Write up

Comments

Code

Operating Systems Homework #3

Gabriel Groover, gtg3vv@virginia.edu

I was able to successfully implement a parallel, binary reduction in C++.

1. Problem

The goal for this assignment was to solve a a simple problem, finding the maximum number in a list, using concurrent thread execution. A barrier implemented using only binary semaphores and primitive data types must be used to ensure synchronization between the threads. It is acceptable to use the POSIX semaphore API for counting semaphores, provided they are only utilized as binary semaphores.

2. Approach

I chose to implement this assignment in C++ in order to take advantage of standard libraries. I also made several design decisions in order to make the code that handled array comparisons and updates simpler.

I first developed a barrier class based on Professor Grimshaw's barrier in class to ensure synchronous execution of each comparison thread. The barrier class prevents all threads from proceeding until k threads have called wait on the barrier (where k is the size of the barrier). I used

three POSIX counting semaphores as binary semaphores to control each thread accessing the barrier. The first of these, mutex, ensures that no two threads can modify the value of the barrier simultaneously. This prevents a race condition with unpredictable results. The second semaphore, wait queue, causes all threads that call on the barrier to wait until the barrier value reaches zero. Each successive call to wait on the barrier decrements the barrier value and calls wait on wait queue. When the barrier value hits zero, all threads have now finished their work, and the wait queue will be signaled until it is empty. The third semaphore, throttle, prevents threads from signaling the wait queue out of order. Without this, the wait queue could receive multiple signals in a row before receiving the corresponding waits and would squash a signal (causing a thread to hang when the barrier is released).

I also designed a main function to handle user input and thread initialization, and two helper functions to do the actual comparisons. I stored the users input in a global vector to take advantage of dynamic sizing. My first helper function, log2(int n), simply calculates the log base two value of a given integer using bit shifts. This is used to determine the number of rounds required to find the max value by calling log2(vector.size()). The number of threads required is the vector size / 2 because in the first round, n/2 threads compare n/2 pairs of adjacent values. After initializing the various thread parameters, the main function spins up each thread and passes it the thread number via a struct. I chose to use an argument struct to keep the implementation flexible in case another parameter was needed. This turned out to be unnecessary as I could have just passed in an int pointer.

The helper function, getMax(void *ptr), does all of the actual comparisons. Each thread keeps track of the current round and persists throughout the entire program execution using a while loop. During each round, a given thread will compare the values at index (threadNum * 2^roundNum) and index (threadNum * 2^roundNum) + (2^(roundNum-1)). The result is then put in the (threadNum * 2^roundNum) position in the array. This method allows the same array to be used for all comparisons without needing to allocate more space. Each thread will know exactly where in the array to look for its values for each round.

After each round, the number of required threads is halved, with the upper half of the thread numbers being done their work permanently. These threads would be trying to compare indexes past the end of the array and instead just wait for remaining rounds. All threads call wait on the barrier regardless, once they finish whatever comparisons were needed for that round. This allows all threads to proceed synchronously and exit once the last round is reached.

The other advantage of doing the comparisons in place is that we know the final result will be in index zero of the array. Thread zero will always be comparing index zero to another power of two index in the array and storing the result back in index zero due to the formula above. In the final round, thread zero will be the last one comparing indexes within the bounds of the array and will place the result of the final comparison in index zero. The main function can then just call join on thread zero and print the result from index zero.

3. Problems Encountered

Most of the problems I had in this assignment came from debugging multi threaded issues or user input in C++. I have never done input in C++ before and had an enormous amount of difficulty getting input to terminate on blank lines. I found it was easier to abandon cin and used getLine to properly handle blank lines.

There were also several issues where my barrier did not behave as expected. The program would either not terminate, or terminate too early. Unfortunately, when the issue could be in many different threads there is no easy way to isolate the problem. I found that print statements were helpful to make sure everything was executing in the correct order. I could make no assumptions about the order of instructions within a round, but I was able to see that all parts of a round were completed before moving on to the next round. It turned out that I was not terminating properly on the last round. I learned that it is extremely important to test multi-threaded code incrementally, and it is much more difficult to debug when the critical section becomes complex.

I also had issues designing a concurrent solution. I was attempting to solve the entire problem at once without properly planning my strategy. Once I broke down the work that a single thread needed to do and wrote that, initializing and joining the threads became much easier. It is extremely important to plan ahead when concurrent programming because of the various synchronization issues bound to arise during testing.

4. Testing

As mentioned above, I made sure to test my barrier before worrying about the actual array comparisons. I started with a simple helper function that only looped through the rounds and calculated indexes for that round. Inside each round, I would print out the thread number, round number and comparison indexes. I could not assume anything about the order of execution within a single round, but it allowed me to verify that each round was completing before the next one started.

I next tested the actual comparisons using varying locations for the max in the list and different input sizes. I wanted to make sure that each one of my threads was working and exiting correctly, so I placed the max value in each location in a 16 number list. I also tried running my program with input list sizes as high as 16k. Each one gave the expected result. I also tested each case multiple times to be sure that the correct result was consistent and not a coincidence of execution order.

5. Conclusions

Overall, I gained a much better understanding of synchronization and the subtle race conditions that come with it. You can never assume anything about the order of functionality in your program and have to protect each statement with the correct locks. This requires a lot of advance planning and careful consideration of possible flaws.

I pledge that I have not given or received aid on this assignment or report.

```
/*
 * Gabriel  Groover (gtg3vv)
 * HW3 - Synchronization
 * Due: 3/1/2018
 * barrier.cpp
 */

#include "barrier.h"

//Barrier Constructor
Barrier::Barrier(int val, int i)
{
    init = i;
    value = val;
    sem_init(&mutex,0,1);
    sem_init(&waitQueue,0,0);
    sem_init(&throttle,0,0);
}

//Barier Wait
void Barrier::wait() {
    sem_wait(&mutex);
    value--;

    //If not all threads are waiting
    if (value != 0) {
        sem_post(&mutex);
        sem_wait(&waitQueue);
        sem_post(&throttle);
    }
    //Signal all threads to proceed
    else {
        for (int i = 0; i < init - 1; i++)
        {
            sem_post(&waitQueue);
            sem_wait(&throttle);
        }
        value = init;
        sem_post(&mutex);
    }
}
```

```
/*
 * Gabriel  Groover (gtg3vv)
 * HW3 - Synchronization
 * Due: 3/1/2018
 * max.cpp
 */


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

using namespace std;

//Log base 2 function using  bitshifts
int log2(int n)
{
   int val = 0;
   while (n >>= 1)
      val++;
   return val;

}

//Helper function for single thread
void* getMax(void* a){
   argStruct* args = (argStruct*) a;
   int currentRound = 1;
   bool done = false;

   //Loop until thread exits
   while (true)
   {
     if (!done)
     {
          //Locate indexes based on thread id and round
          int idx1 = args->tid * pow(2, currentRound);
          int idx2 = idx1 + pow(2, currentRound-1);

          //cout << "round " << currentRound << " " << idx1 << " " <<idx2 << endl;
          //Exit thread if work is done
          if (currentRound > numRounds)
             pthread_exit(0);

          //If thread should still be doing comparisons, store max in array
          if (idx1 < numList.size() && idx2 < numList.size())
```

```
            {
              numList[idx1] = max(numList[idx1], numList[idx2]);
              //cout << "adding indexes " << idx1 << " " << idx2 << endl;
            } else done = true;
      }

        //Wait for all threads to complete
        b->wait();
        currentRound++;
    }
}

//Main function to set parameters and  spool up threads
int main(){

    //Read user input until empty line
    string num;
    while (getline(cin, num))
    {
      if (num.empty())
            break;
      numList.push_back(atoi(num.c_str()));
    }

    //Set round and thread counts
    numRounds = log2(int(numList.size()));
    int numThreads = int(numList.size()) / 2;
    //cout << numRounds << " rounds" << endl;
    //cout << numThreads << " threads" << endl;

    //Initialize thread parameters
    argStruct args[numThreads];
    pthread_t tids[numThreads];
    pthread_attr_t attr;
    pthread_attr_init(&attr);
    b = new Barrier(numThreads, numThreads);

    //Start the threads and pass in thread number
    for (int i = 0; i < numThreads; i++)
    {
       args[i].tid = i;
       pthread_create(&tids[i],&attr,getMax,&args[i]);
    }

    //Join first thread and retrieve max from index 0
    pthread_join(tids[0],NULL);
    cout << numList[0] << endl;

}
```

```
/*
 * Gabriel  Groover (gtg3vv)
 * HW3 - Synchronization
 * Due: 3/1/2018
 * max.h
 */
#ifndef MAX_H
#define MAX_H

#include <vector>
#include "barrier.h"

//Data  Structures
typedef struct argStruct {
    int tid;
} argStruct;


//Global Variables
std::vector<int> numList;
int numRounds;
Barrier *b; //Global barrier  instance

//Methods
int log2(int n); //Computes log base 2 of n
void* getMax(void* a); //Helper function for each thread

#endif
```

```
/*
 * Gabriel  Groover (gtg3vv)
 * HW3 - Synchronization
 * Due: 3/1/2018
 * barrier.h
 */

#ifndef BARRIER_H
#define BARRIER_H

#include <semaphore.h>

//Barrier Class as outlined in lecture on 3/22
class Barrier {
   private:
      int value; //semaphore value
      sem_t mutex;
      sem_t waitQueue;
      sem_t throttle;
      int init; //number of threads initially

   public:
      Barrier(int val, int i);
      void wait();
};

#endif
```