Assignment 5

Computer Systems
Department of Computer Science
University of Copenhagen

Pelle Rubin Galløe <hqr241@alumni.ku.dk>,
Mads B. Christensen <rsv422@alumni.ku.dk>,
Julian S. W. Wulff <bkx655@alumni.ku.dk>

Version 1;
Due: Tuesday, December 3

1 Introduction

This is a technical report for Computer Systems B5-6, 2019. In this assignment we have been assigned to implement a C library for expressing a job queue, where several threads within a process, cooperate in solving an arbitrary problem. The goal for this implementation is to create a synchronisation mechanism that allows different threads to interact in solving a problem, without the risk of race conditions or deadlocks. With this behavior for our implementation, it should allow us to create multithreaded programs, in order to obtain better performance, which will be characterised as throughput and latency.
In order to test and validate that our job queue implementations meets the goals mentioned above, we will write concurrent versions of the two programs `fauxgrep` and `fhistogram` handed out with this assignment.

The target audience for this report is the teachers assistants of the course. The solution is developed in `C`.

As a part of the deliverables is a `Makefile` containing macros for compiling the `C` library, and creating a number of test files to be used for benchmarking fauxgrep. In order to compile the `C` library, the test programs `fibs`, `fauxgrep`, `fhistogram` and their multithreaded counterparts, the following command should be run in the terminal with the `Makefile`:

```
$ make all
```

In order to have some files in varying size and content, which can be used to benchmark `fauxgrep` with, we extended the `Makefile` with the command:

```
$ make testfiles
```

This command uses the `/dev/urandom` file to create 11 files ranging from 1 - 32 MB

2 Implementation of job_queue.c

In order to use the job queue in a concurrent program, the job queue is implemented such that the program should initialize an already allocated `struct job_queue` that holds all the necessary information about the job queue. In our implementation we decided to implement this struct as shown in listing 1

```
struct job_queue {
  void** jobs;
  int capacity;
  int cnt;
  int init;
  pthread_mutex_t lock;
  pthread_cond_t job_empty;
  pthread_cond_t job_full;
};
```

Listing 1: Implementation of `struct job_queue`

Here `void** jobs` holds an array of `void*` that points to the "jobs" pushed onto the queue. `int capacity` is used to express the size of the job queue i.e. how many jobs that can be queued at the same time. This is also used when the job queue gets initialized in order to allocate enough memory to hold all jobs pushed to the queue. To keep track of how many jobs there is in the queue at any given time, we implemented the job queue with `int cnt` that holds this information. We also decided to add `int init` to the struct. This parameter is used as a flag to indicates whether or not a job queue has been initialized or not. This ensure that it is not possible to initialize an already initialized job queue before it has been destroyed. Furthermore it is also used to signal when the job queue has been destroyed. Lastly the struct holds `pthread_mutex_t lock`, `pthread_cond_t job_empty` and `pthread_cond_t job_full`. These are used for the job queue to implement the correct producer/consumer thread behavior discussed in *Operation Systems, ch. 30*[1].

Before the job queue can be used it is implemented such that is has to be initialized by calling the `job_queue_init`, with the allocated `struct job_queue` and the capacity. This functions makes sure to, allocate memory for holding

[1] R.H. Arpaci-Dusseau and A.C. Arpaci-Dusseau, Operating Systems: Three Easy Pieces, Arpaci-Dusseau Books, August, 2018 (Version 1.00)

the number of jobs specified by the capacity, setting the initial job counter `int cnt` to 0 and `int init` to 1 (signaling that the job queue has be initialised and not destroyed yet). Lastly it also initializes the mutex and pthread conditions.

Now that the job queue and its initialization has been implemented we need a way to, push and pop jobs on and of the queue, and lastly destroying the queue. Destroying the queue should allow the last threads to finish their job before terminating the process.

For the above mentioned purposes we implemented the 3 functions; `job_queue_push`, `job_queue_pop` and `job_queue_destroy`. Our implementation of these 3 functions follows pretty much the same pattern, which is the one discussed in *Operation Systems, ch. 30*. Using this pattern we implemented the functions such that they start out by calling `pthread_mutex_lock()` on the job queues' mutex. After this, the functions enter a while loop containing a `pthread_cond_wait()`. By doing this we make sure to check that the conditions for pushing, popping or destryoing is correct in the while loop. When the function receive the correct signal it wakes up at checks the conditions again. If the conditions is correct the functions will either never enter it or break out of it, following the execution the necessary actions, e.g. increment or decrement **int cnt**, adding or extracting a job in the arrat, and signaling that that a jobs has been pushed or popped, with a `pthread_cond_signal()`. Before return we make sure to unlock the mutex by calling `pthread_mutex_unlock()`. Here locking and unlocking the mutex makes sure that the data of the job queue is protected by providing mutual exclusion and schedule the accesses.

3 Implementing multithreading of fauxgrep

The fauxgrep program is run with a directory path and a " needle".

```
$ ./fauxgrep needle path
```

It searches the directory hierarchy for files with lines that contain the "needle" substring. Our job is to rewrite the `fauxgrep.c` file (handed out in the src folder for the assignment) in the fauxgrep-mt.c file, such that we improve the performance of the program by allowing it to be multithreaded with the use of our implementation of the job queue.

For us to use our implemented `job_queue.c` we need a worker function. This "worker" will function as our `start_routine` for the `pthread_create()`. Since `pthread_create()` only takes one `void*` as arguments for the `start_routine`, and the function `fauxgrep` takes two arguments, we implemented a struct we call `workerData`, to work around this problem. This struct contains the fields that our worker will need in order to run the `fauxgrep` rutine. `workerData` contains two fields: a `struct job_queue`, and a `char const* needle`. Our

worker will need the job_queue to pop jobs from (a path to a file) and what to search for (the needle as a char const*). It unpacks the void* argument as a workerData struct. Then enters a loop, either popping a job from the queue and searching it, or breaking out of the loop.
To search the file we use a slightly altered `fauxgrep_file()` function from `fauxgrep.c`. As we have multiple threads calling this function, which prints information to the user, we protect the `stdout` file with `mutex_lock` and `mutex_unlock`.

These workers are created within a for loop running k times. This is determined by the input when the function is called as shown below

```
$ ./fauxgrep -n k needle path
```

We are not asked to implement the traversal of the directory hierarchy, we are given that functionality. We have to interpret the traversal to push a job to the queue. If the traversal finds a file we push a job to our jobqueue in workerData and pass the path of the file as our `data*`, making sure to use `strdup`.

When all jobs have been pushed to the queue we destroy our job queue, making sure to wait until all the threads/workers have finshed their jobs by terminating. This is done by running a loop that calls `pthread_join()` on each of the threads created.

3.1 Benchmarking

In order to benchmark `fauxgrep` against `fauxgrep-mt` we first need some testfiles on which we can search for the needle. These files can be created by using the makefile command as described in the introduction. With these testfiles we benchmarked the two programs by running the following commands on a MacBook pro with 4 physical cores and 8 logical cores:

```
$ time ./fauxgrep ab testfiles
real    0m0.349s
user    0m0.056s
sys     0m0.108s
$ time ./fauxgrep-mt -n 4 ab testfiles
real    0m0.202s
user    0m0.068s
sys     0m0.072s
$ time ./fauxgrep-mt -n 8 ab testfiles
real    0m0.172s
user    0m0.076s
sys     0m0.052s
$ time ./fauxgrep-mt -n 10 ab testfiles
real    0m0.190s
user    0m0.080s
sys     0m0.064s
```

By doing this we see that the time to execute the program is reduced almost by half, by running it with 4 threads, utilizing the 4 cores of the computer. Running the program with 8 threads further decreases the execution time, but using more threads than 8 increases the execution time. This is probably due

to the overhead of having more threads the CPU's logical cores.

When calling the macro `make testfiles` discussed in the introduction we create 11 files with a total size of 154 MB. When we benchmark our program this gives us a throughput of;

1. `time ./fauxgrep ab testfiles`
   (a) $154/0,349 \approx 441,3$ MB per second
   (b) $11/0,349 \approx 31,5$ files pr second (given files with a average of 14MB)
2. `time ./fauxgrep-mt -n 4 ab testfiles`
   (a) $154/0,202 \approx 762,3$ MB per second
   (b) $11/0,202 \approx 54,5$ files pr second (given files with a average of 14MB)
3. `time ./fauxgrep-mt -n 8 ab testfiles`
   (a) $154/0,172 \approx 895,3$ MB per second
   (b) $11/0,172 \approx 64,0$ files pr second (given files with a average of 14MB)
4. `time ./fauxgrep-mt -n 10 ab testfiles`
   (a) $154/0,190 \approx 810,5$ MB per second
   (b) $11/0,190 \approx 57,9$ files pr second (given files with a average of 14MB)

When we tested out the functions we observed the the execution time was largely dependent on the search term. Since we used random generated files we had to make really small search terms in order to find some matches. In the above example we used "ab" as the search term. If used for example used "a" the expectation time increased dramatically. We are pretty confident that this is due to the mutex around the `printf()`, which creates a deadlock. Calling the function on the testfiles with a search term that does not exist in any of the files gives a really fast execution time, which is why we are pretty confident about the reason for the deadlock.

4 Conclusion

We have implemented the job queue as specified in the API in the assignment text. We have tested the correctness of our implementation by running the multithreaded program `fibs` handed out with the assignment. When running this program it calculates the correct numbers and terminates(not prematurely) after the last jobs has finished when pressing Ctrl + D signaling EOF. Furthermore we tested that our implementation of `job_queue_push` blocks if the queue is full instead of failing by reducing the capacity to 2 and pushing 42 to the queue 8 times in a row, making sure that fibs also prints the result 8 times.