CS 4414: Operating Systems

Machine Problem 2: Synchronization

Michael Eller

29 September 2016

The project was successfully completed with a thread barrier implementation using only binary semaphores.

Introduction

The goal of this project was to implement a parallel, binary reduction program in order to find the maximum of a given list of numbers. In this program, pairs of numbers are compared in parallel and the maximum moves on to the next 'round.' Each round needs half the threads of the previous round, so the actual reduction comparisons eventually filter down to only thread 0. In this report, I will explain my implementation of both the binary reduction algorithm with parallel threads and a synchronization barrier along with my analysis of its precision.

Implementation

Maximum-Finding Binary Reduction

The program takes in a list of numbers (assumed to be a power of 2) using *fgets*() to fill a char* buffer and then uses *atoi*() to convert that buffer into integers. In order to prevent any potential race conditions, no global variables are used. Instead each thread has access to a struct that contains a few integers for doing comparisons and a pointer to a $1 \times (log_2(N)+1)$ dimensional vector of vectors. This corresponds to a vector for each round of the binary reduction plus an additional vector that holds the initial list of N numbers.

As specified in the project requirements, $\frac{N}{2}$ threads are created once and reused each round. *The main thread does not participate in comparisons.* The *find_maximum*() function, which is where the threads execute, consists of one for loop. The for loop iterates once for each round, and the threads gather their values from the vector of vectors indexed at the current round. The higher of the two numbers is then added to the next round's vector, indexed at the thread's ID. The threads then all wait at the barrier before continuing to the next round. The final answer is stored in the last vector at position 0.

Barrier

I implemented the barrier as a class with a single method: *wait*(). The barrier also contains two integers (counter for threads waiting and a max number of waiting threads) and three binary semaphores. One semaphore is used as a mutex to lock out all other threads when manipulating the counter variable. The second semaphore is used to block all the threads that need to wait. The third and final semaphore is used to ensure that for every time the waiting semaphore is posted a thread is actually released. This ensures that the binary semaphores are not abused. All threads that call *wait*() increment the counter and wait until all other threads have called *wait*(), which means the counter variable equals the max variable. The threads are then released and the counter is set back to 0.

Problems Encountered

One of the requirements of the project was that the threads created must be reused each round. Simply creating new threads for each round would have made this project much easier. In order to reuse the threads for each round, a vector of vectors was used to store the intermediate values. Initially I used two temporary vectors of ints and alternated which one I cleared and which I filled with values based on the current round. Unfortunately the program tended to segfault often. The vector of vectors implementation worked much better. However, the program still segfaulted, but that was because of the *push_back*() method. Either because of race conditions or because of uninitialized memory, trying to use *push_back*() resulted in a segfault occasionally. To alleviate this, I initialized all the vectors to the correct size and filled them with zeros. I could then access the vectors using the '[]' operator, eliminating the possibility of manipulating the same element from two different threads. Other than that, the project was fairly straightforward.

Testing and Analysis

The problem was first solved with a pthread barrier to ensure that the barrier was not the issue in early stages of development. In order to test the complete program (including the barrier implementation), I wrote a bash script that collects all the test files from a folder and runs each 100 times. I ran each test case 100 times to ensure that unpredictable segfaults had been eliminated. The files are included below.

testing_script.sh

```
#!/bin/bash

# Michael Eller mbe9a
# Machine Problem 2
# 29 September 2016
# testing_script.sh

for entry in tests/*;
do
        for i in `seq 1 100`;
        do
                ./max < "$entry"
        done
done
```

test_input_1

```
47358
-43
494
345
-453
-9
0
9
```

test_input_2

```
1
2
3
4
5
6
7
8
```

test_input_3

```
1001
-321
45
843
669
31
99
100
-7
-1
-4000
243
55
703
21
2
```

test_input_4

```
-1
-100
40
41
43
607
34
77
-21
88
4040
-4040
505
333
221
246
90
90
91
0
-400
80
55
32
56
-78
143
30
9
4
51
-67
```

The results from the test cases 1-4 above were 47358, 8, 1001, 4040 (100x) respectively. No segfaults occurred, so this verifies that the barrier implementation and thread execution is correct.

Conclusion

This implementation of a parallel binary reduction program successfully fulfilled all requirements and design specifications. POSIX semaphores were used as binary semaphores to implement a synchronization barrier. All test cases were successful, and while the project was conceptually difficult, the source code is fairly concise.

Code

makefile

```
# Michael Eller mbe9a
# Machine Problem 2
# 29 September 2016
# makefile

OBJS = maximum_finder.o barrier.o
CC = g++
DEBUG = -g
CFLAGS = -Wall -c $(DEBUG)
LFLAGS = -Wall $(DEBUG)

max: $(OBJS)
	$(CC) $(LFLAGS) $(OBJS) -lpthread -o max

shell.o: maximum_finder.h maximum_finder.cpp
	$(CC) $(CFLAGS) maximum_finder.cpp

barrier.o: barrier.h barrier.cpp
	$(CC) $(CFLAGS) barrier.cpp

clean:
	\rm *.o *~ max

tar:
	tar cfv mbe9a.tar max maximum_finder.h maximum_finder.cpp barrier.h barrier.cpp makefile mbe9a.pdf
```

barrier.h

```
// Michael Eller mbe9a
// OS Machine Problem 2
// 29 Spetember 2016
// barrier.h

#ifndef BARRIER_H
#define BARRIER_H

#include <semaphore.h>
using namespace std;

class Barrier {
public:
	// 1 mutex and 2 binary semaphores will be used
	sem_t mutex, waiter, handshake;

	// counter to hold the number of waiting threads, int for maximum number of threads to wait
	int counter, max;

	// default constructor
	Barrier();

	// constructor
	Barrier(int number_of_threads);

	// only method implemented, called to wait just like the pthread_barrier
	void wait();
};

#endif
```

barrier.cpp

```
// Michael Eller mbe9a
// OS Machine Problem 2
// 29 Spetember 2016
// barrier.cpp

#include "barrier.h"

// default constructor
Barrier::Barrier()
{
        // this don't do nothin'
}

// constructor
Barrier::Barrier(int number_of_threads)
{
        // init variables
        counter = 0;
        max = number_of_threads;
        // mutex for locking out all other threads when in wait()
        sem_init(&mutex, 0, 1);
        // waiter binary semaphore to hold all waiting threads
        sem_init(&waiter, 0, 0);
        // binary semaphore to ensure that only one thread is released at a time --
        // to ensure that binary semaphore is not violated
        sem_init(&handshake, 0, 0);
}

// method used to wait on the barrier --
// threads will be released when the number of threads that
// have called wait() == max
void Barrier::wait()
{
        // block other threads
        sem_wait(&mutex);

        // increment counter
        counter++;

        // if all are waiting
        if (counter == max)
        {
                // reset counter
                counter = 0;

                // release the 'barrier'
                for (int i = 0; i < max - 1; i++) {
                        // release a single thread
                        sem_post(&waiter);

                        // wait for the thread to be released -- essential so that the binary semaphore is not violated
                        sem_wait(&handshake);
                }
                // unlock mutex
                sem_post(&mutex);
                return;
        }
        // unlock mutex and wait on waiter semaphore
        sem_post(&mutex);
        sem_wait(&waiter);

        // allow barrier to continue releasing and exit
        sem_post(&handshake);
        return;
}
```

maximum_finder.h

```
// Michael Eller mbe9a
// OS Machine Problem 2
// 29 Spetember 2016
// maximum_finder.h

#ifndef MAXIMUM_FINDER_H
#define MAXIMUM_FINDER_H

#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <pthread.h>
#include <vector>
#include "barrier.h"
using namespace std;

// struct used to hold information for individual threads
struct pthread_params
{
        // values hold the numbers to compare for the round
        int value_1;
        int value_2;

        // max(value_1, value_2) goes in result
        int result;

        // thread id, used for indexing
        int tid;

        // total number of rounds to perform
        int rounds;

        // 2D vector pointer for winners of each round
        vector<vector<int> >* results;

        // barrier pointer, sychronization object
        Barrier* barrier;
};

// thread function to calculate the higher of the
// two current values in the thread params struct
void* find_maximum (void* input);

#endif
```

maximum_finder.cpp

```
// Michael Eller mbe9a
// OS Machine Problem 2
// 29 Spetember 2016
// maximum_finder.cpp

#include "maximum_finder.h"

// thread function to calculate the maximum of the input numbers
void* find_maximum (void* input)
{
	// cast input into pthread_params
	struct pthread_params* parameters;
	parameters = (pthread_params*) input;

	// main loop, runs once per round
	for (int i = 0; i < parameters->rounds; i++)
	{
		// make sure only 'valid' threads continue comparing and adding to the next round
		if (parameters->tid < (*parameters->results)[i].size()/2)
		{
			// get values from current vector
			parameters->value_1 = (*parameters->results)[i][2*parameters->tid];
			parameters->value_2 = (*parameters->results)[i][2*parameters->tid + 1];

			// determine which value is larger
			if (parameters->value_1 > parameters->value_2)
			{
				parameters->result = parameters->value_1;
			}
			else
			{
				parameters->result = parameters->value_2;
			}

			// add result to next vector
			(*parameters->results)[i + 1][parameters->tid] = parameters->result;
		}

		// wait on all threads before the next round
		parameters->barrier->wait();
	}

	// kill thread when done
	pthread_exit(NULL);
}

int main (int argc, char const *argv[])
{
	// list to hold the numbers
	vector<int> numbers;

	// read input, stop if blank line is received
	char *buffer = (char*)malloc(sizeof(char)*128);
	while(fgets(buffer, 128, stdin)[0] != '\n')
	{
		// use atoi() to cast to int and push to the vector list
		numbers.push_back(atoi(buffer));
	}

	// get number of threads to create
	unsigned int number_of_threads = numbers.size()/2;

	// calculate number of rounds based on the number of inputs -- assuming powers of 2
	int rounds = log2(numbers.size());

	// make nested vector for the results of each round
```

```
        vector<vector<int> > results;
        results.push_back(numbers);
        for (int i = 1; i < rounds + 1; i++)
        {
                // must initialize vectors with size to prevent segfaults
                vector<int> temp (numbers.size()/(2*i), 0);
                results.push_back(temp);
        }

        // create the barrier for synchronization
        Barrier barrier = Barrier(number_of_threads);

        // create paramters list
        pthread_params pthread_params_array[number_of_threads];

        // create threads list
        pthread_t pthread_array[number_of_threads];

        // create pthread attr
        pthread_attr_t thread_attr;
        pthread_attr_init(&thread_attr);

        // create the threads and thread parameter structs
        for (int i = 0; i < number_of_threads; i++)
        {
                // value_1 and value_2 are set in find_maximum()
                pthread_params_array[i].rounds = rounds;
                pthread_params_array[i].tid = i;
                pthread_params_array[i].results = &results;
                pthread_params_array[i].barrier = &barrier;
                pthread_create(&pthread_array[i], &thread_attr, find_maximum, (void*) &pthread_params_array[i]);
        }

        // join on all threads
        for (int i = 0; i < number_of_threads; i++)
        {
                pthread_join(pthread_array[i], NULL);
        }

        // final result is filtered down to the first thread's result
        // display final result to stdin and exit
        printf("%d\n", pthread_params_array[0].result);

        return 0;
}
```