A2 - CompSys

Nikolaj Krarup, ltf688,
Simon Wissendorff Seheim, fzt420
Emil Vedel Thage, prn150

September 2023

Introduction

When programming - no matter the scale - performance is always important to keep in mind. This assignment is about using multiple threads to increase performance of our programs. These threads needs to be implemented in such a way they don't interfere with each other.
To run our programs navigate to the directory contiaining the files and type `make` to compile. Then you can run the different programs using the following commands (If you want to run them using a single thread, leave out the -n flag and the thread number).

- ./fibs -n <number of threads>
  Then you can type numbers in to queue calculations, putting the threads to work on calculating the corresponding Fibonacci number.

- ./fauxgrep -n <number of threads> <needle> <paths>
  Here you can chose to provide multiple paths to different files, or simply give the path to a directory.

- ./fhistogram -n <number of threads> <paths>

Implementation

job_queue

The job_queue is constructed to have various fields such as size, capacity, elements as well as conditions for threads, and pointers keeping track of the front and back of the queue. The array uses a looping structure to ensure full use of the allocated space.
The `job\_queue` has 4 functions, the first being initialisation of the job_queue (`job_queue_init`) which creates a pthread wait conditions and a mutex to handle multiple threads using the same queue. When the `job_queue` has been constructed three functions can be used `job_queue_destroy`, `job_queue_push` and `job_queue pop`. These functions can be used to destroy an empty `job_queue`, push elements and pop elements. Everyone of these functions can be locked via mutexes as well as be paused if conditions for the threads running them haven't been satisfied yet. For example a wait condition for destroy is placed within a while loop, checking that the queue is empty. A signal will be send to the wait condition when popping an element from the queue, waking the thread that is waiting to destroy to check the while loop condition again. This approach has been made for all of our functions, to ensure that the user dosent pop from and empty queue, and dosent push to a full queue.

fauxgrep-mt

Our implementation fauxgreb-mt borrows the structure of the single threaded version, as well as the multi threaded fibs program. We push all the file paths onto our job queue from task 1, inside the `fts_read` loop. However, since the worker threads also needs access to the search needle, we have made a new struct `search_queue`. This struct contains a pointer to our job queue, as well as a pointer to the needle. Our worker function makes use of this struct to call `fauxgrep_file` with the appropriate arguments from within the struct, using our pop function from the job queue. Since we save the job queue and search queue on the stack and not the heap, we don't need to free them, however the path we pop from the queue is freed after the worker has called the search function. When creating our threads we use the same method as the fibs program, providing the thread with the address of the worker function and a pointer to our initialised search queue.

fhistogram-mt

This one works much like our fauxgrep-mt implementation. Here we make use of the provided template as well, but instead of using the `fauxgrep_file` function, we use the `fhistogram` function from fhistogram.c. It's easier to pass arguments to `fhistogram` as it only requires a path. This means an additional struct is irrelevant since we can just push a copy of the path from FTSENT to the queue alone.
Additionally, to prevent a race condition, we also added mutexes to access of the array `global_histogram` since it's a shared resource between branches. Not having these mutexes would result in multiple histograms being printed in the terminal. These histograms would overlap each other making them unreadable since each thread prints a histogram while another is being printed.

Testing

job_queue

The testing of job_queue has been done manually throughout the implementation of `fibs`, `fauxgrep-mt` and `fhistogram`. Whenever we got the expected result from either of the mentioned functions, we were sure that job_queue worked as intended. We've also tested if the job_queue works with more pushes than there are spaces and pops when there aren't any items. It's pushes will wait for there to be space when full and pops will wait till there is an item. The destroy function will only destroy when there are no more items and will stay in a while-loop until there are no more.

The directory used for the `fauxgrep-mt` and `fhistogram-mt` tests bellow contains 247 files and 13,646,587 bytes(13 MB). Testing if `fauxgrep-mt` and `fhistogram-mt` results in the correct prints and histograms was easy since we could compare it with the result of `fauxgrep` and `fhistogram`.
The runtime of the code is however more important. We compared the runtimes of `fhistogram-mt` and `fhistogram` by calling the unix command `time` followed by the function call. The comparisons can be seen bellow:

fauxgrep-mt

Call	**Real**	**User**	**Sys**	**files/s**	**bytes/s**
./fauxgrep-mt -n 6	0m6.995s	0m0.117s	0m3.297s	35.31	1,950,906
./fauxgrep-mt -n 2	0m6.118s	0m0.083s	0m1.722s	40.37	2,230,563
./fauxgrep	0m17.335s	0m0.087s	0m1.694s	14.25	787,227

fhistogram-mt

Call	**Real**	**User**	**Sys**	**files/s**	**bytes/s**
./fhistogram-mt -n 6	0m5.189s	0m1.950s	0m2.169s	47.60	2,629,907
./fhistogram-mt -n 2	0m5.345s	0m2.091s	0m1.361s	46.21	2,553,150
./fhistogram	0m18.061s	0m2.592s	0m1.700s	13.68	755,583

As can be seen by these runtimes, the multi-threaded code is significantly faster than single-threaded, however more than one thread seems not to do much of a difference if none, we'll explain this in the next section.

Limitations and Potential Problems

A problem we ran into was the first iteration of the job_queue. The first iteration saw the the `job_queue_pop` move the entire buffer by one instead of that of the current implementation. The function removed the first element in the buffer and then preceded to move every remaining element one down until all had been rearranged, however an error would occur at the last element since it would try to move an element which did not exist. The implementation used a loop based on the size of the buffer. This caused segmentation faults in the later parts of the assignment since this way of implementing the job_queue_pop led to the function trying to move elements which weren't there.

job_queue is also quite limiting for multiple threads for `fhistogram-mt` and `fauxgrep-mt`. When testing, we realized that any amount of threads greater than 1, would make the program run on essentially the same time, leaving only a noticeable difference when using less than 1. We tried finding the problem by removing the mutexes. While this did improve the runtime it still became consistent on all amount of threads. As such, we came to the conclusion it's our job_queue. The queue prevents faster runtimes for more threads, as no matter what, each thread has to wait for their turn to pop. The more threads, the more threads have to wait, meaning eventually the time evens out on more threads.

Additionally, the runtimes in general fluxuate. This may be caused by caching. We observed that some calls would initially run slower but successive calls would gradually improve the runtime to a point.

Conclusion

We've learned how concurrency works in c and what benefits and limitations it brings. We learned the importance of mutexes and wait conditions when working with multiple threads that share resources, and how the process of creating threads using worker functions works. Our testing showed the impact mutexes and wait conditions have on runtime i.e., more threads does not linearly correlate with a faster run time, instead proving to be a more logarithmic correlation.