CS 4414: Operating Systems – Spring 2018
Machine Problem 3: Threading and Synchronization
Cassie Willis

Completion Level:

The code works to the full extent that it was tested. To the programmer's knowledge, the code is fully functional.

Problem Description:

The goal of this assignment was to develop a program that uses synchronization between multiple threads to find the maximum value from a given list of inputs. The program uses threads to compare two integer inputs, with a separate thread handling each comparison. The threads run synchronously and are reused between 'rounds' where the rounds compare the maximum values of the previous round until only one overall maximum value remains. This value is sent to the command line output stream. Barriers are implemented to synchronize the threads using binary semaphores.

Approach:

The approach to developing this multithreaded, synchronized maximum value finder was to break the problem down into its individual components. Each thread was to perform the same task of comparing two numbers in a list of integers given as input to the program. The inputs were housed in an array, which was overwritten with the maximum values from each thread as each round progressed since the program did not require the storage of the previous non-maximum values. The design decision then came down to deciding to hold the maximum values either in the array in order so that the next round of threads could call the values to compare sequentially, or to hold the maximum values in the location of the first value of the current thread's comparison. An example of each choice is seen below.

Round 1	V0	V1	V2	V3	V4	V5	V6	V7

Round 2	V01_max	V23_max	V45_max	V67_max	V4	V5	V6	V7

Figure 1: Option 1, Maximum values held in order

Round 1	V0	V1	V2	V3	V4	V5	V6	V7

Round 2	V01_max	V1	V23_max	V3	V45_max	V5	V67_max	V7

Figure 2: Option 2, Maximum values held in first comparison value

The latter option was chosen because the author believed that developing an algorithm for determining which array locations to compare in subsequent rounds was less difficult than dealing with the risk of a value at the beginning of the array being overwritten before it could be compared as a maximum value in its own thread. This was a concern because synchronization is not truly 100 percent synchronous and threads comparing later array elements could run before the threads comparing the first array elements. The order in which the threads are run cannot be predicted. The barrier implementation comes into the equation here, aiding in synchronization by

ensuring every running thread has finished its computation and is waiting before the can be released for the next round of computations.

The author determined that the easiest implementation of this problem would be to break the solution down into nine functions, three of which are held in the barrier class implementation, and one of which was the main function. Three of the functions deal directly with threading and are called by each other. A description of each of the functions is listed below.

get_input():

This function uses fgets() and sscanf() to scan the input from the command line into a buffer and, if the input is an integer, add the integer to an inputArray until the input is not an integer. This function also calculates the number of input values given as well as the number of threads needed for the first round of calculations. These three variables are all globally allocated and therefore this function does not have a return value.

print_output(int max):

This function prints the maximum value to the standard output of the command line. It consists of only the printf() statement and does not have a return value.

find_max(int first, int second):

This function takes in two integer inputs and uses a direct comparison of the two parameters to determine which value is the greater of the two input parameters. If the values are equal, the function arbitrarily chooses the first value. The maximum of the two values is returned as an integer.

iterate(void * param):

This function is the function that actually gets called by the pthread_create() function call in main(). The parameter value passed to it is the current thread's identification value (TID). The function uses bit shifting for logarithmic calculations in determining the number of rounds that are required to find the maximum input value. The number of rounds is equal to $log_2(number\ of\ input\ values)$. Once the number of rounds is calculated, the number of threads that are running per round is determined as $numThreads/2^{roundNum}$. The algorithm, housed in max_threading() (described below) is then only called on the active threads for the current round. The inputArray is then updated to house the new maximum number from the algorithm into the proper location, given by max_threading(). The round number is increased and myBarrier.wait() is invoked on the thread. This function returns null.

max_threading(int currThread, int currRound):

This function is the primary algorithm for finding which two values to compare, finding the maximum value of that comparison, and determining where to store the new maximum value. The algorithm for determining the first value's location is

$$firstVal = (currThread \times shift)\ \%\ numberOfInputs$$

Where shift is $2^{currRound}$. The algorithm for determining the location of the second value to compare is

$$secondVal = (currThread \times shift + added)\ \%\ numberOfInputs$$

Where added is $2^{currRound-1}$. These algorithms were determined after manually writing out the values for each round for a 16-input example until a pattern was determined and an algorithm based on that pattern was developed. The algorithm takes advantage of the TIDs in the threads and the round number so that each thread can only call the elements in the array that they are assigned to call and multiple threads cannot be changing the same index values at once.

This function uses the first and second value's locations in the inputArray to compare these two values in the find_max() function. The output from this is put into the global variable newNum, and the function returns the first location index.

main():

The main function handles all of the thread creation, initialization, and joining. The function calls get_input() to initialize the array and uses pthread_t to create an array of thread identification numbers. The function also initializes myBarrier, setting all three binary semaphores to the correct value and setting the initial barrier value to the number of threads. Main creates each of the threads using pthread_create(), using initialize as the calling function and the location of that thread's id-number as the parameter to the initialize function. Pthread_join() is used to suspend each of the threads, and print_output() is called to print out the maximum value found by the threads. The rounds are handled as described above in the iterate function. Finally, main() returns a value of zero, signifying the end of execution with no errors. The main() function also acts as the main thread, and the created threads will continue to suspend until the main thread terminates.

Class CS:

This class is a counting semaphore class developed to handle the barrier, used in thread synchronization. The barrier acts as a door that is shut while all the threads in the current round are running. A thread upon completion of its function hits the door and is added to a wait queue. Once the barrier has received wait signals for every thread, this number determined by the barrier's value, the barrier opens the door and all of the threads can move onto their function for the next round.

The class was implemented for a barrier because the barrier relies on three separate binary semaphores, as described in lecture. All of these semaphores are used in the CS.wait() function of the class. The mutex binary semaphore acts as a lock on the wait() function, signifying that once a thread begins to use this function, no other thread can do so until the current thread releases it. This is done to protect the critical section of the code. The waitq binary semaphore is used to make the current thread wait if the barrier has not yet received all of the wait signals it needs. The throttle semaphore is used to ensure that only one thread can increment the waitq value at a time, ensuring that no threads become lost by never being able to access the barrier release.

The class houses three functions that handle all necessary implementation for the barrier.

CS(int val) : This function initializes a CS class object, aka a barrier, with only the value of the barrier.

inits(int s1, int s2, int s3, unsigned v1, unsigned v2, unsigned v3, int barrierVal) : This function initializes the barrier value again with a correct value, as this function is called after the number of threads needed is determined by get_input(). This function also

initializes the values needed for the binary semaphores using sem_init(), which are all zeroes except for the mutex semaphore's initialization to one.

wait() : This function holds the actual barrier.wait() implementation using the three binary semaphores as described above. The function first calls sem_wait() on the mutex semaphore and decrements the barrier value. The function then checks if the barrier is still waiting (barrier value is not equal to zero), and releases the mutex semaphore using sem_post(), adds a wait to the waitq, and releases the throttle. If the barrier is not still waiting, the barrier releases the waitq and adds a wait to the throttle for every thread, then resets the barrier value and releases the mutex.

Results:

The program is able to successfully create a barrier and multiple threads to calculate the maximum number of a group of inputs using binary reduction. The program implements synchronized multithreading, with each thread performing a max value calculation and waiting on the barrier correctly between rounds. The program can be run using the 'make' command, which creates a max.o target executable.

The program was tested thoroughly by the author both through direct command line input and through cat pipelining from an input.txt file. The author tested with six different power-of-two number of inputs with the maximum value being placed at many different locations within the input value list to ensure that the maximum value could be found regardless of its location in the array. The author also tested with a number of inputs that were not equal to a power of two. For these inputs, the program compared and correctly found the maximum number in the list up to the closest power of two without going over. For example, if seven inputs were provided, the program found the maximum number of the first four inputs.

Analysis:

As evident by the complete functionality, to the author's knowledge, of the maximum value code, this project was a success. This finding was particularly important to the author as previous machine problems have not worked properly, however, the author feels as though she has a greater understanding of synchronization, multithreading, and barriers than of the FAT file system.

Though some errors occurred in developing this program, the author also learned a great deal from this machine problem. With this assignment, the author was able to more deeply understand barriers, multithreading, and synchronization. The author also learned more about the C++ programming language, particularly in the refresher of classes and semaphore implementation. Prior to this problem, the author had implemented semaphores in Advanced Embedded Systems, however, was not able to fully understand how they work, possibly due to not being used to the format of a graduate level class. The author had not worked with C++ in two years.

The author did run into a few errors in creating this solution. The biggest setback was in attempting to create multiple arrays that held the maximum values and having the threads alternate which array they were accessing. This implementation was initially considered because the author did not realize that the initial array could be overwritten, however, once this fact was realized, a much simpler and more robust approach was taken. The author also ran into some issues with implementing the threads, as the program was initially written without threads to

ensure that the algorithm was fully functional. This error actually aided the author in a better understanding of how threads work, which the author may not have fully realized had the threads been implemented correctly in the first attempt. Once the author's barrier was implemented, one last problem was encountered where the threads would run for the first round and then become caught in an infinite loop. This issue turned out to be the mutex releasing twice when it was only supposed to be releasing once, and solving this problem forced the author to fully understand the barrier implementation given in class instead of simply copying the sudo-code into C++.

Finally, the author continued to learn about time management strategies and planning. This project was completed over a greater period of time, which caused less stress and less feeling of a time crunch. The project was also mapped out on paper before any code implementation. Each of the new functions were pseudo-coded to ensure that the author had a clear path of how to implement the functions in the C++ programming language, and the author took advantage for the first time of TA office hours, which was a significant help in completing the project.

Conclusion:

Barrier and semaphores are by no means simple to understand, however, because of this machine problem the author has a much better understanding of how they operate and how they are essential to multithreading, assuming the program has any type of critical section accessed by more than one thread. It is interesting to see how semaphores could be used in even very simple programs, however, it is also clear that they could be used for many very robust and complex implementations as well, especially in cases where critical sections become more and more dangerous if they are not kept in atomic code. Multithreading is also very interesting because speed and efficiency are always desired, and in many cases critical, particularly in real time operating systems such as many embedded applications. This is of particular interest to the author as a computer engineer, as embedded systems are the author's first choice for future career paths. It was useful to take a hands-on approach to semaphores and multithreading in order to more thoroughly understand it, as the author feels as though much was learned through this process.

Pledge:

On my honor as a student, I have neither given nor received aid on this assignment.

Cassie Willis

Code Dump:

```
/* Cassie Willis
 *
 * CS 4414 - Operating Systems
 * Spring 2018
 * 3/26/2018
```

```
 *
 * MP3 - Barrier and Synchronization Problem
 *
 * The purpose of this project is to become familiar with threads, barriers,
 * and synchronization. This project receives integer input from STDIN and
 * finds the max value from the inputs, returning that value to STDOUT.
 * Refer to the write-up for complete details.
 *
 * Compile with MAKE
 *
 */

#include <stdio.h>
#include <pthread.h>
#include <stdint.h>
#include <stdlib.h>
#include <semaphore.h>
using namespace std;

int numberOfThreads = 0; //Number of threads needed for the given round
int numberOfNumbers = 0; //How many integers are given in STDIN before an empty line
int inputArray[8000];
int howManyRunThreads = 0; //Number of active threads in a given round
int newNum;

//Function declarations
int max_threading(int currThread, int currRound);
void* iterate(void* param);
void get_input(void);
void print_output(int max);
int find_max(int first, int second);

//This class uses three binary semaphores to create a barrier_wait function
//so that the threads are synchronized based on the barrier
class CS {
        sem_t mutex;
        sem_t waitq;
        sem_t throttle;
        int bVal;
        int k;
        public:
                //Initialize the binary semaphore values and the barrier value to an existing barrier
                void inits(int s1, int s2, int s3, unsigned v1, unsigned v2, unsigned v3, int barrierVal) {
                        sem_init(&mutex, s1, v1); //0,1
                        sem_init(&waitq, s2, v2); //0,0
                        sem_init(&throttle, s3, v3); //0,0
                        bVal = barrierVal;
                        k = barrierVal;
                }
                //Initialize a barrier
                CS(int val) {bVal = val;}
```

```
		//Barrier wait function that synchronizes the threads
		void wait(void) {
			sem_wait(&mutex); //make the wait() function atomic
			bVal --;
			if (bVal != 0) { //still waiting
				sem_post(&mutex);
				sem_wait(&waitq);
				sem_post(&throttle);
			}
			else { //release
				int i;
				for(i = 0; i < k - 1; i++) {
					sem_post(&waitq);
					sem_wait(&throttle);
				}

				bVal = k;
				sem_post(&mutex);
			}
		}
};

CS myBarrier(numberOfThreads); //Initialize a barrier

//Get the input from STDIN and add values to inputArray
void get_input() {
	char buf[8000];
	while(fgets(buf, sizeof(buf), stdin) != NULL) { //scan the input until the input given is not an
							integer, put input into a temporary array
		int x = sscanf(buf, "%d", &inputArray[numberOfNumbers]);
		if (x != 1) break;
		numberOfNumbers++;
	}
	numberOfThreads = numberOfNumbers / 2;
}

//Print the max number to STDOUT
void print_output(int max) {
	printf("%d\n", max);
}

//Compare two numbers and return the maximum of the two, if equal, return first
int find_max(int first, int second) {
	if (first >= second) return first;
	else return second;
}

//Handle finding the maximum numbers by sending the correct thread number and round number
//to the max_threading function, incrementing the rounds and activating the correct threads
void* iterate(void* param) {
```

```
	int threadNum = *(int*) param; //thread ID number
	int numRounds = 0;
	int number;
	int round = 0;
	int logOf = numberOfNumbers;
	howManyRunThreads = numberOfThreads; //how many threads are running actively in the given
                                                round

	while (logOf >>= 1) numRounds++; //determine the number of rounds to perform the
                                        comparisons using bit-wise version of logarithms
	while(numRounds != 0) {

		howManyRunThreads = (numberOfThreads / (1 << round));
		if (threadNum <= howManyRunThreads) {	//run the comparison algorithm only on
                                                        the active threads
			number = max_threading(threadNum, (round+1));
			inputArray[number] = newNum;
		}
		round++;
		numRounds--;
		myBarrier.wait();
	}
	return NULL;
}

//Handle which two values from the input array are compared, based on the thread TID and the round
number
int max_threading(int currThread, int currRound) {
	int threadNum = currThread; //tid number of the currently running thread (i.e. thread 0, thread 1,
                                    thread 2)
	int roundNum = currRound;
	int shift = 1 << roundNum; //2^(round number)
	int added = 1 << (roundNum-1); //2^(round number - 1)

	int arrCpy[numberOfNumbers-1];	//make a copy of the input array for safety purposes if
                                        multiple threads in function at once for some reason
	int i;
	for(i = 0; i < numberOfNumbers; i++) {
		arrCpy[i] = inputArray[i];
	}
	int curr = (threadNum * shift) % numberOfNumbers;
	int next = ((threadNum * shift) + added) % numberOfNumbers;
	newNum = find_max(arrCpy[curr], arrCpy[next]);
	return curr;
}

int main() {
	get_input(); //get input from STDIN
	pthread_t tids[numberOfThreads]; //Initialize correct number of threads
	int t[numberOfThreads]; //Array of TIDs for each thread
	pthread_attr_t attr;
```

```
	pthread_attr_init(&attr);
	myBarrier.inits(0, 0, 0, 1, 0, 0, numberOfThreads); //Initialize barrier

	int i;
	for (i = 0; i < numberOfThreads; i++) { //create threads running iterate()
		t[i] = i;
		pthread_create(&tids[i], &attr, iterate, &t[i]);
	}
	for (i = 0; i < numberOfThreads; i++) { //Remove threads
		pthread_join(tids[i], NULL);
	}
	inputArray[0] = newNum; //Set final max value
	print_output(inputArray[0]); //Print max value
	return 0;
}
```