A2: Concurrent Programming

Computer Systems 2023-24
Department of Computer Science
University of Copenhagen

David Marchant and Troels Henriksen

Due: Sunday, 29th of October, 16:00
Version 2 (October 13, 2023)

This is the third assignment in the course Computer Systems at DIKU. We encourage pair programming, so please form groups of 2 or 3 students. Groups cannot be larger than 3, and we strongly recommend that you do not work alone.

For this assignment you will receive a mark out of 4 points; You must attain at least half of the possible points to be admitted to the exam. For details, see the *Course description* in the course page. This assignment belongs to the OS category together with A1. Resubmission is not possible.

Introduction

> *When we had no computers, we had no programming problem either. When we had a few computers, we had a mild programming problem. Confronted with machines a million times as powerful, we are faced with a gigantic programming problem.*
>
> — Edsger W. Dijkstra

This assignment is about writing programs in which several threads within a process cooperate in solving a problem. You will need to implement a synchronisation mechanism that allows the different threads to interact without risk of race conditions or deadlocks.

For this assignment, the overall purpose of multithreaded programming is to obtain *performance*, which can be characterised in two ways:

Throughput: how fast can we solve the computational problem? Usually measured in units per time. For example: how many bytes per second can we read from a disk?

Latency: how quickly can we get a response? For example: how long does it take to read just a single byte from the disk? Or, how long does it take from the user clicks a button, to the program responds to that click?

These conceps will become particularly important once we get to the network portion of the course, but we can already start gaining an appreciation for them in the context of multithreaded programming.

When designing a system, throughput and latency are often in tension, and improving one tends to harm the other. This is particularly the case for user interfaces: whenever we spend some time informing the user of the progress of some computation, we have wasted clock cycles that could have gone into solving the problem. However, from a user's point of view, low response latency is often more important. Resolving the tension between throughput and latency is a remarkably thorny problem in general, and you are not expected to come up with a brilliant solution. Instead, we expect you to reflect on the trade-offs available to you, and discuss how you characterise (through measurements) how your solution balances the need for throughput and latency.

Concretely, you will develop a general-purpose concurrent *job queue*, that allows a thread to post *jobs*, which are then read and executed by several available *worker threads*. You will then extend two handed-out single-threaded programs to use the job queue to obtain concurrent execution, hopefully achieving higher performance.

Task 1: Implement the job queue (20%)

The job queue API is already defined for you, in the form of a C header file (`job_queue.h`). You are also given a stub of the corresponding implementation file (`job_queue.c`), along with a `Makefile` and an example program (`fibs.c`) that makes use of the API.

The C API

The job queue API is a little different than what you have seen before. Most significantly, the definition of `struct job_queue` is in the header file, not the implementation file. An API function is used for initialising an already allocated `struct job_queue`:

```
int job_queue_init(struct job_queue *job_queue, int capacity);
```

It is the users responsibility to allocate memory for a `struct job_queue`, after which the function `job_queue_init()` can be used to initialise it, as follows:

```
struct job_queue q;
job_queue_init(&q, 64);
```

The second argument indicates the capacity of the queue (see below). We can insert arbitrary `void*` pointers into the queue with the following function:

```
int job_queue_push(struct job_queue *job_queue, void *data);
```

If the queue is already full, `job_queue_push()` will not fail, but instead block until space becomes available. It is permitted for multiple threads to call this function at the same time. Elements are removed from the queue with the following function:

```
int job_queue_pop(struct job_queue *job_queue, void **data);
```

The second argument is a *pointer to a pointer*, which is where the removed element will be stored. If called on an empty queue, `job_queue_pop()` will *block* until an element becomes available. This happens when some other thread calls `job_queue_push()`. It is permitted for multiple threads to call this function at the same time.

When we are done with a job queue, we must *destroy* it:

```
int job_queue_destroy(struct job_queue *job_queue);
```

The `job_queue_destroy()` function must block until the queue is empty, and then destroy it. This ensures that any work that has been pushed will not be lost. You may assume that only one thread will call this function. If any threads are blocked on a call to `job_queue_pop()` when `job_queue_destroy()` is called, the threads must be woken with `job_queue_pop()` returning -1. **This is the tricky part.**

Task 2: Write a concurrent version of `fauxgrep` (20%)

The program `fauxgrep`, inspired by `fgrep`, searches a directory hierarchy for files with lines that contain some substring (called the "needle"). Whenever a match is found, `fauxgrep` prints a file name and line number, followed by the line.

The directories to search are given as command line options, following the needle. For example, searching for `malloc` in our reference solution to A1:

```
$ ./fauxgrep malloc A1/ref
A1/ref/id_query_eytzinger.c:62:   struct eytzinger_data *data = malloc(
    sizeof(struct eytzinger_data));
A1/ref/coord_query_naive.c:18:   struct naive_data *data = malloc(sizeof(
    struct naive_data));
A1/ref/id_query_naive.c:17:   struct naive_data *data = malloc(sizeof(
    struct naive_data));
A1/ref/id_query_indexed.c:32:   struct indexed_data *data = malloc(sizeof
    (struct indexed_data));
A1/ref/coord_query_kdtree.c:55:   struct node *node = malloc(sizeof(
    struct node));
A1/ref/coord_query_kdtree.c:80:   struct kdtree_data *tree = malloc(
    sizeof(struct kdtree_data));
A1/ref/record.c:155:   struct record *rs = malloc(capacity * sizeof(
    struct record));
A1/ref/id_query_binsearch.c:48:   struct binsort_data *data = malloc(
    sizeof(struct binsort_data));
A1/ref/id_query_indexed2.c:31:   struct indexed_data *data = malloc(
    sizeof(struct indexed_data));
```

For this assignment you are given a fully functional single-threaded implementation of `fauxgrep`, and asked to create a similar program, `fauxgrep-mt`, that uses multithreading to search several files simultaneously. The intent is that this improves the program throughput. You are given a template in `fauxgrep-mt.c`, which you must finish. The first two command line options

to `fauxgrep-mt` may be `-n` *k*, which specifies that *k* worker threads should be used.

The actual traversal of the directory hierarchy is implemented for you, and should not be changed (nor even understood in detail—the Unix directory traversal APIs are fairly complicated). For each file found, `fauxgrep-mt` calls the function `fauxgrep_file()` with the file name and needle as arguments. This function then reads through the file on a line-by-line basis, and prints any matches it finds.

Your task is to modify `fauxgrep-mt.c` as follows:

- Initialise a `job_queue` (see task 1).
- Instead of immediately searching each file found in the directory traversal, it should push the name of the file to the job queue. Remember to use `strdup()`, as the memory containing the file name will be modified when the traversal continues.
- Launch a number of worker threads (given by the `-n` option), where each executes a loop reading a file name from the job queue and search the given file. The threads should exit when the job queue is shut down.
- When all files have been searched, the program should terminate with a succesful exit code.

If the job queue has been implemented correctly, you should not need much synchronisation in `fauxgrep-mt.c` itself—ensuring exclusive access when printing matched lines should be sufficient.

Task 3: Write a concurrent version of `fhistogram` (20%)

The `fhistogram` program is similar to `fauxgrep`, in that it performs analysis on a directory of files. However, instead of printing lines matching some string, it computes a running *histogram* of the statistical distribution of bits in the bytes of the files:

```
$ ./fhistogram A2/ref/
Bit 0: *********
Bit 1: ********
Bit 2: ********
Bit 3: ********
Bit 4: *******
Bit 5: **********
Bit 6: **********
Bit 7: ****
458811 bits processed...
```

The results show that bits are generally uniformly distributed within the bytes of the files within the given directory, but the most significant bit (bit 7) is relatively rarely set, and bits 5 and 6 are relatively frequently set.

What is not visible above is that the histogram is updated *continuously* as more files are processed. Run `fhistogram` yourselfto see this. Although most Unix tools do not provide progress indicators, and wait until the end before they produce the results, it is on most platforms considered good UI design to provide partial visual indicators or partial results to indicate that a long-running computation is taking place. In `fhistogram`, this is done by calling a function `print_histogram()` at appropriate intervals. Not too often, or else the overall performance of the application will suffer, and not too infrequently, or the progress updates will seem choppy. In `histogram`, the printing is done every 100,000 bytes—but this is not necessarily the best interval.

Your task is as follows. As in task 2, you are given a program template, `fhistogram-mt.c`. You must finish this program such that it uses a job queue to process multiple files concurrently. Specifically, for each file you must compute a local histogram (represented as an array of eight `int`s), which is then added to a global running histogram at appropriate intervals, and printed to the screen. You are given convenience functions `print_histogram()`, `merge_histogram()`, and `update_histogram()` to help in this—see `histogram.h` for details.

Task 4: Write a short report (40%)

Alongside your solution, you should submit a short report. The report should:

- Discuss how you ensure mutually exclusive access to shared resources, including (but not limited to) book-keeping data in the job queue, and the global histogram in task 2.
- Benchmark the multithreaded programs written in Task 2 and 3 compared to the original single-threaded versions. Show this for different thread counts (the `-n` option). Explain the results you obtain. Whether you get speedup depends on the test data you are using and the machine you are running the programs on, but you should try to explain the results.

  Show the total running time(s), and characterise the throughput in both *files per second* and *bytes per second*.
- Show how to compile your code and reproduce your benchmark results.
- For Task 2, Discuss how you ensure smooth updating of the running histogram, without reducing throughput too much. You may, at your own discretion, instrument the program such that it measures the average time between updating the histogram display. The `gettimeofday()` function is useful for this purpose.
- Discuss the non-trivial parts of your implementation and your design decisions, if any.
- Explain *why* you wrote your code the way you did, not merely *what* it does. *Purpose is more important than mechanism.*
- Disambiguate any ambiguities you might have found in the assignment.

The above is not intended as a table of contents. Do not structure your report as answers to a series of bullet points. In fact, the bullet points above intentionally overlap, so mirroring their structure in the report will yield a repetitive text. Make sure there is a logical and textual *flow*, that you define terms before you refer to them, and that there is a good narrative.
The report is expected to be 2-3 pages and must not exceed 5 pages.

What to Hand In

Your submission must consist of a modification of the handed-out code, retaining the same structure, as well as a report in PDF format. The code handout consists of a directory `src`, which contains the following files:

`.gitignore`**:** A suitable .gitignore file, should you choose to use Git.

`Makefile`**:** The file that configures your `make` program. You will have to modify this file when you add more test programs.

`fibs.c`**:** An example program that demonstrates how to use the job queue API.

`job_queue.h`**:** The header file for the job queue. You will need to modify the definition of `struct job_queue` in this file, but nothing else.

`job_queue.c`**:** A skeleton implementation of the job queue. It compiles, but every function fails.

`fauxgrep.c`**:** A simplistic `fgrep(1)` clone.

`fauxgrep-mt.c`**:** A slightly modified `fauxgrep.c` that you must extend with multithreading via a job queue.

`fhistogram.c`**:** A single-threaded program for doing statistical analysis of bit distributions.

`fhistogram-mt.c`**:** A slightly modified `fhistogram.c` that you must extend with multithreading via a job queue.

`histogram.h`**:** Code shared between `histogram.c` and `histogram-mt.c`

Submission

You should hand in a ZIP archive containing a `src` directory containing all *relevant* files (no ZIP bomb, no compiled objects, no auxiliary editor files, etc.).

To `make` this a breeze, we have configured the `Makefile` such that you can simply execute the following shell command to get a `src.zip`:

```
$ make ../src.zip
```

Alongside a `src.zip` submit a `report.pdf`. For submission, sign up for a group at:
`https://absalon.ku.dk/courses/70194/groups#tab-23701`
Hand-in as a group as in other courses.