Nicholas Grigorian
ngg3vm
Machine Problem 2
All code can be viewed at http://github.com/GrigorianNick/OSBarrierHomework

This machine problem tasked us with finding the maximum value in a list of N data points by using N/2 pthreads and a custom barrier class. The barrier class has a relatively simple function description: it is initialized to some value V. Threads then call its wait() function, where they are blocked. When the Vth thread calls wait, then all the threads that called wait() are woken up. The goal is to provide a tool to elegantly synchronize multiple threads. My implementation of the barrier class works. I have only tested on my own desktop, so I don't know if it will work in the virtual machine.

To approach this problem, I initially divided labour up four ways. The Max class would keep track of the list data and act as a stable access point for the various threads. The Thread class would be a pthread wrapper, simplifying and automating how pthreads are handled. The last class is the Barrier class, a fairly minimalistic class to act as the rendezvous point for the pthreads. Finally main.cpp would handle initialization, reading user input, starting the comparison pthreads, and printing out the final solution.

Although the initial approach is conceptually clean, implementing it proved to be a major hassle. Specifically, pthread_create() is very particular about which functions it will spawn a thread on. I tried making it work with Thread's compare member function, but eventually settled for pulling that function out of Thread and into main.cpp.

The Barrier class I implemented only uses mutexes. The program flow is surprisingly succinct: when a thread wants to rendezvous with the other threads associated with this particular barrier, it calls the Barrier's enq function. First, it will enter a critical zone where it will decrement count. If count is not zero, then that means not all the threads have arrived at the Barrier. So the thread will leave the critical zone and block itself by attempting to gain a lock on the gate mutex. If, however, the count is zero then that means all the threads have rendezvoused and need to be woken up. So the final thread will leave the critical section and unlock the gate mutex. This will wake one of the threads, which will then immediately unlock the gate mutex after gaining its lock. This will repeat until all threads have unblocked themselves. At this point the Barrier is broken and cannot be used anymore. For this particular assignment, I inserted an extra step into the final thread. Inbetween leaving the critical section and unlocking the gate mutex, it will copy Max's lag array (which holds the answers to the latest round of comparisons) over Max's data array. Then it will create a new lag array half the length of the data array. This is done to create a sliding window for the threads to read and write their solutions.

There is a critical weakness in my Barrier design: it doesn't perform cross-Barrier deadlock checks. It assumes that all threads won't change Barrier access order. For example, if there are two Barriers, A and B, then all threads must rendezvous at Barrier A then Barrier B. If a thread decides to invert this, by enq'ing with Barrier B before Barrier A, then neither Barrier will be broken.

The first thing I verified is whether or not my Barrier class was operational. To do this, I had each thread print a unique message while it was in Barrier's enq function. If the thread was to be blocked, then I had it print out the Barrier's count right before it released the count mutex and started blocking. If the thread was going to trigger the Barrier to release its threads, then I had it print out the count number followed by a delineator pattern before unlocking the count mutex and releasing the gatekeeper mutex. This then created a human-friendly readout that showed each barrier counting down to its destruction. With the Barrier's functionality confirmed, I then proceeded onto timing the program. The result is a seemingly speedy sort. With 4096 data points stored in random.txt, running the command `cat random.txt | ./max` takes an average of 0.065s (as measured by the REAL output of time(1)) to produce an answer. However, a simpler algorithm (found in toy.cpp) managed to consistently clock in at 0.004s. My analysis of this discrepancy is that there are two possibilities:

1. My implementation leaves much to be desired and could be better optimized.
2. Multi-threading is hypothetically faster, but the real-world overhead incurred by switching between 2046 threads slows the actual runtime to a relative crawl.

In conclusion, I have written a minimal Barrier class to enable a multi-threaded comparison algorithm. Although it is theoretically faster than a linear algorithm, practical factors drastically inflates its runtime.

On my honor, as a student, I have neither received or given aid on this assignment.

```
/* barrier.h
 *
 * Author: Nicholas Grigorian
 * CompID: ngg3vm
 *
 * Summary:
 *      barrier.h is the header file for the Barrier class. Barriers are
 *          one-time use classes. After its count is initalized, it will wait for a
 *          certain number of threads to call the enq method. Once that number has been
 *          reached, it "breaks" and releases the attendent threads.
 *
 * Methods:
 *          enq:
 *                  enq is what threads call when they want to rendevous with other
 *                  threads. First the method decrements count. If count is above
 *                  zero, then that indicates not all threads have made it to the
 *                  rendevous point. Thus, the thread will lock on the alread
 *                  locked gate_mutex. If count is zero, then that indicates that
 *                  thread is the last to join. It will then perform any adjustments
 *                  to memory (in this case shuffling Max's variables around) then
 *                  it'll unlock the gate_mutex. This will give the lock to another
 *                  thread, which will immediatly unlock thus repeating the process
 *                  until all threads at the rendevous have been released.
 */

#include <iostream>
#include <pthread.h>
#include <stdlib.h>
#include "max.h"

using namespace std;

class Barrier {
        public:
                int count;
                void enq();
                Barrier(int init_count);
                ~Barrier();
                pthread_mutex_t count_mutex; // Lock on this whenever you want to touch
count.
                pthread_mutex_t gate_mutex; // Lock on this when you're waiting for
rendezvous. Unlock as soon as you get it.
};
```

```
Barrier.cpp:
#include "barrier.h"

using namespace std;

int Max::data_size;
int* Max::data;
int* Max::lag;

void Barrier::enq() {
	pthread_mutex_lock(&(this->count_mutex));
	count--;
	if (this->count <= 0) { // We're the nth thread to unlock. Time to let everybody advance.
		pthread_mutex_unlock(&(this->count_mutex)); // Releasing count mutex
because we don't need it.
		// Shift Thread.lag down into Thread.data and reprep Thread.lag
		if (Max::data_size > 1) { // We don't want prep Thread.lag to 0
			Max::data_size /= 2;
			free(Max::data);
			Max::data = Max::lag;
			Max::lag = NULL;
			Max::lag = (int*)malloc(sizeof(int) * (Max::data_size/2));
		}
		pthread_mutex_unlock(&(this->gate_mutex));
	}
	else {
		pthread_mutex_unlock(&(this->count_mutex));
		pthread_mutex_lock(&(this->gate_mutex)); // Waiting for the proper number of
threads to lock
		pthread_mutex_unlock(&(this->gate_mutex)); // We're free!
	}
}

Barrier::Barrier(int init_count) {
	pthread_mutex_init(&(this->gate_mutex), NULL);
	pthread_mutex_lock(&(this->gate_mutex)); // Erecting the barrier
	pthread_mutex_init(&(this->count_mutex), NULL);
	this->count = init_count;
}

Barrier::~Barrier() {
	pthread_mutex_destroy(&(this->gate_mutex));
	pthread_mutex_destroy(&(this->count_mutex));
}
```

```
/* main.cpp
 *
 * Author: Nicholas Grigorian
 * CompID: ngg3vm
 *
 * Compile:
 * `make`        : compiles max binary
 * `make gen`  : creates a txt file with 4096 random numbers
 * `make run`  : compiles max and gen binaries, creates a txt file with 4096 random numbers,
and pipes the random numbers into max.
 * `make clean`          : cleans up the working directory.
 */

#include <iostream>
#include <pthread.h>
#include <string>
#include <math.h>
#include "thread.cpp"

using namespace std;

// The recursive function that actually finds the largest number.
void *compare(void *arg) {
	Thread * thread = (Thread*)arg;
	// Copying the highest of the comparators in Max::data to Max::lag.
	if (Max::data[2 * thread->thread_id] > Max::data[(2 * thread->thread_id) + 1]) {
		Max::lag[thread->thread_id] = Max::data[2 * thread->thread_id];
	}
	else {
		Max::lag[thread->thread_id] = Max::data[(2 * thread->thread_id) + 1];
	}
	(thread->barrier_list[thread->barrier_counter])->enq();
	// We've compared values and moved them up an array. Now let's see if we need to
terminate.
	if ((thread->thread_id)%2 == 0) { // We're even, we get to live
		thread->barrier_counter++; // We're going to look at the next barrier
		thread->thread_id /= 2;
		if (Max::data_size == 1) {
			pthread_exit(NULL);
		}
		compare(thread); // Go for another round
	}
	else {
		pthread_exit(NULL);
	}
}

int main() {
	int num_of_nums = 0;
	string in;
	getline(std::cin, in);
	while(in != "") {
		num_of_nums++;
		Max::data = (int*)realloc(Max::data, sizeof(int*) * num_of_nums);
		Max::data[num_of_nums - 1] = atoi(in.c_str());
		getline(std::cin, in);
	}
	Max::data_size = num_of_nums;
	Max::lag = (int*)malloc(sizeof(int*) * (num_of_nums/2));
	Thread** threads = (Thread**)malloc(sizeof(Thread**) * (num_of_nums/2));
	for (int i = 0; i < num_of_nums/2; i++) {
		threads[i] = new Thread(i);
	}
	Barrier* barrier;
	// This math is supa-nasty
	for (int i = 1; i < num_of_nums/2 + 1; i*=2) { // i governs how
		barrier = new Barrier(num_of_nums/(2 * i));
		for (int j = 0; j < num_of_nums/2; j+=i) {
			threads[j]->subscribe(barrier);
		}
	}
	for (int i = 0; i < num_of_nums/2; i++) {
		pthread_create(&threads[i]->pthread, NULL, &compare, threads[i]);
	}
	pthread_join(threads[0]->pthread, NULL);
	cout << Max::data[0] << endl;
	return 0;
}
```

```
/* max.h
 *
 * Author: Nicholas Grigorian
 * CompID: ngg3vm
 *
 * Summary:
 *        max.h is the header file for the Max class. Max is used for shuttling
 *        data between threads.
 */
#include <iostream>

using namespace std;

class Max {

        public:
                static int* data;
                static int* lag;
                static int data_size;
};
```

```
/* thread.h
 *
 * Author: Nicholas Grigorian
 * CompID: ngg3vm
 *
 * Summary:
 *      thread.h is the header file for the Thread class. Originally meant to
 *      encapsulate pthreads, it evolved into a record-keeper for pthreads. It has
 *      two primary jobs: keep track of thread_id and to manage the Barrier
 *      subscriptions.
 *
 * Methods:
 *      subscribe:
 *              Adds a Barrier to the barrier list, thereby signing the object up for
 *              that particular barrier's breaking.
 */

#include <iostream>
#include <pthread.h>
#include "barrier.cpp"

using namespace std;

struct Thread {
        public: // Erythang's public
                pthread_t pthread;
                Barrier** barrier_list;
                int barrier_list_size;
                int barrier_counter;
                int thread_id;
                Thread(int id);
                void subscribe(Barrier* barrier); // Keep track of barriers we need to work
through
};
```

thread.cpp:

```
#include "thread.h"

using namespace std;

Thread::Thread(int id) {
	this->thread_id = id;
	barrier_list_size = 0;
	barrier_counter = 0;
	barrier_list = NULL;
}

// We want to pay attention to the barrier object being passed in
void Thread::subscribe(Barrier* barrier) {
	barrier_list_size++;
	barrier_list = (Barrier**)realloc(barrier_list, sizeof(barrier_list) * barrier_list_size);
	barrier_list[barrier_list_size - 1] = barrier;
}
```

toy.cpp:

```
#include <iostream>
#include <string>
#include <stdio.h>
#include <stdlib.h>

using namespace std;

int main() {
        int *data = NULL;
        int data_length = 0;
        int max = 0;
        string in;
        getline(std::cin, in);
        while(in != "") {
                data_length++;
              data = (int*)realloc(data, sizeof(int*) * data_length);
              data[data_length - 1] = atoi(in.c_str());
              getline(std::cin, in);
        }
        for (int i = 0; i < data_length; i++) {
                if (data[i] > max) {
                        max = data[i];
                }
        }
        cout << max << endl;
}
```

gen.cpp:

```
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

using namespace std;

int main() {
	int thing;

	srand(time(NULL));

	for (int i = 0; i < 4096; i++) {
		thing = rand() % 10000;
		cout << thing << endl;
	}
	cout << endl << endl;
}
```