COP290 Lab-2 Report

Amaiya Singhal	Sarwagya Prasad	Om Dehlan
2021CS50598	2021CS10639	2021CS10076

PART 2

Testing

To run and check each component of our project, we have made a `makefile` which compiles the needed source files, create an executable file and then runs it.

```
files = $(shell ls data | sed "s/^/data\//")

main:
    gcc -o obj/main test/main.c src/list.c src/hm.c src/mythread.c
    ./obj/main $(files)

list:
    gcc -o obj/list_test test/list_test.c src/list.c
    ./obj/list_test

hashmap:
    gcc -o obj/hashmap test/hashmap_test.c src/hm.c src/list.c src/mythread.c
    ./obj/hashmap
```

The makefile

- `make list` and `make hashmap` run the list and hashmap test files respectively.
- `make main` runs the main test file, with all the files present in `data` directory as arguments.

Checkpoint 1

Testing Hashmap in `hashmap_test.c`

After completing the code for the hashmap (`hm.c`), we tested it on the provided test file – `hashmap_test.c` and got the following output:

```
sarwagya@LAPTOP-RL00B8ME:~/RAM-bharose-lab2/Assignment2-COP290/part2$ make hashmap
gcc -o obj/hashmap test/hashmap_test.c src/hm.c src/list.c src/mythread.c
./obj/hashmap
Testing hashmap!
Iterate empty hashmap!
Done iterate empty hashmap!
key hello, count 23
key world, count 24
key hello, count 2
Testing hashmap done!
```

Output of `hashmap_test.c`

This is the expected output. Thus we were convinced that each individual function in `hm.c` works correctly.

Testing Hashmap in `main.c`

Next we ran the file on `main.c` by commenting out thread related commands and replacing them with a direct call of the function `readFile()` on the 4 .txt files provided in the `data/` directory.

```
sarwagya@LAPTOP-RL00B8ME:~/RAM-bharose-lab2/Assignment2-COP290/part2$ make main
gcc -o obj/main test/main.c src/list.c src/hm.c src/mythread.c
./obj/main data/in1.txt data/in2.txt data/in3.txt data/in4.txt
```

command

```
key world, count 4
key test, count 1
key hello, count 4
Testing threads done!
```

Hashmap Values

This is the expected output confirming the correctness of the implemented hashmap.

Checkpoint 2

Concurrency with threads

Firstly we implemented the functions `mythread_init`, `mythread_create` and `mythread_join` in `mythread.c`.

After commenting out lines relating to `mythread_yield`, `acquire_bucket` and `release_bucket`, we ran the `main` file on the 4 .txt files provided, the correct output was obtained.

```
key world, count 4
key test, count 1
key hello, count 4
Testing threads done!
```

Hashmap Values

This output matches the output obtained from the single thread program in Checkpoint 1.

Checkpoint 3

Locks

In this case each thread ran individually and only switched to the next thread once the previous one was completed.

Next we implemented `mythread_yield` so that the threads could run concurrently.

After commenting code related to `acquire_bucket` and `release_bucket`, we ran the `main` file and got the following output:

```
key world, count 2
key test, count 1
key hello, count 2
Testing threads done!
```

Hashmap Values

Clearly, this output was wrong as it reports less frequency of the words.

We also checked the code for a set of files such that each file contained multiple occurrences of the same word., i.e. in1.txt contained only 5 `hello`, in2.txt only contained 3 `test` and in3.txt contained only 4 `world`, and obtained the correct output:

```
key world, count 4
key test, count 3
key hello, count 5
Testing threads done!
```

Hashmap Values

Race Condition

Thus we could conclude that the threads were interfering with each other for access to the memory of each word in the hashmap, i.e., we were running into a **race condition** where different threads are trying to increment the same value, but read the same value rather than reading the value incremented by other threads.
For example if thread 1 and thread 2 both read the word **hello**, they read the current frequency of hello in the hashmap, say 2, and each increment it to 3 and write it back. So we got an incorrect value of 3 as opposed to the correct value of 4.

When we ran it on files containing only a single word, there was no chance of interfering as each thread utilized different memory and thus we obtained the correct output.

Solving the Race Condition

To solve the race condition issue, we implemented **locks** which ensured that while one thread is working on a particular memory (in our case: one slot/bucket of the hashmap), other threads could not have access to it until the thread finishes.

After implementing `acquire_bucket` and `release_bucket`, we ran the main code to get:

```
key world, count 4
key test, count 1
key hello, count 4
Testing threads done!
```

Hashmap Values

which was the expected correct output. Thus, we had resolved the race condition.

PART 3

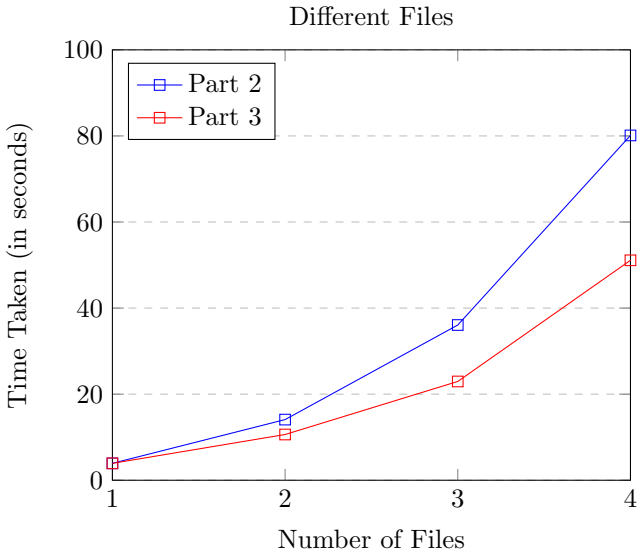
Plotting from data:

1. Different Files: We observe that the time taken is more for Part 2 than Part 3. This is because Part 2 implements concurrency whereas Part 3 uses parallelism and takes advantage of the multiple cores of the system with multiple threads running at once. This reduces the running time.

No. of Files	Part 2	Part 3
1	4.92915500	4.87719500
2	14.08958100	10.62400800
3	36.06447200	22.95074200
4	80.11478500	51.09996900
5	126.81183300	73.60158900
6	200.98808500	108.03950900

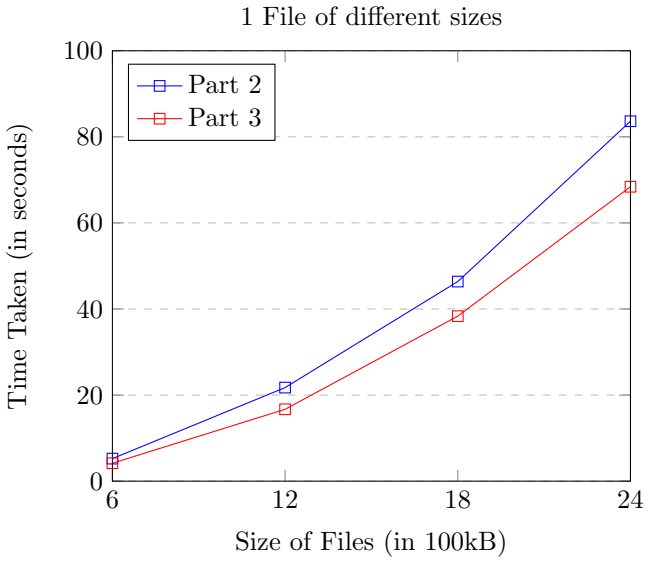
2. Same file multiple times: We observe that the times in Part 2 are more than Part 3 by a significant amount and this time is more than the first case where we ran different files. This is because running multiple instances of the same file will lead to multiple conflicts and hence increases the time significantly.

No. of Files	Part 2	Part 3
1	4.87719500	3.97829800
2	21.06525300	15.80600400
3	52.61153600	36.81120600
4	103.78924800	65.79055500



3. One File of Increasing sizes: We observe that the times in Part 2 are more than Part 3 but by a small amount. This is because since we are running on only 1 file, we do not take advantage of parallelism and hence the results in both the parts are comparable. The time is O(n^2).

No. of Files	Part 2	Part 3
1	4.87719500	3.97829800
2	21.06525300	15.80600400
3	52.61153600	36.81120600
4	103.78924800	65.79055500



---

4. Files containing same word repeated times: We observe that the times in Part 2 are almost the same as Part 3.

No. of Files	Part 2	Part 3
1	43.50957600	35.80605200
2	86.80219800	80.46090200
3	130.90629800	127.08874900
4	179.70722000	189.76057300

Tokens

Token Distribution

- Om : 10
- Sarwagya : 10
- Amaiya : 10

Token No.	Person	Work done
1	Om	Part 1
2	Om	Creating git repo
3	Amaiya	Creating list.c
4	Amaiya	Testing list.c
5	Amaiya	Creating hm.c
6	Amaiya	Testing hm.c
7	Sarwagya	Creating mythreads.c
8	Sarwagya	Creating mythreads.c
9	Sarwagya	Testing mythreads.c
10	Amaiya	Testing main.c without threads
11	Amaiya	Testing main.c with threads without yield
12	Sarwagya	Implementing mythread_yield
13	Sarwagya	Implementing locks
14	Sarwagya	Testing main.c with yield
15	Sarwagya	Testing main. with locks
16	Sarwagya	Creating make file for Part 2
17	Sarwagya	Creating make file for Part 3
18	Om	Running Part 2 code with large files
19	Om	Running Part 3 code with large files
20	Om	Debugging segfaults for large files
21	Om	Debugging hash function giving negative values
22	Sarwagya	Uploading final code on git
23	Om	Collected data for graphs and tables
24	Om	Double checked all functions with submission guidelines
25	Amaiya	Writing report in latex
26	Amaiya	Plotting graphs in latex
27	Amaiya	Making tables in latex
28	Amaiya	Checked and reviewed final code
29	Om	Writing comments for Doxygen
30	Om	Creating the doxygen file