Project 1

CS6456 (F2016) Operating Systems
William Young
27 September 2016

1 Introduction

This paper describes the implementation of a concurrent merge-sort algorithm on a user-provided integer file based on the principle of multithreading. The main program, *mergeSort*, can sort sequences of integers with values ranging from -1000 to 1000, inclusive, in nondecreasing order. *mergeSort* utilizes the POSIX pthreads API to provide concurrency during each round of sorting. This paper will first present the structure of *mergeSort* before moving on to discuss its methodology and implementation. The paper concludes with an evaluation of performance.

2 Structure

The *mergeSort* solution tar file consists of four archived elements:

- `mergeSort.c`
- `tp.c`
- `tp.h`
- `Makefile`

`mergeSort.c` is dependent on `tp.c` and `tp.h` and includes the main method, a helper method to task the threadpool, and the multithread sort method.

`tp.c` is dependent on `tp.h` and consists of four methods:

- `threadpool* threadpool_create(int num_threads)`
    - Create threadpool object and all individual threads
    - Have all threads call `threadpool_get_task`
- `void* threadpool_get_task(void* tpv)`
    - Idle until task is available (`pthread_cond_wait`)
    - When task is available, remove from task queue and execute

- `int threadpool_add_task(threadpool* tp, void (*function)(void*), void* args)`
    - (Called from main thread)
    - Add a task to the threadpool object's task queue and wake up at least one idling task
- `int threadpool_exit(threadpool* tp)`
    - Wake up all idling tasks via `pthread_cond_broadcast`
    - Wait until all threads have terminated
    - Deallocate all threadpool object arrays
    - Deallocate threadpool object

`tp.h` consists of headers for the methods implemented in `tp.c`.

`Makefile` includes the necessary gcc calls to compile the program.

3 Methodology and Implementation

The design of the program, described below in detail, is straightforward.

1. The main thread opens and reads the contents of the target input file (received via command line) into a global array. If the number of integers provided is not a power of two, the end of the array is padded with integers of increasing value until the length of the array is a power of two. (The sorting algorithm utilized in this solution performs faster on pre-sorted values, thus the rationale behind padding with increasing values.) The result of the logarithm (base 2) on the padded array size determines the number of rounds of sorting that will be performed. Finally, the main thread calls `threadpool_create(array-size/2)` to initialize the threadpool.

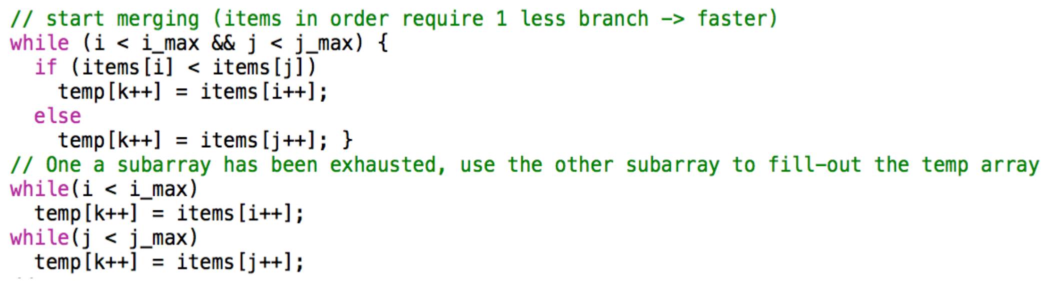
2. In `threadpool_create,` a threadpool object is created with a number of threads equal to half the unsorted-item array size, since each thread will sort two items. Newly created threads are sent to `threapool_get_task` where they either receive a task (call to perform a function) or idle until a task is available.

3. As soon as `threadpool_create` returns an initialized threadpool object, the main thread calls `mSort`, a helper function designed to task the threadpool with

the correct number of sorting operations and the correct bounds on each sort. Each level of sorting (based on the logarithm (base 2) of the array size) performs a sort on half the number of items compared to the previous level, until only one sort operation remains (the entire array). Once all operands and bounds are computed, `mSort` adds a number of tasks to the threadpool equal to the number of sort operations the must be performed. `mSort` then waits until each task is complete before starting the next level of sorting.

4. In `threadpool_add_task`, each incoming task (including function name and arguments) is added to a task queue and `pthread_cond_signal` is invoked to wake up and task at least one idling thread. In this project, all tasks are calls to `merge`, a sorting function in `mergeSort.c`.

5. In `merge`, the arguments (after unpacking and recasting to their original types) are fed to the bounds of the sort algorithm as indices. Every task in *mergeSort* operates on non-overlapping bounds of the original array, thus concurrent access is permissible. The algorithm itself is simple, and is included here for



```
// start merging (items in order require 1 less branch -> faster)
while (i < i_max && j < j_max) {
  if (items[i] < items[j])
    temp[k++] = items[i++];
  else
    temp[k++] = items[j++]; }
// One a subarray has been exhausted, use the other subarray to fill-out the temp array
while(i < i_max)
  temp[k++] = items[i++];
while(j < j_max)
  temp[k++] = items[j++];
```

reference.
(Note: *items* is the input array and *temp* is a temporary array to hold the intermediate sorting results. It, too, is non-overlapping.) Once all tasks in a round have been handled, the threads return to an idling state awaiting the next level of tasking.

6. In `main`, the main thread waits for a signal from `mSort` that all rounds of sorting have been completed. Upon receipt of the signal, it displays the sorted items and then calls `threadpool_exit`.

7. `threadpool_exit` begins by setting the *exit* flag in the threadpool object to *true* and then broadcasting to all threads. In `threadpool_get_task`, as each thread is awakened by the broadcast, it observes that the *exit* flag has been set and calls `pthread_exit` instead of attempting to pop a new task off the queue. Back in `threadpool_exit`, the method `pthread_join` is called on each [terminated] thread in the threadpool. Finally, the task queue, thread array, and threadpool object are deallocated, returning to `main`.

4 Conclusion

mergeSort was tested successfully on a myriad of inputs, including but not limited to:

- a null file
- a file with 3000 random integers
- a file with 4096 random integers
- a file with 8 random integers
- a file with 21 random integers
- a file with 4096 identical integers

In each test case, *mergeSort* produced output consisting of 10 integers per line, space-delimited. No sentinel values were included in the output.

5 References

1. Kerrisk, Michael. *The Linux Programming Interface.*
2. Goetz, Brian. “Thread pools and Work Queues.” <ibm.com/developerworks/library/j-jtp0730/>
3. “Thread pool.” *Wikipedia.* <https://en.wikipedia.org/wiki/Thread_pool>
4. Straub, J. “Threadpool Documentation.” <http://faculty.washington.edu/jstraub/isilon/isilon2/Unit7/threadpool/doc/html/index.html>