CS 446/646 – Principles of Operating Systems

Homework 2

Due date: Thursday, 10/3/2024, 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.

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 18.04**. 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`** and store your **`Makefile`**, your **`.c`** source code, and any text document files in it. Do not include any executables. Then compress the folder into a **`.zip`** or **`.tar.gz`** file with the same filename as the folder, and submit that on WebCampus.

Note: You have been provided with **`oneThousand.txt`** and **`oneMillion.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`**](https://linux.die.net/man/3/pthread_create) (in Linux accomplished through an eventual *System Call* to [**`clone`**](https://man7.org/linux/man-pages/man2/clone.2.html)). Note that in Linux, there is no *Thread Pool* with a pre-allocated number of *Threads* (but there is a *-tunable-* **`threads-max`** limit).

When the *Main Thread* is finished, the *Process* will (should) eventually be terminated, and every other *Thread* that was created, and hasn't already finished itself by returning -or- by [**`pthread_exit`**](https://linux.die.net/man/3/pthread_exit), (e.g. because the *Main Thread* instead of having waited on it by *Joining* it [**`pthread_join`**](https://linux.die.net/man/3/pthread_join), chose to resume its own execution by *Detaching* it [**`pthread_detach`**](https://linux.die.net/man/3/pthread_detach)), will also be terminated.

Note: An implementation-specific (i.e. not described by the Standard) detail is whether a *Detached Thread* should remain alive after the *Main Thread* is terminated and until the *Process* becomes reaped.

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`**.

You are asked to write a Threaded program that sums over an array and calculates the milliseconds taken to fully execute the summation, and then output those values.

General Directions:
Name your program **`threaded_sum.c`** . 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 may only use the following libraries, and not all of them are necessary:

```
<stdio.h>
<string.h>
<stdlib.h>
<sys/wait.h>
<sys/types.h>
<unistd.h>
<fcntl.h>
<errno.h>
<sys/stat.h>
<pthread.h>   // for pthreads
<sys/time.h>  // for gettimeofday
```

You will also need the following struct declaration in your code:

```
typedef struct _thread_data_t {
    const int *data;
    int startInd;
    int endInd;
    pthread_mutex_t *lock;
    long long int *totalSum;
} 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 data.txt 4`**), and check that 3 arguments have been provided (executable, filename to read data from, number of Threads to use). If 3 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 (argument #2).

  Then it should check whether the number of Threads requested (argument #3) 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. Create a **`timeval`** struct and store the current time using [**`gettimeofday`**] (https://linux.die.net/man/2/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`** variable that will be used by any created Threads to implement required locking, using [**`pthread_mutex_init`**] (https://linux.die.net/man/3/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), and run a loop of as many iterations, and call within it **pthread_create** 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 they are all finished with their work before proceeding to the next step.

Finally, store the time that the program reached this point in another **timeval** struct. Calculate the total execution time. Print out the final sum, and the total execution time and exit from the *main Thread* to terminate.

Note: Calculating total time of execution can be done using the difference between the start and end **timeval** values. The printed-out value should be in ms. Look here for sample code.

- **readFile**
  Input Params: **char[], int[]**
  Output: **int**
  It will take the supplied filename, create an 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. 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.

- **arraySum**
  Input Params: **void***
  Output: **void***
  It is assumed to operate on **thread_data_t*** input data (but 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).

  It should 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.

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

  Note: During this process, 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**](https://linux.die.net/man/3/pthread_mutex_lock) and [**pthread_mutex_unlock**](https://linux.die.net/man/3/pthread_mutex_unlock) to lock and

unlock your *Critical Section*, but the important thing to ask is "Which part is the *Critical Section*?". An improper choice will affect the Threaded program's *Efficiency*.

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.

Finally answer the following questions in a text document of your choice (.pdf, .doc, .docx, .odt, .txt., etc.):

Run your code with a different combination of:

i) amount of values (e.g. 1000 1,000,000 …)

ii) requested Threads for each amount (1 2 4 8 16 …)

and answer the following questions:

a) What are your observations?

b) Do you find the observed behavior reasonable? Why / Why not?

c) What kind of considerations did you make for your implementation of the *Critical Section*? Provide reasoning / justification.

[EXTRA] What do you think would happen if instead of having the Threads loop over the **`int`** array to compute a local **`arraySum`**, and finally update the **`totalSum`**, we were instead just directly adding to **`totalSum`** each time we were accessing an array element within each Thread (i.e. within the *Thread's* **`for`** loop)? Why?

Submission Directions:

When you are done, create a directory named **`PA2_Lastname_Firstname`**, place your **`Makefile`** (which has to have at least one default target that builds your **`threaded_sum`** application executable), your **`threaded_sum.c`** source code file, and the text document providing your answers to the above questions into it, and then compress it into a **`.zip`** or **`.tar.gz`** with the same filename as the folder.

Upload the archive (compressed) file 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 due date and time. Projects that are up-to 24 hours late will receive a 15% penalty, ones that are up-to 48 hours late will receive a 35% penalty, ones that are up-to 72 hours late will receive a 60% penalty, and anything turned in 72 hrs after the due date will receive a 0.

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 will receive an automatic 0.