// 2D FFT Using threads
// George F. Riley, Georgia Tech, Fall 2009
// This illustrates how a mutex would be implemented using Leslie Lamport’s
// "Bakery Algorithm". This algorithm implements a correct mutex
// without any specific "atomic" instruction support from the hardware.

#include <iostream>
#include "pthread.h"
#include "math.h"
#include <sys/time.h>
#include "complex.h"
#include "InputImage.h"

using namespace std;

// Define a helper class to compare a tuple (number, thread-id)
// for less than.
class NumberId {
public:
    NumberId(int n, int t) : number(n), threadId(t) {}
    bool operator< (const NumberId& rhs);
public:
    int number;
    int threadId;
};

bool NumberId::operator < (const NumberId& rhs) {
    // Less than if lhs.number < rhs.number, or
    // if lhs.number == rhs.number AND lhs.threadId < rhs.threadId
    if (number < rhs.number) return true;
    if (number == rhs.number && threadId < rhs.threadId) return true;
    return false;
}

class BakeryMutex {
public:
    BakeryMutex(int nThreads);
    void Lock(int myId);  // Lock the mutex
    void UnLock(int myId);  // UnLock the mutex
private:
    int N;  // Number of threads
    bool* choosing;  // True if choosing a ticket, one per thread
    int* number;  // Ticket number chosen, 0 if no ticket
};
BakeryMutex::BakeryMutex(int nThreads)
 : N(nThreads)
{
    // Allocate the two thread specific values, "choosing" and "number"
    choosing = new bool[N];
    number = new int[N];
    // Initialize
    for (int i = 0; i < N; ++i)
    {
        choosing[i] = false;
        number[i] = 0;
    }
}

void BakeryMutex::Lock(int myId)
{
    // First note that we are in the process of "choosing" a ticket number
    choosing[myId] = true;
    // Find the maximum already chosen, and pick that number + 1
    int maxTicket = 0;
    for (int i = 0; i < N; ++i)
    {
        if (number[i] > maxTicket) maxTicket = number[i];
    }
    // Set my number to the maxTicket + 1
    number[myId] = maxTicket + 1;
    // Indicate we are no longer choosing
    choosing[myId] = false;
    // Now defer to anyone with a smaller ticket. If we have ties
    // (choosing the same ticket) defer if their threadId is
    // less than ours
    for (int i = 0; i < N; ++i)
    {
        while(choosing[i]) {} // Spin if someone else is choosing
        while(number[i] != 0 &&
            NumberId(number[i], i) < NumberId(number[myId], myId))
        {
            // Spin while some other thread has a lower ticket number
        }
    }
    // At this point, we have the lowest ticket number and have essentially
    // claimed the lock.
}

void BakeryMutex::UnLock(int myId)
{
    // Release our ticket number
    number[myId] = 0;
}

Program threaded-fft-bakery.cc (continued)
// We use global variables in lieu of member variables for this example
Complex** h; // Points to the 2D array of complex (the input)
Complex* W; // Weights (computed once in main
unsigned N; // Number of elements (both width and height)
unsigned nThreads; // Desired number of threads
unsigned activeCount = 0; // Number of active threads

// pthread variables
// We will replace the activeMutex and coutMutex with our
// implementation to observe effects. We can’t replace the exit mutex
// since it is needed for the condition variable (which we did not
// implement a replacement for.
BakeryMutex* activeMutex;
pthread_mutex_t exitMutex;
pthread_cond_t exitCondition;
BakeryMutex* coutMutex;

// Add a verbose flag to turn on/off extra outputs
bool verbose = false;

// Helper routines
void DumpTransformedValues() { // Code omitted for brevity
}

void TransposeInPlace(Complex** m, int wh) { // code omitted for brevity
}

void LoadWeights() { // Compute the needed W values. Omitted for brevity
}

void Transform1D(Complex* h) { // The simple 1D transform we did earlier. Code omitted for brevity
}

Program threaded-fft-bakery.cc (continued)
void * FFT_Thread(void * v)
{
    unsigned long myId = (unsigned long)v; // My thread number
    unsigned rowsPerCPU = N / nThreads;
    unsigned myFirstRow = myId * rowsPerCPU;
    // We have to do a mutex around the "activeCount++". Why?
    activeMutex->Lock(myId);
    activeCount++;
    activeMutex->Unlock(myId);
    if (verbose)
    {
        coutMutex->Lock(myId);
        cout << "MyId is " << myId << " myFirstRow " << myFirstRow << endl;
        coutMutex->Unlock(myId);
    }
    // Call the 1D FFT on each row
    for (unsigned i = 0; i < rowsPerCPU; ++i)
    {
        Transform1D(h[myFirstRow + i]);
    }
    // Now notify the main thread we have completed the rows
    pthread_mutex_lock(&exitMutex); // Insure only one thread signals the exit
    activeMutex->Lock(myId); // Insure only one thread changes active
    activeCount--;
    activeMutex->Unlock(myId);
    // Don’t need cout mutex here. Why?
    cout << "Thread " << myId << " exited, activeCount " << activeCount << endl;
    if (activeCount == 0)
    { // We are the last thread to exit. Signal the main thread
        // that all threads are done
        pthread_cond_signal(&exitCondition);
    }
    pthread_mutex_unlock(&exitMutex);
    return 0;
}
int main( int argc, char** argv)
{
    verbose = argc > 3;
    InputImage image(argv[1]);
    nThreads = atol(argv[2]); // Number of threads
    N = image.GetHeight(); // Assume square, width = height
    h = image.GetRows(0, N); // In this case, we get all rows

    // Start the timer here, after loading the image
    struct timeval tp;
    gettimeofday(&tp, 0);
    double startSec = tp.tv_sec + tp.tv_usec/1000000.0;

    LoadWeights(); // Only need to do this once

    // Initialize the BakeryMutexes
    activeMutex = new BakeryMutex(nThreads + 1);
    coutMutex = new BakeryMutex(nThreads + 1);

    // Initialize the pthread mutex and condition variables
    pthread_mutex_init(&exitMutex, 0);
    pthread_cond_init(&exitCondition, 0);

    // We lock the exitMutex to be sure no threads exit until
    // all threads created, and we are waiting on the condition signal
    pthread_mutex_lock(&exitMutex);

    // Create the threads
    for (unsigned i = 0; i < nThreads; ++i)
    {
        pthread_t t;
        pthread_create(&t, 0, FFT_Thread, (void*)i);
    }

    // Now wait for them to finish pass 1
    pthread_cond_wait(&exitCondition, &exitMutex);
    if (verbose) cout << "All threads finished pass 1" << endl;

    // Transpose the matrix and schedule threads to do rows again
    TransposeInPlace(h, N);

    // Start the threads again
    for (unsigned i = 0; i < nThreads; ++i)
    {
        pthread_t t;
        pthread_create(&t, 0, FFT_Thread, (void*)i);
    }

    // Now wait for them to finish pass 2
    pthread_cond_wait(&exitCondition, &exitMutex);
    if (verbose) cout << "All threads finished pass 2" << endl;

    // Transpose back and write results
    TransposeInPlace(h, N);
    gettimeofday(&tp, 0);
    cout << "Calculated FFT " << (tp.tv_sec+tp.tv_usec/1000000.0) - startSec << " seconds" << endl;
    DumpTransformedValues();
}