Brian Team

dot4qu

## CS 4414 Operating Systems Machine Problem 2 – Threads

**Pledge:**

On my honor, I have neither given nor received help on this assignment.

**Problem Statement:**

The purpose of this problem can be taken directly from the assignment itself: “to familiarize [us] with synchronization primitives, specifically: (a) how they can be used to construct more sophisticated primitives; and (b) how multiple threads of execution can use them in order to solve problems cooperatively”. By separating the individual comparisons of two numbers into individual threads, most of the computation could be performed concurrently. A struct also needed to be devised that could successfully pass in the data necessary to perform the comparisons, but also be personalized to each thread. I ended up being successful in this endeavor with tests containing up to 215 numbers, which used 214 threads.

**Approach:**

My very first step was to implement the proper reading in of numbers from stdin. To constantly read lines from either a file or from the user, I used a while-loop calling the ‘getline’ function to read each input number into a string. Each iteration of the loop made sure that the string input was not empty, for if it was, the assignment specified we should stop reading input. A vector was used to hold the ‘atoi’ value of each number string converted to an int.

While creating the first version of my code, I opted to use the POSIX barrier class before attempting to use my own implementation. This was simply to verify that the creation of threads and the maximum-finding algorithm were working correctly for debugging purposes.

There are a few simple steps that are necessary to implement multithreaded programming. The first is to create an array of type pthread\_t with size of n/2, where n is the count of numbers input. This array is used to store the thread ID’s as they’re created. The next step is to create a single pthread\_attr object with the default values to be used for each of the threads. Finally, the threads themselves must be created with a call to ‘pthread\_create’. This function takes in 4 arguments: a place to store the created thread ID, the attributes to create the thread with, the function to run on the thread, and a pointer to the arguments for the thread.

The next overall step was to figure out what values to place within the struct to pass as an argument to each of the threads. Since the assignment did not allow the exit and recreation of threads, once ‘pthread\_create’ was called for each thread, they needed to run continuously until the maximum value was found. Therefore, a simple algorithm needed to be developed for finding the maximum in an array for multiple iterations before the struct could be created. The algorithm chosen is as follows: First, two numbers right next to each other are compared, and the larger number is placed in the lefthand spot. Next, the leftmost (larger) number needs to be compared with the spot one farther than the one it just tested, because another thread already compared those two numbers and placed the larger in its own leftmost spot. This algorithm iterates, with the starting pointer remaining at the first index and the second, comparison, pointer moving 2n spots farther each time. Here, n represents the amount of time this algorithm has repeated. A graphical depiction of this solution is shown below in Figure 1.

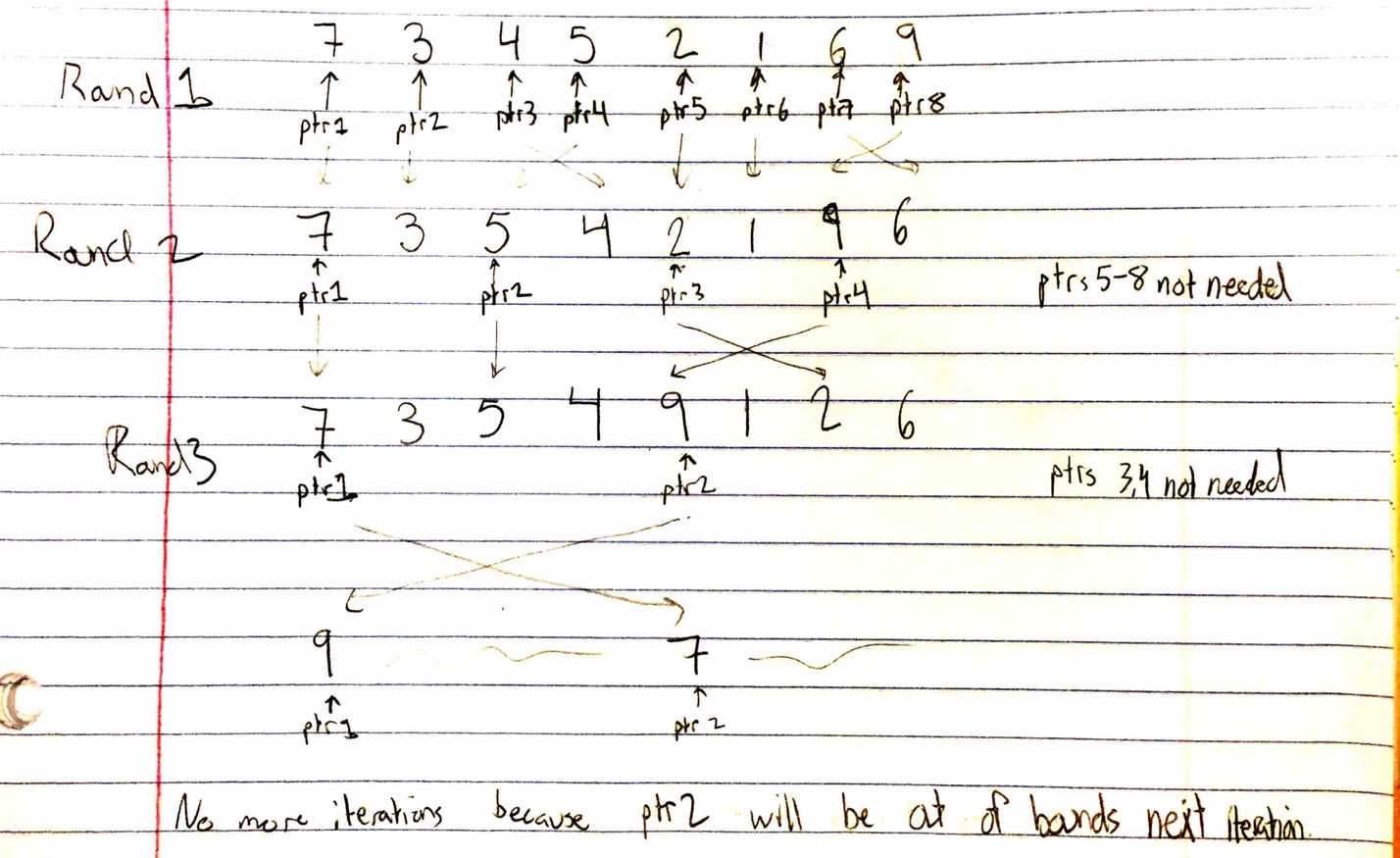


Figure 1: Maximum finding algorithm example

To run this algorithm successfully, I realized each thread would need the following arguments: a pointer to the overall numbers vector, the original number of threads created (N/2), the current thread number, and a start index. The first two parameters are self-explanatory, but the second two require more detail. The thread number is not the threads ID, but rather whatever iteration that the loop was on when creating all of the argument structs. Thus, these ranged from zero to N/2-1. The start index value was also calculated in that loop, but was incremented by two every loop iteration. This is because the starting index for the first round for every thread needed to be the lefthand element of a pair of elements. For example, the start indices for the four threads that would have been created for the above example would have been 0, 2, 4, and 6. I called the struct that held all four of these values ‘args\_struct’, and N/2 were created and placed in an array. Each individual struct was passed into its specific thread by indexing the args array with the thread creation loop’s index.

Within the ‘find\_max’ function, there was a single while-loop that checked if the ‘rounds’ variable was still less than or equal to the stored N/2 value. This variable was doubled each iteration, since only half of the threads were needed every subsequent loop. Within the while-loop, there was an if-statement to check if the current running thread was even necessary. Since there were half of the comparisons performed each round, each time the while-loop iterated, half of the remaining threads weren’t needed. By checking the thread\_num variable of each thread against the round number, half of the previous number of threads used could bypass the comparison step. Within this statement, there is a very simple if-else-statement to move the larger of the two numbers to the left hand pointer’s position.

At the end of each iteration of the while-loop inside the ‘find\_max’ function, there needed to be a way to ensure that each thread finished its comparison before moving onto the next round. The POSIX barrier class was used for this step initially, and was later implemented within my own Barrier class. This barrier object waited and blocked all threads that called it until a specific number had waited on it. Then it signaled every thread to wake them back up and continue executing.

The code for this Barrier class was not too difficult and was derived in class. Each thread that called wait first had to wait to acquire the mutex binary semaphore to ensure the counter variable was not being edited while it was in its critical section. Next, it incremented the counter variable and checked the new value against the val variable, which held the total value for the barrier to wait on. If the counter was now equal to val, it was time to release all of the waiting threads. The function then iterated through a for-loop, signaling the ‘waiter’ binary semaphore that every other thread was waiting on. If the counter variable was still less than val, the thread simply waited on the ‘waiter’ BS itself. To ensure that this program would not cause any race conditions nor overpump the ‘waiter’ binary semaphore, a handshake was implemented with another binary semaphore called ‘handshake’. A ‘wait’ on the ‘handshake’ BS after the waiter.signal() line was added along with a handshake.signal() after the waiter.wait() line. This served to ensure only one single waiting thread could be released for each call to waiter.signal() within the for-loop. Otherwise, multiple waiter.signal() calls could be executed, and the released threads could run, reenter the releasing for-loop, and cause program deadlock. In both cases, the whether the current thread had to wait or it was releasing the other threads, it always needed to release the mutex to ensure other threads had access to the barrier.

Once the number of threads exceeded a certain amount, this required them to be run as ‘detached threads’. Unfortunately, this meant calls to the POSIX function ‘join’ would not work. To force the main function to wait on the execution of all of the threads before exiting and printing the maximum, a counting semaphore was implemented. This semaphore was initialized to a count of n-1, where n was the total numbers input. Within the main function, the code to wait on the semaphore was placed directly after the for-loop which created all of the threads. Within the ‘find\_max’ code that the threads were executing, I reasoned that the zeroth thread was going to be executing n-1 times, since that would always hold the max in the farthest left index at the end. Therefore, each iteration of the ‘find\_max’ while-loop I checked the current thread’s number. If it was zero, I signaled the semaphore once. Therefore, on the final execution of ‘find\_max’ and with the greatest value in the zeroth array index, the semaphore would be signaled, its value would reach zero, and the main function would complete and exit.

**Results:**

Upon completion of my program, it would run for up to an input of 215, or 32,768, numbers. This is because an input of that size would spawn 214, or 16,384 threads. Attempting to run it with more than that would result in an immediate error upon thread creation. This is because there is a finite amount of threads that a machine can spawn, and especially a virtual machine. However, this was recognized by the professor, so the program can still be considered a success. It was also given that no error or bounds checking was needed on the input, for only valid numbers and input file length of a power of two would be tested. Therefore, no edge case testing (zero input numbers, etc.) and results are necessary.

**Analysis:**

This entire assignment was fairly simple overall in terms of the maximum-finding algorithm and the basic code setup. The complicated part was figuring out how to implement the maximum-finding algorithm so that it reused the necessary threads rather than exiting and re-creating them each time. The runtime of the main function is a simple linear runtime, for the two loops the create both the threads and their argument structs run only N/2 times. The time complexity for the overall running of the algorithm is more difficult to gauge. If it were running within a single thread, it would be of Nlog(N), since the amount of numbers to compare is halved each iteration but there are still N/2 rounds to run. However, since it is multithreaded, it will be even more efficient than that, running at just N/2. This is because each round can be considered constant time since every comparison is happening simultaneously.

There could also most-likely be some optimization in terms of space complexity throughout the code. Memory use was deemed mostly irrelevant for the sake of clarity within variables, i.e. some variables could have been reused rather than allocating memory for second, similar versions. Since the code for this assignment was so short and simple as stated before, memory and space usage was not the primary goal.

**Conclusion:**

The purpose of this assignment was not to test our coding skills too vigorously, but to make sure we understood the process and idea behind multithreaded programming. By forcing the reuse of threads rather than creating and destroying new ones each round, the homework instilled an understanding of the concepts behind thread-use rather than just a single application. The use of a struct to pass arguments to the thread function was also important, for it led us to think about what exactly our algorithm would need to execute properly, rather than just having global variables or everything located in the main scope. Overall, while difficult to debug, this assignment served as a great introduction to the advantage of using multiple threads to accomplish tasks more efficiently.

**Makefile**

# Written by Brian Team (dot4qu)

# Date: 9/28/16

# This makefile is responsible for compiling and linking HW2, the max finding process synchronization problem

CC=g++

CFLAGS=-pthread -ggdb

DEPS=max.cpp barrier.cpp max.h barrier.h

OBJS=max.o barrier.o

%.o: %.cpp

$(CC) -c -o $@ $< $(CFLAGS)

max: $(OBJS)

$(CC) -o $@ $^ $(CFLAGS)

clean:

@rm -f \*.o max

**Barrier.h**

// Written by Brian Team (dot4qu)

// Date: 9/28/16

// This is the header file for the Barrier class

#include "pthread.h"

#include "semaphore.h"

#include "stdio.h"

#ifndef BARRIER\_H

#define BARRIER\_H

class Barrier {

public:

int val;

int counter;

sem\_t \*handshake;

sem\_t \*mutex;

sem\_t \*waiter;

Barrier(int val);

void wait();

};

#endif

**Barrier.cpp**

// Written by Brian Team (dot4qu)

// Date: 9/28/16

// This .cpp file is responsible for implementing the barrier class to cause the threads to wait

#include "barrier.h"

//constructor

Barrier::Barrier(int value) {

this->val = value;

this->counter = 0;

this->mutex = new sem\_t;

this->waiter = new sem\_t;

this->handshake = new sem\_t;

sem\_init(this->mutex, 0, 1);

sem\_init(this->waiter, 0, 0);

sem\_init(this->handshake, 0, 0);

}

void Barrier::wait() {

//wait for mutex

sem\_wait(this->mutex);

//pre increment counter for # threads waiting on barrier

this->counter++;

//are we the very last thread to enter the barrier?

if (counter == val) {

//last thread into barrier, release everyone. i < val - 1 because we want to awake val - 1 other threads, we never slept

for (int i = 0; i < (val - 1); i++) {

//signal the waiter to release a waiting thread from down below

sem\_post(this->waiter);

//wait on the handshake so we know only a single thread crosses the waiter-wait to then signal our handshake

sem\_wait(this->handshake);

}

// reset counter to let threads start to pile up again

counter = 0;

//awaken mutex for others to enter again

sem\_post(this->mutex);

//were done, get back to work

return;

}

//always give back the mutex lock

sem\_post(this->mutex);

//implied else clause which means we're not the last one in, we need to wait on the waiter

sem\_wait(this->waiter);

//signal the handshake so release loop above knows a signal thread has passed the waiter and is restarted

sem\_post(this->handshake);

}

**max.h**

// Written by Brian Team (dot4qu)

// Date: 9/28/16

// This is the header file corresponding to the max.cpp file

#include <stdlib.h>

#include <stdio.h>

#include <vector>

#include <iostream>

#include <string>

#include "semaphore.h"

#include "pthread.h"

#include "math.h"

#include "barrier.h"

#ifndef MAX\_H

#define MAX\_H

//simple macro for error checking

#define CHECK\_ERR(x) if (x) {\

printf("%s: Error at line %d with value %d \n", \_\_func\_\_, \_\_LINE\_\_, x); \

return 0; \

}

//struct to hold args for passing into threads

struct thread\_args {

std::vector<int> \*input\_nums; /\* pointer to full input array \*/

int start\_idx; /\* pointer index throughout the input array to start from when comparing two numbers \*/

int half\_n; /\* holds # of threads, n/2 \*/

int thread\_num; /\* holds index of thread 0 - n/2 for checking what threads are still being used each iteration \*/

};

//thread function

int find\_max(int a, int b);

#endif

**max.cpp**

// Written by Brian Team (dot4qu)

// Date: 9/28/16

// This .cpp file is responsible for implementing the spawning and running of the max algorithm

#include "max.h"

//global semaphore declaration to use between main and thread func

sem\_t main\_sem;

//global barrier declaration to init in main and wait on in thread func

Barrier \*barrier;

/\* this function finds the max of two elements and stores it in the position held by its leftmost pointer.

After each max is found, 'round' is incremeneted to handle different indexing and offsetting as the

threads parse more and more values. Each thread waits on the barrier class after finding a max so that no

threads will get ahead of each other and grab the wrong values from the array that havent been set yet.

Once completely finished, the pthreads must call exit. Since they're detached, they can't be joined in main. \*/

void\* find\_max(void \*input) {

int round; /\* keeps track of of how far to offset comparison pointer for how many previous sets of comparisons weve done \*/

int half\_n; /\* half of total numbers input initially. serves to let us know when to stop comparing \*/

int start\_idx; /\* index for lefthand pointer to begin comparison at. thread 0 will be 0, thread 1 will be 2, etc \*/

std::vector<int> \*input\_nums; /\* pointer to original input vector whose addr was passed in args \*/

int thread\_num; /\* holds current thread number to tell if this is one that's still doing work (lower halves) or is unneeded and can skip to wait \*/

std::vector<int>::iterator start\_iter; /\*iterator that points to the more left value to be swapped\*/

std::vector<int>::iterator offset\_iter; /\*iterator that points to the more right value to be swapped\*/

//pthread\_detach(pthread\_self());

//cast input args

thread\_args \*args = (thread\_args\*) input;

//begin round as 1 and double each loop iteration

round = 1;

//assign local vars from struct

half\_n = args->half\_n;

input\_nums = args->input\_nums;

thread\_num = args->thread\_num;

start\_idx = args->start\_idx;

//while still maxes left to compute ( round < n/2 )

while (round <= half\_n) {

//for this is true for lower and lower portions of overall thread array each iteration. others skip over and just wait

if (thread\_num <= (half\_n / round - 1)) {

//take element at start\_idx and compare to round\*2 to see if we need to swap to get bigger on left

if (input\_nums->at(start\_idx) > input\_nums->at(start\_idx + round)) {

//do nothing, bigger element is already in lefthand index

} else {

//bigger element is on right, need to swap

//setting iters to beginning element and second element compared to swap

start\_iter = input\_nums->begin() + start\_idx;

offset\_iter = input\_nums->begin() + start\_idx + round;

iter\_swap(start\_iter, offset\_iter);

}

//sem is decremented by every thread still doing work (n - 1) threads total

sem\_wait(&main\_sem);

}

//double round var to ensure were indexing far enough next iter

round \*= 2;

//double start\_idx var to use earlier threads to keep moving down the line and comparing deeper results

start\_idx \*= 2;

//wait on barrier for all other threads to perform their swaps

barrier->wait();

}

//finished with all comparisons, max num is in input\_nums[0]. need to exit since we're detached, otherwise we'd float

pthread\_exit(NULL);

}

int main(int argc, char \*argv[]) {

std::vector<int> input\_numbers; /\*vector to hold all of the input values\*/

std::string input\_char; /\*string to hold char representation of input numbers\*/

int n = 0; /\*hold total number of ints input\*/

int half\_n = 0; /\*number to hold the total number of threads we need to create\*/

pthread\_t \*thread\_ids; /\*array to hold the thread id's of all n/2 threads\*/

pthread\_attr\_t default\_attrs; /\*single attrs object that we can use to initialize all of our threads. \*/

thread\_args \*args\_arr; /\*array to hold each of the args so once we call create on the threads we have the return values\*/

int i = 0; /\*multiuse iterator for loops\*/

int args\_arr\_idx = 0; /\*holds slower iterating value for populating args arr with doubly moving value through the input numbers\*/

int err = 0; /\* var used for checking return values of functions for errors\*/

int sem\_val = 0; /\*holds value still in semaphore. main checks this after firing off every thread to know when all calculations are done \*/

//keeps pulling in lines (chars) until EOF or line is empty

while ( getline(std::cin, input\_char) && !input\_char.empty()) {

input\_numbers.push\_back(atoi(input\_char.c\_str()));

n++;

}

int hi = input\_numbers.max\_size();

//need to take n/2 for since it currently equals total number of values entered

half\_n = n / 2;

//creating new barrier object holding a value of n/2 for all of the threads

barrier = new Barrier(half\_n);

//allocating space for every thread necessary

thread\_ids = new pthread\_t[half\_n];

//instantiating default attrs

CHECK\_ERR( pthread\_attr\_init(&default\_attrs) );

//CHECK\_ERR( pthread\_attr\_setdetachstate(&default\_attrs, PTHREAD\_CREATE\_DETACHED) );

pthread\_attr\_setstacksize(&default\_attrs, 65536);

CHECK\_ERR( sem\_init(&main\_sem, 0, n - 1) );

//initializing array with n/2 args

args\_arr = new thread\_args[half\_n];

//populating each arg with pointer to vector and its corresponding start location

for (i = 0, args\_arr\_idx = 0; args\_arr\_idx < half\_n; i += 2, args\_arr\_idx++) {

//initialize new args

thread\_args \*new\_args = new thread\_args;

//set values as we iterate down the input array

new\_args->input\_nums = &input\_numbers;

new\_args->start\_idx = i;

new\_args->half\_n = half\_n;

new\_args->thread\_num = args\_arr\_idx;

//adding newly created args into array

args\_arr[args\_arr\_idx] = \*new\_args;

}

//looping through and creating n/2 threads

for (i = 0; i < half\_n; i++) {

err = pthread\_create(

&thread\_ids[i],

&default\_attrs,

find\_max,

&args\_arr[i]

);

pthread\_detach(thread\_ids[i]);

CHECK\_ERR(err);

}

//loops through checking sem value until it's been 'wait'ed down to zero by the threads. Ensures main halts until all calcs are done

sem\_getvalue(&main\_sem, &sem\_val);

while (sem\_val > 0) {

sem\_getvalue(&main\_sem, &sem\_val);

}

printf("Max is %d\n", input\_numbers.at(0));

//delete/deallocate alloced mem

free(barrier);

free(thread\_ids);

free(args\_arr);

return 0;

}