Assignment 6: Concurrent Sorting

Due: Friday, March 17, 10pm

Starter code: See Assignment 6 on Canvas for the Github Classroom link.

Submission: This is a **pair assignment**, but you can work alone, if you so choose.

Submit the contents of your repository via Gradescope. See Deliverables below for what to submit. If you are working with a partner, do not forget to include their name with the submission.

There will be no autograder for this assignment. Read the requirements and run tests locally. We will evaluate the ability of your implementation to sort large enough inputs correctly using the correct number of threads.

In this assignment, we will turn a single-threaded naive implementation of merge sort into a multi-threaded one.

Task 1 - Concurrent Merge Sort

The starter code contains an implementation of merge sort in the file `msort.c`. The implementation is based directly on the top-down example from the above mentioned Wikipedia article. Your first task is to make sure you understand how this algorithm works. Do ask questions if you feel you are missing something. Note that the implementation works with two arrays: the input array and a "target" or "helper" array. The target array will contain the resulting sorted array. However, this implementation also modifies the original array.

The `msort` program takes a single argument which is the number of long integers to be read from standard input. Once it reads in the given number of elements and/or the standard input is closed, the array is sorted and the result is printed to standard output.

Your main task is to use the POSIX threads (pthread) library routines to turn the provided merge sort implementation into a concurrent one that improves on the performance of the single-threaded version. The Wikipedia article contains some pseudocode to get you started under *Merge sort with parallel recursion*. It shouldn't take too much effort. Implement your version in `tmsort.c`, which initially contains the same implementation as `msort.c`.

The concurrent implementation is parametrized in the number of threads it should use. This parameter is taken from the environment variable `MSORT_THREADS`, which is read in by the starter code (take a look!). Your implementation should not use more than the given number of threads, but should use exactly that number on a large enough input. E.g., when `MSORT_THREADS=10` and the input array has $1,000$ elements, the implementation should use exactly 10 threads (including the main thread). You can set environmental variables globally for the current shell using `export` (see How to Set and List Environment Variables in Linux, or you can just take a look how we do it in the Examples below.

Task 2 - Experimentation

Your second task is to perform a few experiments on at least two different machines that are capable of compiling and running pthread-based code. These include:

1. Our XOA VM
2. Khoury Login or the VDI hosts
3. Your Linux machine
4. Your macOS machine
5. Your Windows machine running WSL2 or other environment implementing pthreads.

In `experiments.md`, write a brief summary of your experiments with large enough inputs. Use the provided file as a template. Write for each machine:

1. Specs (CPU **including the number of cores**, memory, disk capacity, OS)
2. Input data size and how you created it (see Hints & Tips). Use a large enough number of elements, so that the single-threaded implementation takes at least 10 seconds to sort them.
3. The approximate number of processes running before you start each experiment (obtainable by running `ps aux | wc -l`)
4. Run experiments for different thread counts (1, 2, number of cores, double the number of cores, ...), one after another. For each thread count, run the sort at least 4 times with the same arguments/data set. Note how much time the sorting portion took (look for the line `"Sorting completed in N seconds."`).

   Keep increasing the thread count and figure out at which point adding new threads does not translate to (significantly) improved performance. How does this number relate the number of cores on the machine you run the experiments on?

Deliverables

Task 1 Implement the threaded merge sort in `tmsort.c`. Include any additional `.c` and `.h` files your implementation relies on. However, you shouldn't need to implement many additional functions and/or data structures that would imply separate compilation.

Task 2 Report on the results of your experiments and your observations in `experiments.md`.

Commit the code and the experiments to your repository. Do not include any executables, `.o` files, or other binary, temporary, or hidden files.

Once you are done, remember to submit a ZIP file with your solution to Gradescope and do not forget to include your partner.

Rubric

- 30% on good programming style
  - Basics: meaningful purpose statements; explanation of arguments and return values
  - Explicitly stated assumptions
  - Correct use of types (e.g., not assigning -1 to an unsigned)
  - Short, understandable functions (generally, < 50 lines)
  - Consistent indentation and use of whitespace
  - Minimal use of global variables (see Hints & Tips)
  - Explanatory comments for complex blocks of code
  - Clean and understandable thread management making it easy to see how many threads are being created, when they are joined, what (if any) data is shared, etc.
- 45% for a working multi-threaded merge sort
  - Does not leak memory
  - Sorts the given input correctly
  - Works on large inputs (e.g., $100,000,000$ elements)
  - On inputs that are large enough, uses exactly the number of threads given by the `MSORT_THREADS` environment variable
- 25% for running and describing experiments

Hints & Tips

- `man` is your friend. Check out `pthreads`, `pthread_create`, `pthread_join`, ...
- You can use the `nproc` command to get the number of cores available on your machine. On Linux, reading the file `/proc/cpuinfo` will give you detailed information about the processor on your system.
- The thread function passed to `pthread_create` only takes a single `void *` argument. To pass custom arguments, create a "helper" struct and pass a pointer to it. Depending on where you are waiting for the thread to join, you might need to allocate it on the heap.
- If multiple threads are reading and writing the same variable, consider using `pthread_mutex_lock` and `pthread_mutex_unlock` to ensure consistency.
- It is possible to complete this assignment *without* mutexes.
- You can use `shuf -i1-N` to generate `N` random numbers between 1 and `N`. E.g. `shuf -i1-1000 > thousand.txt` generates $1,000$ random numbers and writes them to `thousand.txt`. See `man shuf`.
- First ensure `tmsort` consistently returns the same result as `msort` on the same input. You can check that by running `diff <(./msort 1000 < thousand.txt) <(./tmsort 1000 < thousand.txt)`.
- We provided a convenient makefile target that does this: `make diff-N`, where `N` is the size of input. Example: `make diff-1000`
- Once you are sure `tmsort` is consistent with `msort`, you might want to redirect `stdout` to `/dev/null` to avoid printing the result when doing timing experiments: `./tmsort < input.txt > /dev/null`. Timings are printed to `stderr` so they will still be visible.
- Compile and test *often*.
- Use `assert` to check that your assumptions about state are valid.
- **Write your own (unit) tests.** Doing so will save you time in the long run, especially in conjunction with the debugger. In office hours, the instructors or the TAs may ask you to show how you tested code that fails.
- Unit tests can be just a bunch of functions and a main, with `assert`s to check expected results. Use our tests for queue/vector from Assignment 4 as an example.
- Follow good coding practices. Make sure your function prototypes (signatures) are correct and always provide purpose statements. Add comments where appropriate to document your thinking, although strive to write self-documenting code.
- Split code into short functions. Avoid producing "spaghetti code". A multi-branch **`if-else if-else`** or a multi-case **`switch`** should be the only reason to go beyond 40-50 lines per function. Even so, the body of each branch/case should be at most 3-5 lines long.
- It is ok to use global variables, but be judicious with their use. Give them meaningful names and explain their purpose.

Examples

Here are some examples of running our implementation of `./tmsort`.

1. Basic usage, input from terminal.

```
$ ./tmsort 5
Running with 1 thread(s). Reading input.
Enter 5 elements, separated by whitespace
5
4
3
2
1
Array read in 4.505015 seconds, beginning sort.
Sorting completed in 0.000069 seconds.
1
2
3
4
5
Array printed in 0.000024 seconds.
```

2. Using pre-generated input.

```
$ shuf -i1-10 > ten.txt
$ ./tmsort 10 < ten.txt
Running with 1 thread(s). Reading input.
Array read in 0.000034 seconds, beginning sort.
Sorting completed in 0.000006 seconds.
1
2
3
4
5
6
7
8
9
10
Array printed in 0.000040 seconds.
```

3. Varying the number of threads.

```
$ shuf -i1-100000000 > hundred-million.txt
$ MSORT_THREADS=1 ./tmsort 100000000 < hundred-million.txt > /dev/null
Running with 1 thread(s). Reading input.
Array read in 12.193864 seconds, beginning sort.
Sorting completed in 19.202604 seconds.
Array printed in 10.442048 seconds.
$ MSORT_THREADS=2 ./tmsort 100000000 < hundred-million.txt > /dev/null
Running with 2 thread(s). Reading input.
Array read in 12.093944 seconds, beginning sort.
Sorting completed in 10.288052 seconds.
Array printed in 10.403824 seconds.
```

```
$ MSORT_THREADS=4 ./tmsort 100000000 < hundred-million.txt > /dev/null
Running with 4 thread(s). Reading input.
Array read in 12.199927 seconds, beginning sort.
Sorting completed in 6.118842 seconds.
Array printed in 10.817068 seconds.
```