Gillian Covillo & Joshua Perez 03-20-2020

Pthread Report

**Outside Main:**

The first 3 lines of the code import the libraries needed for the code to run; including the standard library, the standard input/output library (to get input from the user), and the pthread library. The next three lines (that are not blank) are used to initialize global variables that can be used throughout the code. The variables **lb** and **ub**, standing for the lower bound and upper bound respectively, are used to help split the pthreads into semi-equal parts when summing them as broken-up sections. The next part following this is a single line section which initializes the runner method which takes 1 void parameter.

**Main:**

The following section is encapsulated in the main function. The main function takes 2 arguments, the first being an object of type **int**, called **argc**, and then a pointer to an array. This (the array) in fact is a neat way to get multiple inputs from the user in the command line as opposed to having to use direct(?) **scanf** and **printf** statements to get the user input. First, the pthread attributes are initialized and give the default attributes. Then, the variable names, N and M, are assigned respectively to the first and second command line arguments that were declared in the **argv** array. The third line of the main statement initializes a ‘**tid**’, which is the id of a thread, and declares it of type **thread\_t**. It is assigned to the size of M, which again is how many threads the user wants to create. The variable **q** is set to equal **N/M**, and was initialized to **int** so that it is ~~basically~~(synonym?) the same as floor division, in order to get an equal distancing number for all but the last thread, which needs to be either of an equal distance as the previous threads or ~~what ever~~(synonym?) the remainder is. The remainder is done by doing the modulo of **N** and **M** and set to the variable **rem**.

The **for** loop is used to create as many threads as specified by the user and to set a lower ~~bound~~ and upper bound for each thread. In order to find the lower bound and upper bound, three cases were used. The first is for the initial thread, which will always have a lower bound set to 1 and therefore an ~~upbound~~(upperbound?) only **q**-1 distance away. The second is for the last thread to be created, where the upper bound is the user specified N, and the lower bound calculated by using both **q**, **ub**, and **rem**. The last case is used for ~~all other~~(“the remaining” instead?) threads (other than first and last) which also have respective formulas for finding lower bound and upper bound.

Outside the **for** loop, the first statement creates a new thread. The **pthread\_create** statement takes four arguments, the first being a pointer to the **pid**, i.e. pthread id. The id, in this instance, is the address of the **i-number** (0-M) pthread stored in the **pid** array declared earlier. The second argument is used to link the **pthread\_attr** (pthread attributes) which were declared at an earlier stage. The next argument, **runner**, is the function that each thread must go through during creation and will be explain more in the following section. The last argument is the address of the **i** variable, which is then passed to the runner function. The **pthread join** statement then joins the thread specified in the first argument with the previously terminated thread. The second argument was specified to **Null** because the global variables were used to update ~~the final sum (~~**~~holder~~**~~)~~ **holder** to get the final sum to print at the end (3 ‘to’s in one sentence). Therefore, a return value ~~did not need to be specified~~ was not needed to ~~be~~ pass~~ed~~ between the threads as the second join statement. Finally, the last print statement prints the final summation of 1 to the specified destination, having used all the pthreads to update ~~the~~ **holder** to find the final sum (to many ‘to’s again).

**Runner:**

**First two lines of runner???**

The runner function is broken into two parts, the first **if** statement checks if **i**, the pthread iterative, is on its last run (last pthread creation). If it is, then it follows the steps to calculate the sum, assign the calculation to **sum**, print ~~the~~ **sum**, and update **holder**; since finding the sum is slightly different, ~~since~~ it may not be at the same equal distance from the lower bound to the upper bound ~~as~~ compared to the other pthreads. The last **else** statement then, ~~also~~ calculates~~,~~ using a different formula, assigns it to ~~the~~ **sum**, prints it, and updates ~~the~~ **holder**.