// 2D FFT Using threads
// George F. Riley, Georgia Tech, Fall 2009
// This illustrates how a mutex would be implemented (both a "buggy" version
// and a good one).

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

using namespace std;

// This uses the Intel "Atomic Exchange" to help with mutex locking
uint32_t atomic_xchg32(volatile uint32_t * mem, uint32_t val)
{
    uint32_t prev = val;
    asm volatile ("lock; xchgl %0, %1"
                 : "=r" (prev)
                 : "m" (* (mem)), "0"(prev)
                 : "memory");
    return prev;
}

class BuggyMutex {
public:
    BuggyMutex();
    void Lock(); // Lock the mutex
    void UnLock(); // UnLock the mutex
private:
    // True if locked
    bool locked;
};

BuggyMutex::BuggyMutex()
: locked(false)
{ // Mutex is initially not locked
}

void BuggyMutex::Lock()
{ // This is buggy! Why?
    // See if another thread has the mutex locked. If so, just "spin"
    while(locked)
    { // Do nothing.. This is called "spinning"
    }
    // Lock is free, lock it and exit
    locked = true;
}

void BuggyMutex::UnLock()
{ // Free the mutex
    locked = false;
}

Program threaded-fft-mutex.cc
class GoodMutex {
public:
    GoodMutex();
    void Lock(); // Lock the mutex
    void UnLock(); // UnLock the mutex
private: // True if locked
    bool locked;
};

GoodMutex::GoodMutex()
    : locked(false) // Mutex is initially not locked
{}

void GoodMutex::Lock()
{
    // Use the atomic exchange to implement the locked flag
    while(atomic_xchg32((uint32_t*) &locked, true))
    {
        // Another thread has the lock, do nothing. This is called "spinning"
    }
    // Lock is set true by atomic exchange, so nothing more needed
}

void GoodMutex::UnLock()
{
    // Free the mutex
    locked = false;
}

// 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
// two implementations 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.
BuggyMutex activeMutex;
//GoodMutex activeMutex;
pthread_mutex_t exitMutex;
b乎read_mutex_t exitMutex;
b乎read_cond_t exitCondition;
BuggyMutex coutMutex;
//GoodMutex 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
}
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();
    activeCount++;
    activeMutex.UnLock();
    if (verbose)
    {
        coutMutex.Lock();
        cout << "MyId is " << myId << " myFirstRow " << myFirstRow << endl;
        coutMutex.UnLock();
    }
    // 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(); // Insure only one thread changes active
    activeCount--;
    activeMutex.UnLock();
    // 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 pthread mutex and condition variables
    // The buggyMutex variables are initialized in constructor
    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();
}

Program threaded-fft-mutex.cc (continued)