------------
// CONSTANTS

#define SUBTOTAL_EMPTY 0

//----------------------------------------------------------------------------
// STRUCT: Stores the value of subtotal and the ID of the thread that
//         created it. Also stores row number that the thread calculated.

typedef struct
{
    int value;
    int rowNumber;
    long threadID;
} Subtotal;

//----------------------------------------------------------------------------
// STRUCT: Stores 3 locks for use in the producer-consumer problem.
//         Mutex provides mutual exclusion to data.
//         Full and empty are conditions that the producer and consumer
//         wait until they are met.

typedef struct
{
    pthread_mutex_t mutex;
    pthread_cond_t full;
    pthread_cond_t empty;
} Synchron;

//----------------------------------------------------------------------------
// GLOBAL VARIABLES FOR USE IN MULTITHREADS

Subtotal subtotal;
Synchron locks;
int grandTotal;
int status;

// MATRIX POINTERS AND DIMENSIONS
int* first;
int* second;
int* product;
// SEE README FOR WHAT THESE VARIABLES REPRESENT (AND REASON FOR NAMING)
int M;
int N;
int K;

//----------------------------------------------------------------------------
// FUNCTION DECLARATIONS

void* producer();
void* consumer();
int destroyLocks();
int createLocks();
void freeMatrices(int*, int*, int*);
void printMatrix(int*, int, int);
void printMatrices(int*, int*, int*, int, int, int);

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

```

```c
/**************************************************************************
*  FILE: pmms.c
*  AUTHOR: Connor Beardsmore - 15504319
*  UNIT: OS200 Assignment S1 - 2016
*  PURPOSE: Matrix multiplication using multithreading and POSIX locks
*  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 POSITIVE VALUES
    if ( ( M < 1 ) || ( N < 1 ) ||  ( K < 1 ) )
    {
        printf( "ERROR - Matrix dimensions must be positive value\n" );
        return -1;
    }

    // MAP MATRICES ARRAYS 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();

    // 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, ENSURES 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;

    // ACTUAL MULTIPLICATION CALCULATIONS
    // SEE README FOR HOW THIS IS PERFORMED
    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 created by thread.

void* consumer()
{
    grandTotal = 0;

    // LOOP M TIMES FOR EACH ROW OF PRODUCT MATRIX
    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;
```

```c
}

//---------------------------------------------------------------------------
// 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 arrays associated with the matrices imported

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

//---------------------------------------------------------------------------
// FUNCTION: printMatrix()
// IMPORT: matrix (int*), rows (int), cols (int)
// PURPOSE: Print matrix contents to stdout for debugging purposes

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

    // 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*), M,N,K (int)
// PURPOSE: Prints the contents of three different Matrices to stdout

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);
}

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

```

```c
/***************************************************************************
*  FILE: fileIO.h
*  AUTHOR: Connor Beardsmore - 15504319
*  UNIT: OS200 Assignment S1 - 2016
*  PURPOSE: Header file for fileIO.c
*  LAST MOD: 07/05/16
*  REQUIRES: stdio.h
***************************************************************************/

#pragma once
#include <stdio.h>

//--------------------------------------------------------------------------
// FUNCTION DECLARATIONS

int readFile(char*, int*, int, int);

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

```c
/****************************************************************************
*  FILE: fileIO.c
*  AUTHOR: Connor Beardsmore - 15504319
*  UNIT: OS200 Assignment S1 - 2016
*  PURPOSE: Perform reading of matrix elements from a file
*  LAST MOD: 07/05/16
*  REQUIRES: fileIO.h
****************************************************************************/

#include "fileIO.h"

//---------------------------------------------------------------------------
// FUNCTION: readFile()
// IMPORT: filename (char*), matrix (int*), rows (int), cols (int)
// EXPORT: status (int)
// PURPOSE: Read matrix from file and store its elements in int array

int readFile(char* filename, int* matrix, int rows, int cols)
{
    int nRead;
    int offset = 0;

    // OPEN FILE AND CONFIRM NO ERRORS OCCURRED
    FILE* f = fopen( filename, "r" );
    if ( f == NULL )
    {
        perror( "ERROR - opening file!\n" );
        return -1;
    }

    // ITERATE TO FILL ALL MATRIX ROWS
    for ( int ii = 0; ii < rows; ii++ )
    {
        // ITERATE TO FILL ALL MATRIX COLS
        offset = ii * cols;
        for ( int jj = 0; jj < cols; jj++ )
        {
            nRead = fscanf( f, "%d", ( &matrix[offset + jj] ) );
            if ( nRead < 0 )
            {
                // CHECK THAT ENOUGH VALUES HAVE BEEN READ
                if ( (offset + jj) < (rows * cols - 1) )
                {
                    fprintf( stderr, "ERROR - not enough matrix values\n" );
                    return -1;
                }
                // CHECK THAT NO ERROR FORCED EARLY EXIT
                else if ( ferror(f) )
                {
                    perror( "ERROR - reading matrix file!\n" );
                    return -1;
                }
            }
        }
    }

    fclose(f);
    return 0;
}
//---------------------------------------------------------------------------

```

```bash
#! /bin/bash

# AUTHOR: Connor Beardsmore
# DATE: 07/04/16

# COMPILE PROGRAM
make pmms

# RUN PMMS PROGRAM FOR COMBINATION OF M N K VALUES
# FROM 1 TO 100

for i in {1..100};
do
    for j in {1..100};
    do
        for k in {1..100};
        do
            echo $i $j $k
            ./pmms allOnes.txt allOnes.txt $i $j $k
        done
    done
done

```