Dennis Huang

dlh4fx

CS 4414-001

<u>Operating Systems Homework 2 Write Up</u>

Problem Description

I was able to complete the assignment and get my code working. The objective of this assignment was to implement a binary, parallel reduction using threads. When the program is executed, the user will be able to input *N* integers (*N is a power-of-two)*, and the program will return the max of the input. The assignment introduced concepts of multi-threading and synchronization, which was achieved by implementing a barrier made of binary semaphores.

Approach

The first step was to read in user input. Based on the homework description, a user will indicate the end input by inputting an empty line after the last integer. An infinite while loop was used to do this, and the only way to break out of it was to input a newline character. Initially, the program read in all user input as strings, and a char array was created to read in standard input. If the input was not just a newline character, the string was converted into an integer and placed into the array.

The next step was to determine the number of threads needed to find the max integer. The number was found by simpling dividing the number of integers total by 2. To attach values to a thread a struct was created. The struct was given several fields, including a p_thread id, max integer value, and some integers for indices that are used in the max finding algorithm. An array of these structs was then created, with its length being the number of threads needed to be created. To handle dynamically creating threads, the code that created threads was done in a for loop. The for loop attached initial values to the struct, and then pthread_create was called. Each thread went to a function called "maximum" to start the next algorithm.

After casting the parameter to a struct, the first step of the maximum function was to compare the numbers, as shown in the diagram in the homework sheet. Each thread struct was given an index value. The first thread was given an index value of 0, and each subsequent struct's index was incremented by 2. The purpose of this was so each thread could compare integers in the array of user input. The two integers would be the integer at the index of the array, and then the integer at the index + 1 of the array. Whichever one was greater was then assigned to the integer “max” of the struct. This integer value was then stored in a different array that will be used in the next step of the algorithm. The next step also introduced the use of the barrier.

The purpose of the barrier was for a thread to wait for all the other threads to reach the barrier before continuing in the code. There were a total of three binary semaphores used for the barrier: one as a mutex, and two as turnstiles. These semaphores were initialized in main before the threads were created. Three semaphores were used to be able to reuse the barrier in between rounds of the threads comparing integer values. The barrier would be called twice: once before the while loop (which is now) and once inside the while lop. The barrier was called here, because I wanted all the threads to have written their current max integer value into the array before comparing them any further. To make the barrier functional, the mutex was told to wait. An integer called “semcount” was then incremented. Once semcount was equivalent to the number of threads, the second turnstile was locked, and the first was unlocked. Next, the mutex and the first turnstile were signaled. The second time the barrier was called followed the same format, except the first turnstile was locked and the second one gave the signal. This ensured that the threads were synchronized after each round before continuing to the next.

Now that all of the threads' max integers were stored in an array, the last part of the algorithm is to compare the integers in rounds. This consisted of rounds, and the number of rounds was found by dividing the number of threads by 2. Like before, each thread was responsible for comparing two integers. Each thread was given an index value, and it compared the integer at that index with the integer in the next index. After finding the greater of the two, the thread would then write the integer to the index of the array that was half its assigned index. In other words, if a thread's assigned index was 4, it would write its max integer to index of 2. Once done, the thread will wait at the barrier for the other threads, and the number of rounds was decremented. The array was used to store the intermediate results in between rounds. This continued until all of the rounds were finished. At the end, this algorithm would make sure that the absolute max was in the 0 index of the array. Back in main, the threads were joined to make sure the threads were finished, and the absolute max was printed to stdout before finishing the program.

Results

The program runs very well. The program executes quickly and does not keep the user waiting. There were problems when developing the program where after typing in a newline character, the program wouldn't print out the max integer at the end. This was due to incorrectly joining the threads, which was fixed for the final build.

The program correctly determines the max out of the integers input by the user. I have tested it multiple times using input of different lengths with random integers. Each time, the program has output the correct maximum of the set. While testing, I also had a bunch of print statements that marked which thread was currently executing and if the thread was at a barrier waiting. This helped me track how my program worked, and if there was a problem, it helped me find the source of error to fix it.

Analysis

One of the biggest problems encountered when coding was race conditions. Initially, I had the int that contained the number of rounds (a global variable) decremented outside of the mutex, which caused the program to output a different max each time I ran it. Since the variable was shared between all threads, it must be stored inside the mutex. Another race condition was found when the barrier was called in the while loop. At first, I had it so that the barrier would continually decrement semcount. This led to the barrier not being able to get called the next time through, which led to threads overwriting the max value at their own leisure. This was fixed by adding an integer value to determine whether semcount was incremented or decremented.

Conclusions

This assignment helped me learn more about threads and how they work. The program also taught me how to handle multi-threading and making sure that all the threads are synchronized to produce the correct results. I also learned how to implement barriers using semaphores, and how to make the barrier reusable in between rounds.

Pledge

I pledge that I have neither given nor received any help on this assignment.