Writeup:
Comments:
Code:

CS 4414 Operating Systems – Fall 2017

Homework 2: Synchronization

Jonathan Colen jc8kf@virginia.edu

The assignment was completed and runs correctly.

1. Problem Description:

The objective of this assignment was to learn synchronization techniques and apply them to a basic problem. The program had to find the maximum value in a list of *N* numbers, which were given via an input file, by using *N/2* threads. In order to avoid synchronization problems, a barrier had to be written using POSIX semaphores.

2. Approach:

The program accepts a list of numbers, each on their own line, via standard in. The inputs are parsed into numbers using the *sscanf* method and the values are stored in a linked list. After all inputs have been read, the linked list is flattened into an array and the length of the array is stored as a value *N*. Each input list is assumed to have a length that is a power of two.

The maximum array value is found by running the method *find_max.* This method generates *N/2* threads and has each thread run through *log(N)* rounds of comparisons. Each thread is given an *Argstruct* which contains a pointer to the array being operated on, the length of the array *N*, the index of the thread, and a pointer to a *Barrier* structure shared by all the threads. The threads are created in a for-loop running from 0 to *N/2*, and the index stored in the *Argstruct* is taken to be twice the value of the counter variable for a given thread. For example, the thread created at *i = 1* has index 2. The goal of index is to signify which spot in the array the thread is responsible for.

For each round, the threads compare the value stored at their own *index* in the array with a value stored at *index+offset*. The higher of the two values is stored in the array spot at *index*. Here, *offset* is initialized at 1 for round 1 and doubled for each subsequent round. Some threads are inactive in later rounds. For example, in round 2, thread 0 compares index 0 with index 2. Index 2 will not be compared with index 4, so the thread responsible for index 2 will be inactive in that round. Similarly, the thread responsible for index 4 will be inactive for round 3 since thread 0 will compare index 0 and index 4. A thread can determine that it should be inactive for a given round if its index is not evenly divisible by *2^r*, where *r* is the round number which starts at 1 and goes to *log(N)*. If a thread is active, it performs the comparison and stores the higher value in its slot in the array. Whether the thread is active or inactive, at the end of each round, it doubles its offset value and has the *Barrier* call wait.

The *Barrier* is implemented as a struct containing integers *N* and *count* as well as pointers to three binary semaphores: *waitq*, *mutex*, and *throttle*. It can be passed as an argument to two functions, *bar_init(Barrier* bar, int n)* and *bar_wait(Barrier* bar)*. Upon calling *bar_init*,

the Barrier is initialized with *count* set to 0 and *N* set to the integer argument of the function. The three semaphores are initialized using *sem_init* from the POSIX semaphore library. *Waitq* and *throttle* are initialized with a value of 0 and *mutex* is initialized to 1 so that the first thread to access the barrier can access the section protected by *mutex*.

The goal of *bar_wait* is to force the first *N-1* processes calling it to block. The *N*th call of *bar_wait* will result in all blocking processes to be resumed. This is done by keeping track of the number of processes who have called it in the variable *count*. First, the function waits on *mutex*, which protects against simultaneous accesses of the variable *count*. Next, *count* is incremented. If it is equal to *N*, *waitq* is signaled and *throttle* is waited on *N-1* times. This is done in a for-loop, and the loop will not progress at each iteration until the *wait* call on *throttle* completes. Because the semaphores are being used only as binary semaphores, without calling *wait* on *throttle*, there is no way to ensure that the signaled process actually resumed. After the loop completes, *count* is reset to 0 and *mutex* is signaled using *sem_post*.

If *count* is not yet equal to *N*, then *mutex* is signaled and the process calls *wait* on *waitq*. This will cause the process to block until it is signaled by the *N*th process calling *bar_wait*. After the *wait* call, the process signals *throttle* to allow the for-loop that the *N*th process is undergoing to continue.

In each round, the threads perform their comparisons if necessary, store the maximum of the two values in the array spots designated by their *index* values, and then call *bar_wait*. The *bar_wait* call ensures that each process is operating on the same round so that the correct values are being compared. At the end of all of the rounds, the maximum value is stored in index 0 of the array, since thread 0 is always active based on the criteria established earlier. The threads then exit and the *find_max* method returns the value stored in index 0 of the array.

It is important to note that no additional memory is allocated outside of that required for the *Argstruct* for each thread and the array itself. Each thread stores the maximum of the two values it compares in a place that other threads will access in later rounds. This also ensures that the maximum is placed in position 0 and the main thread needs to do no comparisons.

3. Problems Encountered:

The implementation of the *Barrier* and multithreaded algorithm was very straightforward. The most difficult part was managing the input and ensuring that edge cases would not cause any errors. The in-class discussions of implementations of synchronization structures using binary semaphores greatly simplified the threaded aspect of this problem, which at first seemed like it would be the most troublesome.

4. Testing:

The program was tested by generated a large number of files containing randomized lists of number and piping them into the executable. The result was compared to that of a separate program which found the maximum value by iterating sequentially through the array and storing the maximum value. The number files were generated by a Python program and the test was managed by a Bash script. There have been no input files for which the multi-threaded max-finding program gives a different result than that of the sequential max-finding program.

5. Conclusion:

This assignment was successful in teaching the use of semaphores and related structures in solving classic synchronization problems. I have learned the correct implementation of barriers, counting semaphores, and events as a result of this problem and the related in-class discussions. The multithreaded max-finding program utilizing barriers works as expected and runs quickly.

6. Pledge

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

```
/*
*      Name:         Jonathan Colen
*      Email:        jc8kf@virginia.edu
*      Class:        CS 4414
*      Professor:    Andrew Grimshaw
*      Assignment:   Machine Problem 2
*
*      The purpose of this program is to demonstrate correct implementation of the Barrier class
*      using binary semaphores. The Barrier implementation is used to perform a multi-threaded
*      max-finding routine in an array of numbers input via standard in. The array should be fed
*      in as a series of numbers separated by new lines and the output will be the maximum number
*      in the array.
*
*      This program can be compiled via
*             gcc -pthread -o max maxfinder.c -lm
*      And can be run via
*             ./max
*
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
#include <semaphore.h>
#include <math.h>

/*
*      Structure for a singly linked list of floats
*      This structure was used to simplify input parsing
*/
typedef struct int_node    {
       float val;
       struct int_node * next;
} Inode;

/*
*      Structure containing the necessary variables for implementing a barrier.
*      The barrier can be passed as an argument to bar_wait. The first N-1 threads
*      to call bar_wait will block, and the Nth wait call will free all of them.
*/
typedef struct barrier     {
       int N;                      //The Nth wait call will cause all processes blocking to resume
       int count;                  //Stores the number of threads which have called wait
       sem_t * waitq;              //POSIX semaphore (used as binary) that causes processes to block as desired
       sem_t * mutex;              //Binary semaphore controlling access to count variable
       sem_t * throttle;   //Used to ensure signalling (Nth) process knows it has signaled correctly
} Barrier;
```

```
/*
*      Structure passed as an argument in pthread_create. Contains a pointer to the array
*      being operated on by all threads, the length of the array, the index that particular
*      thread is responsible for, and a pointer to the Barrier struct being used.
*/
typedef struct arg_struct  {
       float * array;      //Pointer to the array being searched
       int N;              //Stores the length of the array
       int index;          //Stores the array index the thread is responsible for comparing/storing into
       Barrier * bar;      //Pointer to the barrier
} Argstruct;

/*
*      Initializes the Barrier with all internal variables set to correct values.
*      @param n - the value to be stored in N
*/
void bar_init(Barrier * bar, int n)     {
       bar->N = n;               //Set the N value of the barrier to the integer argument
       bar->count = 0;           //At first, no processes have called bar_wait
       bar->waitq = (sem_t*)malloc(sizeof(sem_t));
       bar->mutex = (sem_t*)malloc(sizeof(sem_t));
       bar->throttle = (sem_t*)malloc(sizeof(sem_t));
       sem_init(bar->waitq, 0, 0);             //waitq is initialized to zero so the first process blocks
       sem_init(bar->mutex, 0, 1);             //mutex is initialized to one so the first process can access count
       sem_init(bar->throttle, 0, 0);   //throttle is initialized to zero so the signalling process blocks
}

/*
*      The first N-1 processes calling bar_wait will block. The Nth process will signal N-1
*      times allowing all processes to resume. This is used to ensure all threads are coordinated
*      to be running as expected as a given time (for example, to ensure all threads are in the
*      same round).
*/
void bar_wait(Barrier * bar)     {
       sem_wait(bar->mutex);            //Wait on mutex before accessing count
       (bar->count) ++;                 //Increment the number of processes that have called count
       if(bar->count == bar->N)   {                  //Signal N-1 processes since the Nth process is still running
              int i;
              for(i = 0; i < (bar->N)-1; i ++) {
                     sem_post(bar->waitq);            //Signal to allow a waiting process to resume
                     sem_wait(bar->throttle);   //Ensure the process resumed by waiting on throttle
              }
              bar->count = 0;                 //Reset count to zero
              sem_post(bar->mutex);     //Signal mutex so other processes can access it
       } else {
              sem_post(bar->mutex);            //Release lock on count
              sem_wait(bar->waitq);            //Set process to wait
```

```
                sem_post(bar->throttle);   //Signal to freeing process that it has been freed
        }
}

/*
*       Flatten a linked list into an array. Useful here since the array gives O[1] access
*       for each thread rather than O[n] access for a linked list.
*/
void ll_to_array(float * arr, Inode * ll)       {
        Inode * head = ll;
        int count = 0;
        while (head) {
                arr[count] = head->val;
                count ++;
                head = head->next;
        }

}

/*
*       Accept input until an empty line is reached. Each line contains an input number.
*/
int load_input(Inode * ll) {
        int input_length = 100;           //No number should be longer than this, right?
        char input[input_length];
        float val;                              //Stores the numerical input values
        int count = 0;                          //Keeps track of how many values have been read in
        Inode * curr = ll;                //Pointer to the current linked list node

        while(1)        {
                if(fgets(input, sizeof input, stdin) == NULL)  {       //If EOF, return
                        return count;
                }
                if(sscanf(input, "%f", &val)  < 1 || input[0] == '\n')       {
                        return count;//Return if pattern not matched or empty line is found
                }

                curr->val = val;                                                        //Store value in ll-node
                curr->next = (Inode*)malloc(sizeof(Inode));           //Allocate next ll-node
                curr = curr->next;                                                      //Move to next ll-node

                count ++;    //Increment the number of values in the linked list
        }
}

/*
*       Function called by each thread to find the maximum value in the array.
*       arg is a void pointer to an Argstruct which contains the necessary information for each thread
```

```
*/
void * find_max_thread(void * arg)      {
       Argstruct * a = (Argstruct*) arg;

       int level = log2(a->N);          //Each thread runs log2(N) levels
       int cmp_offset = 1;              //Index offset of the value to compare to, starts at 1
       int i;
       for(i = 1; i <= level; i ++)     {      //Start at level 1. Solution exists at level = level (last one)
              //In round 1, all threads are active. In round 2, threads 0, 4, 8 are active.
              //The pattern is that in round n, threads that satisfy index = k*2^n where k is an integer
              //are active.
              if ((a->index) % ((int)pow(2, i)) == 0) {      //Checks if the thread is active in this round
                     int cmp = a->index + cmp_offset;        //Determine the index that should be compared
                     if ((a->array)[a->index] < (a->array)[cmp])    //Put the bigger of the two in index
                            (a->array)[a->index] = (a->array)[cmp];
              }

              cmp_offset <<= 1;   //Double the offset for the next round
              bar_wait(a->bar);   //Barrier wait whether this thread is useful this round or not
       }

       pthread_exit((void*)a);    //Exit the thread
}

/*
*      Multi-threaded max-finding function. Generates count/2 threads operating on array.
*      After all threads have finished, the value will be stored in array[0], which will
*      be returned.
*/
float find_max(float * array, int count)      {
       Argstruct args[count / 2]; //Argument structures for each thread
       pthread_t tids[count / 2]; //IDs for each thread to be used in pthread_join
       pthread_attr_t attr;             //Attributes for the threads

       Barrier * bar = (Barrier*)malloc(sizeof(Barrier));   //Allocate barrier
       bar_init(bar, count / 2);                                        //Initialize barrier
       pthread_attr_init(&attr);                                        //Initialzie attributes

       int i;
       for(i = 0; i < count / 2; i ++)  {
              args[i].array = array;    //Set pointer to the array
              args[i].N = count;        //Set the number of values in the array
              args[i].index = 2 * i;    //Start by comparing each subsequent pair, so each thread checks 2*i
              args[i].bar = bar;        //Set pointer to the barrier
              pthread_create(&tids[i], &attr, find_max_thread, (void*)&args[i]);
       }

       for(i = 0; i < count / 2; i ++)  {
```

```
		pthread_join(tids[i], NULL);
	}

	return array[0];	//Return the maximum value which is stored in array[0]
}

/*
*	Main method. Reads and parses input, finds maximum, and prints max value to stdout.
*/
int main()	{
	int count = 0;			//Stores the number of values in the list
	Inode * input = (Inode*)malloc(sizeof(Inode)); //Linked list for input reading

	count = load_input(input); //Load input into linked list
	float * array = (float*)malloc(sizeof(float) * count);	//Allocate necessary memory for array
	ll_to_array(array, input); //Flatten linked list into array
	//Free the Inodes now
	Inode * tmp = input;
	while(input) {
		tmp = input->next;
		free(input);
		input = tmp;
	}

	//Find max and print it out
	float max = find_max(array, count);
	printf("%f\n", max);

	return 0;
}
```