CS 446/646 – Principles of Operating Systems

Homework 2

On Time Due date: Sunday 3/3/2024, 11:59 pm
Late Due date: Sunday 3/10/2024, 11:59 pm

Objectives: You will implement the general Threading API in C using the **`pthread`** library. You will be able to describe how Unix implements Thread creation and termination within a Process. You will compare between output times of varied Thread counts executing a Threaded-array-sum operation. You will compare threaded output times to looped output times.

General Instructions & Hints:

- All work should be your own.
- The code for this assignment must be in C. Global variables are **not** allowed and you must use function declarations (prototypes). Name files exactly as described in the documentation below. All functions should match the assignment descriptions. If instructions about parameters or return values are not given, you can use whatever parameters or return value you would like.
- All work must compile on **Ubuntu 22.04**. Use a **`Makefile`** to control the compilation of your code. The **`Makefile`** should have at least a default target that builds your application.
- To turn in, create a folder named **`PA2_Lastname_Firstname.`** Place the response pdf in it, and create two subdirectories, **`loop`** and **`thread`**. Place the makefile ad .c file for each program into its respective subdirectory. Do not include executables or the supplied txt files. Compress the main folder (**`.zip`** or **`.tar.gz)`** into a file with the same filename as the main folder, and submit on WebCampus.
- You may ***only*** use the stdio, stdlib, string, pthread, ctype, and sys/time.h libraries, and not all of them may be necessary.
- You have been provided with **`oneThousand.txt`** and **`ten.txt`**. These files are there to use as data to test your code. Please note that you should be able to calculate the sum of up to 100,000,000 values, and that the data tested for your grade might be different.

Background:

When you launch a Process that uses Threads, the first Thread is reserved for the main function (*main Thread*), and subsequently any other Threads are created via **`pthread_create`** (in Linux accomplished through an eventual *syscall* to **`clone`**. Note that in Linux, there is no *Thread Pool* with a pre-allocated number of *Threads* (but there is a -*tunable*- **`threads-max`** limit).

The *main Thread* will wait (using **`pthread_join`**) to terminate until each child thread has completed the work it was allocated. At the time the *main thread* terminates, every child Thread that was created and hasn't already terminated itself by returning -or- by [**`pthread_exit`**] creates a zombie process, and it must be rounded up by the operating system by detaching (**`pthread_detach`**) and then terminating. If **`pthread_join`** is used to force the parent to wait for each child to finish, then there is no need for **`pthread_detach`**.

The Thread API involves:
1) Calling **`pthread_create`** and passing the (pointer) to the function that the created Thread should execute.
2) Exiting by using **`pthread_exit`** after the Threaded function executes, as well as after all work is done in the *main Thread* (in **`main()`**).
3) Waiting for created Threads to finish before exiting the *main Thread* using **`pthread_join`**.

There are **3 parts** to this assignment: 1) write a non-threaded program that sums an array and calculates milliseconds taken to fully execute summation; 2) write a Threaded program that sums over an array and calculates the milliseconds taken to fully execute the summation; 3) respond to questions based on the observed behavior of the programs.

TLDR; I'm asking you to write code to read data from a file into an array, and then sum that array's data and output how long it took to sum so that you can answer questions about behavior.

Directions, PART I: Looped Summation

Name your program **`looped_sum.c`** and name the generated executable **`looped_sum.`** All code should be written in c. Write a program to calculate the sum of numbers found in a file. The filename should be provided as a command line argument (**`./looped_sum ten.txt`**).

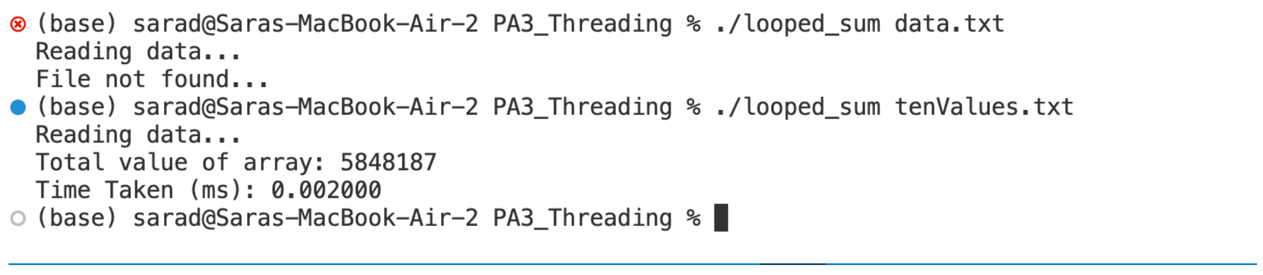
- **`main`**
  Input Params: **`int argc, char* argv[]`**
  Output: **`int`**
  Functionality: **`main`** uses supplied **`command line arguments`** as parameters to a call to **`readFile`**. Read from the file using your read method, an int array created in **`main`**, and the filename supplied on the command line by the user. The return from **`readFile`** should be stored and if the method returns -1, main should return 1 for failure. Otherwise, use **`gettimeofday`** to store the time that summation starts at. Next, call **`sumArray`** and pass it the array that the data was stored in and the number of elements in the array. Store the return from **`sumArray`**, then store the time the program finished using **`gettimeofday`** and calculate the total time of execution. Don't forget to convert it to ms!

  Finally, main should output the total value of the array and time taken to calculate it in ms.

- **`readFile`**
  Input Params: **`char[], int[]`**
  Output: **`int`**
  Functionality: This function will take the supplied filename, create a **`FILE`** input stream and open it for reading using **`fopen`**. If the file is not found, this method should print **`"File not found..."`** and -1 should be returned to indicate failure to main. Otherwise, the entire file should be parsed using **`fscanf`** and the values from the file should be placed in the passed in integer array. This method should have a counter for the number of values in the file that is incremented each time **`fscanf`** is successful. It will finally close the **`FILE`** stream and return the number of values parsed.

- **`sumArray`**
  Input Params: **`long long int[], int`**
  Output: **`long long int`**
  Functionality: Take the data array after data is read in by **`readFile`** and the number of elements in the array that was returned from **`readFile`**. Use the number of elements in the array to iterate through the array and find the array's total sum. Return the total sum.

When complete, your output should look like so (note: time taken will vary):



```
⊗ (base) sarad@Saras-MacBook-Air-2 PA3_Threading % ./looped_sum data.txt
  Reading data...
  File not found...
● (base) sarad@Saras-MacBook-Air-2 PA3_Threading % ./looped_sum tenValues.txt
  Reading data...
  Total value of array: 5848187
  Time Taken (ms): 0.002000
○ (base) sarad@Saras-MacBook-Air-2 PA3_Threading %
```

The purpose of this portion of the assignment is to show you linux performance of a looped array sum as a baseline for comparison to the threaded version in part II.

Directions, PART II: Threaded Summation

Name your program **`threaded_sum.c`** and name your executable **`threaded_sum`**. You will turn in C code for this. To build a program with **`pthread`** support, you need to provide the following flags to gcc: **`-pthread`**

You will need the following struct declaration in your code (it can be stored in **`threaded_sum.c)`** :

```
typedef struct _thread_data_t {
    const int *data; //pointer to array of data read from file (ALL)
    int startInd; //starting index of thread's slice
    int endInd; //ending index of thread's slice
    long long int *totalSum; //pointer to the total sum variable in main
    pthread_mutex_t *lock; //critical region lock
  } thread_data_t;
```

You will need to write a minimum of the following functions (you may implement additional functions as you see fit):

➢ **`main`**

Input Params: **`int argc, char* argv[]`**
Output: **`int`**
Functionality: It will parse your command line arguments (**`./threaded_sum 4 data.txt 0`**), and check that arguments have been provided (executable, number of threads to use, filename to read data from, whether to use locking or not). If 4 arguments aren't provided, **`main`** should tell the user that there aren't enough parameters, and then return -1.

Otherwise, **`main`** should first call **`readFile`** which will read in all data from the file that corresponds to the command-line-provided filename.

Then it should check whether the number of Threads requested and make sure that it is less than the amount of values read, otherwise output "Too many threads requested" and return -1.

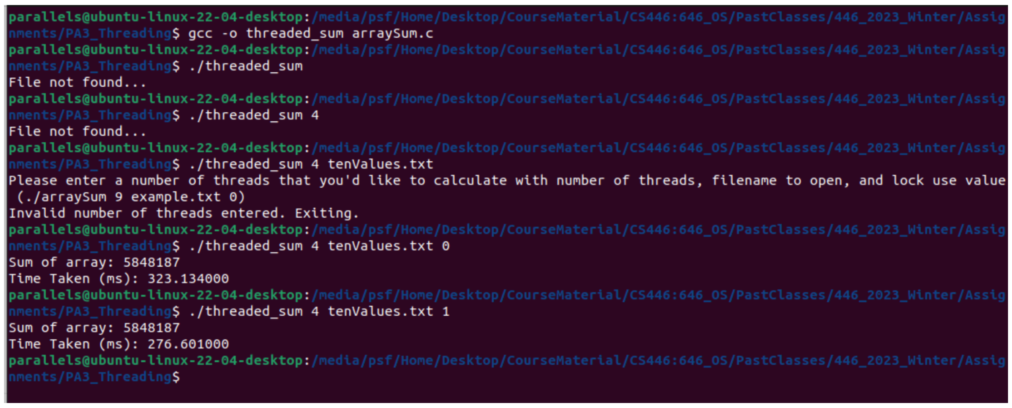
Create and initialize to 0 a **`long long int totalSum`** variable which will hold the total array sum. Store the current time using **`gettimeofday`**; this is the time that the entire Threaded implementation of summation initiates at. The program should now create and initialize a **`pthread_mutex_t`** pointer variable. If the user chooses not to use locking by setting the fourth command line argument to be 0, set the variable to be pointer to NULL. Otherwise, the mutex pointer variable will be used by any created Threads to implement required locking, so initialize it with **`pthread_mutex_init`**

At this point, the program should construct an array of **`thread_data_t`** objects, as large as the number of Threads requested by the user. Then it should loop through the array of **`thread_data_t`**, and set the pointer to the **`data`** array containing the previously read values, the **`startIndex`**, and **`endIndex`** of the slice of the array you'd like a Thread to process (i.e. to sum through), and the **`lock`** to point to the previously created **`pthread_mutex_t`**. *Note:* There are several ways to calculate the start and end indices, this is left up to your implementation.

The program should now make an array of **`pthread_t`** objects (as large as the number of Threads requested by the user). Store the time that threading begins using **`gettimeofday`**. Then run a loop that iterates over all **`pthread_t`** objects , and call **`pthread_create`** within the loop while passing it the corresponding **`pthread_t`** object in the array, the routine to invoke (**`arraySum`**), and the corresponding **`thread_data_t`** object in the array created and initialized in the previous step (the one that contains per-Thread arguments such as the start and end index that the specific Thread is responsible for).

Subsequently, the program should perform **`pthread_join`** on all the **`pthread_t`** objects in order to ensure that the *main Thread* waits until all children are all finished with their work before proceeding to the next step.

Finally, store the time that the program reached this point in another **`gettimeofday`**. **`Calculate the total execution time.`** Don't forget to convert to ms! Print out the final sum, and the total execution time and exit from the *main Thread* to terminate. The output should look something like this:



```
parallels@ubuntu-linux-22-04-desktop:/media/psf/Home/Desktop/CourseMaterial/CS446:646_OS/PastClasses/446_2023_Winter/Assig
nments/PA3_Threading$ gcc -o threaded_sum arraySum.c
parallels@ubuntu-linux-22-04-desktop:/media/psf/Home/Desktop/CourseMaterial/CS446:646_OS/PastClasses/446_2023_Winter/Assig
nments/PA3_Threading$ ./threaded_sum
File not found...
parallels@ubuntu-linux-22-04-desktop:/media/psf/Home/Desktop/CourseMaterial/CS446:646_OS/PastClasses/446_2023_Winter/Assig
nments/PA3_Threading$ ./threaded_sum 4
File not found...
parallels@ubuntu-linux-22-04-desktop:/media/psf/Home/Desktop/CourseMaterial/CS446:646_OS/PastClasses/446_2023_Winter/Assig
nments/PA3_Threading$ ./threaded_sum 4 tenValues.txt
Please enter a number of threads that you'd like to calculate with number of threads, filename to open, and lock use value
 (./arraySum 9 example.txt 0)
Invalid number of threads entered. Exiting.
parallels@ubuntu-linux-22-04-desktop:/media/psf/Home/Desktop/CourseMaterial/CS446:646_OS/PastClasses/446_2023_Winter/Assig
nments/PA3_Threading$ ./threaded_sum 4 tenValues.txt 0
Sum of array: 5848187
Time Taken (ms): 323.134000
parallels@ubuntu-linux-22-04-desktop:/media/psf/Home/Desktop/CourseMaterial/CS446:646_OS/PastClasses/446_2023_Winter/Assig
nments/PA3_Threading$ ./threaded_sum 4 tenValues.txt 1
Sum of array: 5848187
Time Taken (ms): 276.601000
parallels@ubuntu-linux-22-04-desktop:/media/psf/Home/Desktop/CourseMaterial/CS446:646_OS/PastClasses/446_2023_Winter/Assig
nments/PA3_Threading$
```

➢ **`readFile`**

Input Params: **`char[], int[]`**
Output: **`int`**
Functionality: readFile in part II is identical to readFile in part I. You may reuse the same code.

➢ **`arraySum`**

Input Params: **`void*`**
Output: **`void*`**

Functionality: This function is executed by each **thread_data_t*** object, which is what is "passed" as a parameter. Since the input type is **void*** to adhere by the pthread API, you have to typecast the input pointer into the appropriate pointer type to reinterpret the data.

Sum the **thread_data_t->data** array from **thread_data_t->startIndex** to **thread_data_t->endIndex** (only a slice of the original array), into a locally defined **long long int threadSum** variable. If the command line arguments indicated that the user doesn't want to use a lock, the **thread_data_t->lock** variable will be NULL (since you set it in main). This indicates you should NOT use the lock and unlock operations in the critical region. If the user indicated that they want to use a lock, it is important to consider how the **thread_data_t->lock** variable which is of **pthread_mutex_t** will be leveraged to ensure Thread *Safety*. Use **pthread_mutex_lock** and **pthread_mutex_unlock** to lock and unlock your *Critical Section*. To decide where to place the lock and unlock calls in this function, ask "Which part is the *Critical Section*?". An improper choice will affect the Threaded program's *Efficiency*.

Once it is done, it should update with its local sum the value stored in **thread_data_t->totalSum**.

Tips:
Generally speaking, timing will vary between program runs, and will be affected by the number of Threads the user requests, as well as by the location of your mutex lock and unlock operations. The **totalSum** should NOT be affected across changing any of the above.

Directions, PART III: answer questions below in a pdf

1. **Run looped_sum and threaded_sum a few times with whatever data file you like.**
   a. What output is the same when a program with the same data is run many times? Explain.
   b. What output changes when a program is run with the same data many times? Explain.

2. **Run looped_sum with the tenValues.txt file many times.**
   a. Do you get the same sum when you run threaded_sum with ten_values.txt and no lock?
   b. Do you get the same sum when you run threaded_sum with ten_values.txt and a lock?
   c. How does the run time of looped_sum and threaded_sum (locked AND not locked) compare?
   d. Is the total time to calculate the sum for the three cases different? Were they what you expected? Why or why not?

3. **Run looped_sum with the oneThousandValues.txt file many times. If run times do not vary consistently, try running with oneHundredMillion.txt instead.**
   a. Do you get the same sum when threaded_sum runs with oneThousandValues.txt and no lock?
   b. Do you get the same sum when threaded_sum runs with oneThousandValues.txt and a lock?
   c. How does the run time (in ms) of looped_sum and threaded_sum (locked AND not locked) compare?
   d. Is the total time to calculate the sum for the three programs different? Were they what you expected? Why or why not?

4. **Does the use of a lock in a threaded program have any impact on performance? How does the number of threads and the amount of data affect the performance of the threaded program with and without locks?**

5. **Is the lock necessary to compute consistent values every time the program is run? Why or why not? Why do you think that occurs?** You should run the program with and without a lock and with a few different data files to get the full picture.

6. **What considerations decided what was the *Critical Section*? Explain.**

Submission Directions:

To turn in, create a folder named **`PA2_Lastname_Firstname.`** Place the response pdf in it, and create two subdirectories, **`loop`** and **`thread`**. Place the makefile ad .c file for each program into its respective subdirectory. Do not include executables or the supplied txt files. Compress the main folder (**`.zip`** or **`.tar.gz)`** into a file with the same filename as the main folder, and submit on WebCampus.

Early/Late Submission: You can submit as many times as you would like between now and the due date. A project submission is "late" if any of the submitted files are time-stamped after the on time due date and time. There is no penalty for late submission; however, a 15 point bonus will be added to students who complete all 5 assignments by the on time due date.

Verify your Work:
You will be using **`gcc`** to compile your code. Use the **`-Wall`** switch to ensure that all warnings are displayed. Strive to minimize / completely remove any compiler warnings; even if you don't warnings will be a valuable indicator to start looking for sources of observed (or hidden/possible) runtime errors, which might also occur during grading.

After you upload your archive file, re-download it from WebCampus. Extract it, build it (e.g. run **`make`**) and verify that it compiles and runs on the ECC / WPEB systems.

- Code that does not compile and run will receive an automatic 0.