FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

Assignment 5: Concurrent Programming

CompSys 2017
Department of Computer Science
University of Copenhagen

Peter, Tobias & Xueying

3. December 2017

1 Compiling and running the code

The `job_queue`-library and test programs may be compiled and tested via a UNIX-like command line in the following way:

1. Unzip `src.zip` and navigate to `.../src/`.
2. Compile the library and test programs by running this command:
   `make`
3. Run the `fibs` test program with the supplied input file, using this command:
   `./fibs -n 4 < fibs.input`
4. Run the `fauxgrep-mt` test program using this command:
   `./fauxgrep-mt -n 4 return .`
5. Run the `histogram-mt` test program using this command:
   `./fhistogram-mt -n 4 .`

2 Implementing the C API

2.1 `job_queue_init()`

The `job_queue_init()`-function simply initiates all fields (conditions, semaphores and variables) of the `job_queue` struct. Worth noticing is that the size allocated for the buffer is computed as `capacity * sizeof(void*)`, to account for the size of the types it is supposed to hold.

2.2 `job_queue_push()`

The majority of the function for pushing items into the queue handles various situations where no items may be pushed into the queue.

Initially it handles the case where the queue is dead by doing nothing except returning. Thereafter the function handles the case where the queue is full and not dead by calling `pthread_cond_wait ((job_queue->cond), (job_queue->mutex))`.

Finally `job_queue_push` returns negative one in the case where the queue has died while the push-function waited. The remaining part of push computes the index of the item to be inserted and inserts it into the buffer. The index-computation may require further explanation: since we are always adding new items at the back of the queue, the back of the queue has index $i_{last} = (i_{first} + \#items)\%jq\_capacity$. The modulo-operation ensures that the queue *wraps around* and restarts at index 0 - E.G. $i_{last} = (3+6)\%8 = 1$ and so we place the new item at index 1, since index 0 and 3-7 are occupied.

2.3 `job_queue_pop()`

Like `job_queue_push()`, `job_queue_pop()` first handles situations where it is not possible to pop items from the queue. Initially it handles the case where the queue is dead. Secondly the function is set to wait while there are no items in the queue and the thread is not dead. Lastly, the function ensures that the `job_queue` isn't empty and hasn't died while pop was waiting - in that case the function must return negative one. Once everything is in order, `job_queue_pop()` proceeds to extract the *first* item of the buffer, before incrementing the index-value of the first item. Do notice that in the case where the *first* index pointed to the last position in the queue, the first-index is instead updated to point to the first position (index 0) in the queue.

2.4 `job_queue_destroy()`

The purpose of `job_queue_destroy()` is to shut down the current job queue and prevent further operations such as `job_queue_pop()` or `job_queue_push()` from manipulating the queue. The preparatory condition for this function to successfully execute is that the queue is not already dead. When the queue is empty, the pending `job_queue_destroy()` can proceed and free the memory reservation for this job queue.

2.5 Concurrency

Since `job_queue_push()`, `job_queue_pop()` and `job_queue_destroy()` may all run concurrently, all functions initially lock the `job_queue` mutex and unlock it right before they return. This ensures that all functions enjoy mutually exclusive access to the shared resource that is the job queue.

3 `fibs`

The reader might notice that the `fibs`-program returns erroneous results, E.G. `fib(10)` returns 89, the 11th fibonacci number, when the result should have been 55. In general, the program computes the $n + 1$'th fibonacci number.

4 `fauxgrep-mt`

The program `fauxgrep-mt` is an upgraded version of `fauxgrep`, which searches a directory hierarchy for files with lines that contain a keyword as known as needle. Our implementation allows the program to assign the task to numbers of threads that search several files simultaneously which will improve its performance theoretically.

We used `fauxgrep` and `fibs` as reference, added the `worker` function as well as `fauxgrep_file` function, structured a new arg_struct which enables `worker` to take two arguments: a path and a needle as `fauxgrep_file` needs. The rest of the modification is similar to `fhistogram-mt` in the upcoming section.

5 `fhistogram-mt`

Our implementation of the multi-threaded version of `fhistogram` is fairly trivial. We have included the `worker`-function and thread creation for-loop of the `fibs`-program. Meanwhile, an existing while-loop has been modified to instead push new paths to the job queue. Additionally it was necessary to introduce a mutex and modify the `fhistogram`-function to ensure that each thread has exclusvie access to writing to `stdout` - otherwise we may risk several threads writing at once.

6 Concurrent version of `fauxgrep`

6.1 Performance:

To test the performance of fauxgrep, we tried to find the word "the" in 101 copies of the Adventure of Sherlock Holmes. (1) Every text file included 9525 lines of text including line breaks. The overall size of all the files was 77.5 MB (81.294.698 bytes) The performance test was then done using different numbers of threads.

	single thread	32 threads	64 threads	128 threads
Real time	22.71	22.656	22.431	22.251
User time	0.703	0.531	0.656	0.547
System time	6.063	5.219	5.969	5.75
Bytes per second	3579687	3588219	3624212	3653530
Files per second	4.4	4.41	4.46	4.49

Table 1: Performances of `fauxgrep` and `fauxgrep-mt` with different amount of threads

As the values of the table indicates, the difference in speed between our multi-threaded version and the single-threaded version is so small that it is uncertain whether or not `fauxgrep-mt` actually runs faster than the `fauxgrep` version.

7 Concurrent version of `fhistogram`

7.1 Performance:

To test the performance on our multi-threaded fhistogram, we ran it on a folder containing 180 image files. We then ran it multiple times, using different numbers of threads. We noted the average time depending on the number of threads used and created the following table:

	single thread	32 threads	64 threads	128 threads
Real time	6.82	5.67	4.08	1.45
User time	5.01	4.87	3.74	1.13
System time	0.75	0.48	0.43	0.192
Bits processed	292.836.524	243.648.953	193.249.046	61.833.348
Bytes per second	42.937.907	42.971.596	47.364.962	42.643.688
Files per second	26	32	44	124

Table 2: Performances of `fhistogram` and `fhistogram-mt` with different amount of threads

As the table indicates, the more threads we are using, the faster the program will run. However, since the amount of bits processed is also being reduced depending on the number of threads, it gets unclear if the program is truly running faster or if it simply just processing fewer bits.

The program could be running faster though, had we adjusted the interval between each histogram-update. However, since we could not find a fine balance between a significant increase in running time and smooth updating of the histogram, we decided not to change the interval.

Besides the aforementioned, such optimizations would muddy the picture. We are solely interested in measuring the performance gained from using multiple threads. Introducing other optimizations would give us a false impression of the performance achieved.

References

[1] "Sir Arthur Conan Doyle", `The Adventures of Sherlock Holmes,` http://norvig.com/big.txt
4