Assignment 2 — Concurrent Programming

Schmidt, Victor Alexander, `rqc908`
Ibsen, Helga Rykov, `mcv462`
Barchager, Thomas Haulik, `jxg170`

Sunday 16:00, October 30th

With respect to the question of how to **ensure mutually exclusive access to shared resources**, including book-keeping data in the job queue, and the global histogram in task 3, it is essential to use ”locks” (`mutex`) on accessing shared variables — in our case, it is the the `job_queue` structure. To be more specific, it it critical to lock the `job_queue` before `pushing` an element to and `popping` an element from the queue in order to avoid the scenario of possible race conditions, when several threads are trying to push to and pop from the queue simultaneously.

Another possible undesirable outcome with multi-threaded implementation is a deadlock. To avoid it, it is critical to make sure that those threads that are put to sleep by `pthread_cond_wait()` are ”awakened” at the right moment so that they can continue their work when the desired condition has occurred. For instance, in `job_queue_push`, if the `job_queue` is full, the threads are blocked/put to sleep until the condition `queue_cond_not_full` satisfies. Likewise in `job_queue_pop()`, if the `job_queue` is empty, the threads are immediately put to sleep and are ”signalled” up to resume their work the very instance there are new ”jobs” in the `job_queue`.

A note should be made on the choice of the list implementation of the `job_queue`. It appears more reasonable to implement the queue as a list because the runtime of `job_queue_pop` implemented with lists is constant ($O(k)$), while the runtime of `job_queue_pop` with arrays is linear ($O(n)$).

Benchmarks

The benchmarks have been generated, using the following programs; `fauxgrep.c, fauxgrep-mt.c, fhistogram.c, fhistogram-mt.c`

The programs have been run on a computer with 6 cores and 12 logical processors. The directory used to generate these benchmarks has a **readable size** of 456712059 `bits` across 271 files.

fhistogram

# of additional threads[1]	0	1	2	4	8	12	16
real runtime (seconds)	7, 234	7, 298	3, 754	2, 146	1, 358	1, 275	1, 225
`Bytes` / second	7.891.762	7.822.555	15.207.514	26.602.520	42.039.033	44.775.692	46.603.271
`files` / second	37, 46	37, 13	72, 19	126, 28	199, 56	212, 54	221, 22

In the above table, we observe the benchmark of the program `fhistogram`. We observe that the throughput increases, as the number of threads increases. The speedup of `fhistogram-mt -n 16` in relation to `fhistogram` would be:

$$\frac{46.603.271}{7.891.762} = \frac{221,22}{37,46} \approx 5,90$$

... which means the speedup of throughput is $5,9$. Another thing to observe when looking at the table, is that the first couple of extra threads make a big difference in throughput, whereas the difference between $8, 12$ and 16 threads has smaller improvements to throughput. This is reminiscent of a logarithmic

[1]Number of threads specified with the ”-n” option. 0 additional threads indicates a synchronous program, in this case the use of `fhistogram/fauxgrep`, whereas every other column in the benchmark was created using `fhistogram-mt/fauxgrep-mt`.

function, which in this context reminds us of the visualization of *Amdahl's law*, predicting *strong scaling*. With that said it might also be worth noting that the percentile speedup is bigger, the closer the runtime gets to 0.

fauxgrep

When creating the benchmark for `fauxgrep` it is important to note, that the program requires us to pass on a "needle" / a `string` to compare to. The needle used in this benchmark is the `string` "test" in the previously stated directory.

# of additional threads[1]	0	1	2	4	8	12	16
real runtime (seconds)	$0,246$	$0,240$	$0,137$	$0,097$	$0,075$	$0,072$	$0,069$
`Bytes` / second	232.069.136	237.870.864	416.708.083	588.546.468	761.186.765	792.902.880	827.376.919
`files` / second	$1101,63$	$1129,17$	$1978,10$	$2793,81$	$3613,33$	$3763,89$	$3927,54$

To explain the results of the `fauxgrep` benchmark, it looks very alike to `fhistogram` in that its speedup based on the number of threads increases a lot with a few extra threads. To calculate the speedup of throughput between `fauxgrep-mt -n 16` and `fauxgrep`:

$$\frac{827.376.919}{232.069.136} = \frac{3927,54}{1101,63} \approx 3,57$$

... which means the speedup of throughput is roughly $3,57$. Everything said about `fhistogram` applies here as well. `fauxgrep` should probably be tested on a more suited directory (a bigger directory), the reason we stuck with the same one as `fhistogram`, is because we knew the directory's size and for the sake of consistency.

To create these benchmarks, we used the `time` prefix. If you want to check these benchmarks for yourself, run `make all` from the terminal and check the runtime when running each program. However, remember machine specifications and the directory of choice's size may cause your result to vary from ours.

To calculate `Byte` per second and `files` per second, we have made use of the formulas beneath:

$$\frac{\texttt{Byte}}{s} = \frac{\#\text{bits}}{\text{runtime in seconds} \cdot 8}$$
$$\frac{\texttt{files}}{s} = \frac{\#\text{files}}{\text{runtime in seconds}}$$

... these formulas are quite obvious, but we decided to include them for the sake of clarity.

Our task was to create a concurrent version of fauxgrep and fhistogram. But before we can do this, we first had to create a **Job Queue**. The idea of a job queue is basically that we want to able to place many jobs in a queue, which will be executed in an asynchronous manner and follow the First-In-First-Out (**FIFO**) principle. The jobs will be queued by the main thread, and executed by either one or many different threads. As stated, we made good use of both mutex locks, conditional waits and broadcasting to ensure that our thread did not encounter any deadlock or race-condition along the way. And when the program has terminated and all waiting threads must be deleted, it is important to set an implemented destroy-flag to indicate that the queue is about to be destroyed. Prior to that, though, all ”waiting” threads must be awakened by broadcasting, so that they can finish there job, if there is any, and then be deleted safely, after which the queue can also be deleted.

One of the disambiguities in our process, was the interpretation of the `job_queue` assignment description in task 1: What happens to the threads waiting to push jobs onto the queue, when the queue gets the order to destroy itself? We assume that all waiting threads should release the jobs they were trying to push and return. This implementation is unfortunately completely redundant in the context of `fhistogram` and `fauxgrep`, as there is ever only one thread pushing to the `joq_queue` and flagging the `queue` for destruction, namely the parent thread, which also means that no other threads are pushing, while the queue is being destroyed. That only happens sequentially.

`fhistogram-mt`, which is our handed-in (multi-threaded implementation) has quite a trivial implementation. In actual fact, though, it is not very trivial: it is made up of various functions and parts from the handout, already defined beforehand. To elaborate, `fibs.c` thread creation and the implementation of the worker function has been an inspiration. Besides, the non-threaded version `fhistogram.c` had the `fhistogram` function, which has been modified and otherwise completely implemented in our multi-threaded version. The `main` function was also already half-made, we just made a few modifications so that it fit the code we brought in.

`fauxgrep-mt` has been implemented in a similar way based on `fauxgrep.c` of course. We chose to implement the multi-threaded programs based on the handout, because the assignment read that we should mainly focus on multi-threading by already having defined most of the functionality of those programs.

A smooth updating of the histogram is therefore mostly just a by-product of the already well-functioning, good code of `fhistogram.c`. It is important to note that the way we ensured this smoothness in the multi-threaded version, `fhistogram-mt`, was to use a `mutex` lock when updating the global histogram, such that no two threads are racing each other when updating and printing the global histogram. Since we used a lock, we found it important not to merge / update and print the global histogram too often, to retain an acceptable througput: Otherwise the threads would all just be waiting for the lock, instead of counting the `bits`!

Finally, after having run `./fibs` multiple times, we have discovered that the produced — and similar for both one- and multi-threaded implementation — output does not match the expected one. For instance, we got the following results for Fibonacci numbers: fib(0) = 1; fib(1) = 1; fib(2) = 2; fib(3) = 3; fib(4) = 5; fib(5) = 8, etc. The expected output should have been: fib(0) = 0; fib(1) = 1; fib(2) = 1; fib(3) = 2; fib(4) = 3; fib(5) = 5. The erroneous output seems to be the result of the additional (superfluous) operation performed by the compiler either in the `job_queue` or in the `fibs` program. We are inclined to blame the latter, tough, mainly due to the fact that the results for running a single thread are the same as for running several threads — incorrect in both cases.