Washington State University Vancouver

Systems Programming - CS 360

Assignment 7

Instructor:
Ben McCamish

Overall Assignment - 100 points

Use the two files provided in the handout, `assignment7.c` and `assignment7.h`.

You will be modifying `assignment7.c`, but you are not permitted to change `assignment7.h`. You will be submitting your version of `assignment7.c` which I will compile and link with a test program that will (naturally) depend on you following and not modifying the interface declared in `assignment7.h`.

The file `assignment7.c` contains a procedure which uses the classic quicksort algorithm to sort an array of strings. The nature of quicksort is that following its partition step, two recursive calls can be made to sort each partition. You may notice that these two "sub" sorts are independent of each other and could each be given its own thread of execution. Of course, the threads that might be created or employed to sort each partition would have to be synchronized at least to the extent that the user's calling thread does not return until all the "sub" sorts are completed (and any created threads are joined or destroyed).

Your task is to multi-thread the supplied `assignment7.c` program so that it employs as many threads as it is told to (via a preceding call to `setSortThreads()`).

This assignment requires you to modify `assignment7.c` to make use of as many threads as you are told (but no more), in order to sort the given array (assuming the nature of the array is such that all the threads requested can be employed). You may do this using `pthread_create()` and `pthread_join()`, although the overhead of creating a thread is significant. In order to beat me you may to use a pool of threads or consider how you partition your threads for the sorting.

You will want to build a test bed around your sort program, both to test that your code does indeed sort correctly and to time it. Consider calling the library procedure `clock()` before and after your `sortThreaded()` procedure is called to find out the performance of your code. You may also want to code up calling the library's `qsort()` as a benchmark for comparison.

Specifications and Restrictions

- (35 points) Program should work on autolab and correctly sort the input. **Note:** Since the program I provided already sorts, you won't receive these points unless you also beat the original sort. This may happen by random chance (unlikely), so unless you have implemented multi-threading, don't assume those points will be consistent.
- (35 points) Your code should be faster then the original version supplied.
- (14 points) You must beat my times. Consider the fact that creating threads all the time (what I am doing) takes a considerable about of time. A better method might be to create a pool of threads. You might also be able to be clever about how you use the threads that you created and ignore the pool entirely.
- (16 points) Must be robust, including error catching. You must catch errors and print out an appropriate error message containing the `errno` and the message produced by that error. This means you will need to use `errno.h` and `string.h`, libraries at least.
- (Note) To create and manipulate threads, use `pthread_t`, `pthread_create()`, `pthread_join()`, and `pthread_mutex_init()`.
- (Required) Clean up any memory (or threads) you allocate.
- You may also consider making any additional functions you add to the `assignment7.c` static.
- I have provided some sample files for you to use as a collection of words to sort. How you use them is up to you.
- Autolab will only accept 20 submissions, so make them count. I would suggest not using Autolab to debug your code. Only submit if you are reasonably confident that you can beat my times.
- Submit only the .c file called `assignment7.c`.

What to turn in (NOTE: ONLY THE .c FILE):

- assignment7.c (no header files)