Computer Systems - A2: Concurrent Programming

Aditya Fadhillah (hjg708) Usama Bakhsh (mhw630) Nikolaj Schaltz (bxz911)

October 30, 2022

Design and thoughts

We started out with task 1 since both **fhistogram** and **fauxgrep** builds upon task 1. Here we first added the necessary fields to **jobqueue.h** in the struct **job_queue**. We also added the helper function job, to handle the data and the stuct void pointer next. In the **job_queue.c** we had to implement four functions; **job_queue_init** to create an array. It takes in the struct **job_queue** and capacity which is how big the array. **job_queue_destroy** which uses threads to destroy an array if the array is empty. **job_queue_push** to insert an integer into the array and **job_queue_pop** to remove and integer from the array. We ensure mutually exclusive access to shared resources by have locking the critical area which is the while statement in the **job_queue_destroy**, **job_queue_push-pop**. This is so our shared resources does not get overwritten by other threads. Another way we block threads from interrupting each others work is with **pthread_mutex_wait** which blocks other threads from doing their work until a condition have been fulfilled. An example would our **job_queue_push**;

```
int job_queue_push(struct job_queue *job_queue, void *data) {
          if (job_queue != NULL) {
                  pthread_mutex_lock(&job_queue -> mut);
                while (job_queue -> count + 1 > job_queue -> capacity) {
                        pthread_cond_wait(&job_queue -> emptyQ, &job_queue -> mut);
                }
                struct job *newJob = malloc(sizeof(struct job));
                newJob -> next = job_queue -> head;
                newJob -> data = data;
                job_queue -> head = newJob;
                job_queue -> count++;

                pthread_cond_signal(&job_queue -> fillQ);
                pthread_mutex_unlock(&job_queue -> mut);
                return 0;

        }
        return 1;
}
```

Here we can see that **pthread_cond_wait** utilises the lock **mut** and the condition for **emptyQ** that **newjob** (an integer) should be inserted in the head of the array. We use **pthread_cond_signal** to unblock when the condition has been reached. In **job_queue_destroy** we used **pthread_cond_broadcast** instead of **pthread_cond_signal**, because we want to release all our threads when the array has been destroyed.
Our **fhistogram-mt** and **fauxgrep-mt** are designed by extracting code from fibs, including the way it book-keeps by using the worker function and book-keeps. Other design decisions include; deciding to track the amount of jobs currently in the **job_queue** with count, as this allowed us to easily check whether the **job_queue** is at max capacity or if its empty as we cant push a job on a full **job_queue** and we cant pop a job on an empty **job_queue**. We also decided to use a linked list as our data structure for **job_queue**. The reason for this is that it allowed us to to easily clear a job from the

job_queue after popping. Each job has a pointer to the next job, this allowed us to easily change the first or last element in the **job_queue** when pushing and popping, as we just had to change the pointers.

Running the programs and benchmarks

Our **Makefile** helps us compile and run all of our implementations. To compile the implementations type in in the terminal

```
$ make all
```

To run the different implementation we use different commands, but they are not too dissimilar to one another.

```
$ ./fauxgrep (input) (file/folder)
$ ./fauxgrep-mt -n (thread) (input) (file/folder)
```

Benchmark for fauxgrep

To determine the run time for our programs for fauxgrep and fauxgrep-mt we use the **c** command

```
$ time -p
```

before we type in the run command. The **time** command gives us the real, user and system time for that specific run. We are only interested in the real time, real time is the time that has elapsed during a specific run, and we considered it to be the running time. So we will not take the other the two into consideration.
For our testing data we use the **src** folder from the previous assignment, **A1**. We then searched for the string **int**. The command for this would be

```
$ time -p ./fauxgrep int ../../A1/src
```

The real time fluctuates a little each run, so we run the program 5 times and each time we noted down the real time, so we can take the average time of them.
Real time: 1.28, 1.10, 1.48, 1.44, 1.35
We then calculate the average time, to find the running time of **fauxgrep**.

$$RT_0 = \frac{1.28 + 1.10 + 1.48 + 1.44 + 1.35}{5} = 1.33\text{seconds}$$

To find the throughput in bytes per second we have to know how large the testing data is, and by looking at the folder size of **A1** folder, we now know that the size of the testing data is 82,392,061 bytes. We then find the throughput by dividing the size of the folder with the average running time. The throughput for **fauxgrep** is

$$TP_0 = \frac{82,392,061\text{bytes}}{1.33\text{seconds}} = 61,948,918.0451\text{bytes/seconds}$$

We then do the same with **fauxgrep-mt** where we calculate the average running and throughput for different amount of threads. We test for 1, 4, 10, 50, 200 and 1000 threads.
Running time:

$$RT_1 = \frac{1.28 + 1.08 + 1.11 + 1.47 + 1.41}{5} = 1.27 \text{ seconds}$$

$$RT_4 = \frac{1.26 + 1.40 + 1.28 + 1.17 + 1.13}{5} = 1.248 \text{ seconds}$$

$$RT_{50} = \frac{1.09 + 1.24 + 1.16 + 1.04 + 1.15}{5} = 1.136 \text{ seconds}$$

$$RT_{200} = \frac{1.10 + 1.05 + 1.39 + 1.37 + 1.43}{5} = 1.268 \text{ seconds}$$

$$RT_{1000} = \frac{1.46 + 1.47 + 1.41 + 1.52 + 1.31}{5} = 1.434 \text{ seconds}$$

Throughput:

$$TP_1 = \frac{82,392,061 \text{ bytes} 1.27 \text{ seconds}}{=} 64,875,638.5827 \text{ bytes/seconds}$$

$$TP_4 = \frac{82,392,061 \text{ bytes}}{1.248 \text{ seconds}} = 66,019,279.6474 \text{ bytes/seconds}$$

$$TP_{50} = \frac{82,392,061 \text{ bytes}}{1.136 \text{ seconds}} = 72,528,222.7113 \text{ bytes/seconds}$$

$$TP_{200} = \frac{82,392,061 \text{ bytes}}{1.268 \text{ seconds}} = 64,977,966.0883 \text{ bytes/seconds}$$

$$TP_{1000} = \frac{82,392,061 \text{ bytes}}{1.434 \text{ seconds}} = 57,456,109.484 \text{ bytes/seconds}$$

From the results we can see that with multi threading the program runs faster at first but as the number of threads become larger, the program ended up running slower than without multi threading.

Benchmark for fhistogram

Just like in **fauxgrep** we used the folder **A1** as our testing data to find the average running time and throughput of **fhistogram** and **fhistogram**. The command for finding the running time of **fhistogram** is

```
$ time -p ./fhistogram ../../A1/src
```

Running time:

$$RT_0 = \frac{0.17 + 0.29 + 0.24 + 0.22 + 0.21}{5} = 0.226 \text{ seconds}$$

And as we use the same folder, **A1** we already know its size, 82,392,061 bytes. So the throughput for **fhistogram** is

$$TP_0 = \frac{82,392,061 \text{ bytes}}{1.226 \text{ seconds}} = 364,566,641.593 \text{ bytes/seconds}$$

We then do the same for **fhistogram-mt** where we calculate the average running and throughput for different amount of threads. We test for 1, 4, 10, 50, 200 and 1000 threads.
Running time:

$$RT_1 = \frac{0.21 + 0.19 + 0.24 + 0.22 + 0.24}{5} = 0.22 \text{ seconds}$$

$$RT_4 = \frac{0.22 + 0.21 + 0.18 + 0.19 + 0.20}{5} = 0.2 \text{ seconds}$$

$$RT_{50} = \frac{0.21 + 0.20 + 0.21 + 0.23 + 0.24}{5} = 0.218 \text{ seconds}$$

$$RT_{200} = \frac{0.23 + 0.24 + 0.33 + 0.30 + 0.38}{5} = 0.296 \text{ seconds}$$

$$RT_{1000} = \frac{0.38 + 0.35 + 0.36 + 0.45 + 0.49}{5} = 0.434 \text{ seconds}$$

Throughput:

$$TP_1 = \frac{82,392,061 \text{ bytes} 0.22 \text{ seconds}}{=} 374,509,368.182 \text{ bytes/seconds}$$

$$TP_4 = \frac{82,392,061 \text{ bytes}}{0.2 \text{ seconds}} = 411,960,305 \text{ bytes/seconds}$$

$$TP_{50} = \frac{82,392,061 \text{ bytes}}{0.218 \text{ seconds}} = 377,945,233.945 \text{ bytes/seconds}$$

$$TP_{200} = \frac{82,392,061 \text{ bytes}}{0.296 \text{ seconds}} = 278,351,557.432 \text{ bytes/seconds}$$

$$TP_{1000} = \frac{82,392,061 \text{ bytes}}{0.434 \text{ seconds}} = 189,843,458.525 \text{ bytes/seconds}$$

Just like with **fauxgrep-mt** can we see from the results that with multi threading the program runs faster with multi threading, but there is of course an upper limit where this increase in running time can happen, with large enough number of threads the running time will instead become slower than without multi threading

Discussion of the data from benchmarking

As we saw earlier the number of threads does not change the time it takes to search all the way through in **fauxgrep**. If we had a bigger file, then we could be able to spot the difference in running time and bytes per second. We would be able to see that if we had too few threads or too many threads would influence the running time negatively. A perfect amount of threads would have to be found, to get the smallest amount of seconds to run the program. We ensure that the **fhistogram** updating is smooth by only merging and printing it every 100,000 char that is read. This reduces the lag and processing power significantly compared to if it were to merge and print it for every single char that it reads.

We also tried to make it so the **fhistogram** would update more smoothly by reducing the byte interval from 100,000. We come to the conclusion that when the interval approaches 15,000 it begin to negatively influence the running time of the program. We will therefore compare the running time when the byte interval is 100,000 and when it is 20,000, and this comparison is for **fhistogram-mt** with 3 threads.
Running time:

$$(100k)RT_4 = \frac{0.22 + 0.21 + 0.18 + 0.19 + 0.20}{5} = 0.2 \text{ seconds}$$

$$(20k)RT_4 = \frac{0.20 + 0.21 + 0.22 + 0.22 + 0.21}{5} = 0.212 \text{ seconds}$$

Throughput:

$$(100k)TP_4 = \frac{82,392,061 \text{ bytes}}{0.2 \text{ seconds}} = 411,960,305 \text{ bytes/seconds}$$

$$(20k)TP_4 = \frac{82,392,061 \text{ bytes}}{0.212 \text{ seconds}} = 388,641,797.17 \text{ bytes/seconds}$$

Comparison in pro-cent:

$$\frac{411,960,305 - 388,641,797.17}{411,960,305} 100\% = 0.057\%$$

This means that the program run 0.057% slower when the byte interval is reduced to 20,000. This is an insignificant reduction in running time for the program to update more smoothly.

Disambiguate and Ambiguities

In this assignment there have been a slight confusion in particular in task 4 where it is not clear whether we are supposed to talk about **fhistogram** or **fauxgreb**. The assignment calls **fhistogram** for task 2 in task 4. We have decided to talk about what we think is appropriate to talk about.