

## Data Structure

- **Implementation**

From the given codes, I have completed the `hashmap.c` and `hashmap.h`. I implemented the given functions. At the same time, I added three essential structs in `hashmap.h`, which are called `hashmap`, `node` and `pair`.

### In `hashmap.h`:

For the struct `hashmap`, it is used to store the size of the hashmap, the 4 function pointers, `objsize`, the size (number) of the array, which is used to store the nodes in the hashmap, and a lock. This lock is `pthread_rwlock_t`, which is used for the `hashmap.c`. As for the lock in this struct, I have commented 5 other locks, such as `pthread_mutex_t` and `pthread_spinlock_t`. This is because the lock I used in `hashmap.c` is the `pthread_rwlock_t` and the rest are just for my testing and research. The reason of choosing `pthread_rwlock_t` would be discussed later in the report.

For the struct `node`, it is used to store the key, value of a node, the next node, and a struct `pair`.

For the struct `pair`, it is used to store the key and value of a node, when `*_entry` is called.

In the `hashmap.h`, there are other structs, `lockk` and `thread_data`, as well as other functions, which are not included in the scaffold originally. All of them are used for my own testing and comparison between different types of locking methods. In other words, they are not really necessary for marking. Only `hashmap.c` and `hashmap.h` are essential for this assignment.

### In `hashmap.c`:

#### **Safety Guarantees :**

I shall talk about the thread-safe here, instead of talking in every function, as I feel that it would be quite duplicating.

For the write locks, I start the lock whenever insert or remove functions are called. This is to ensure only one thread is able to access the functions per time. Not only to align the threads orderly, but also prevent deadlock occur. Hence, it is able to withstand multiple write functions without any fault. As there are several returns in the functions, I place the unlock before each return statement.

This would then ensure the threads, who are locked when accessing the function, would always be unlocked when finishing the tasks.

As for the read locks, I make the threads to get their own index number, generated from their own key first. If there is no node from that index currently, the function would just terminate. Then, lock them. I only unlock them just before the return statement of the function. This would ensure the threads, which are locked, will always be unlocked when finishing the tasks in the function.

- For `hashmap_init()`:

I simply allocated the variables in the parameter to the struct `hashmap`. For the array in the struct `hashmap`, I used the `malloc` function in the `qalloc.c` to allocate memory for it, with the size of `(node*) * size`. Since the array is a double pointer, I used a loop to set the `array[x]` to `NULL`. This to ensure the other function, when accessing this array with an index number, no zero page error would occur. In addition, I set the size of the `hashmap` equal to the size in the parameter initially. When `insert` or `remove` function is called, the size of it would increment or decrement respectively. However, the method that I used to find the index of a node is about, `index = hash(map, key) % size`, and I need to make sure the “size” in the equation should always be the same. This would be beneficial for `destroy` function. Therefore, as mentioned earlier, the variable, `arraysize`, is used here to replace the `size` to `map->arraysize`. This `arraysize` will never be changed after the initialization of the `hashmap` has started. And lastly, `pthread_rwlock_init()` occurs here.

- For `hashmap_insert()`:

A total of 3 node variables are created here. They are about the initializing of a new node with the given key and value, a node to get the current position of the node in the `hashmap` and a node to get the previous position of that node. Firstly, with the usage of the hash function, I would get the index in the `hashmap`. If the index is free of node, I would place it there. If the position is non-empty, I will iterate the next position of the node, until an empty place is found. If the insertion is successful, the size of the `hashmap` would be incremented by 1. However, if an identical key is found, which means the key has already been presented in the `hashmap`, the new value would replace the old value of that node.

- For `hashmap_get()`:

With the key provided, I would find a corresponding index of it first. The value of the node will be return as a `void*` if found and null will be return if not found.

- For `hashmap_get_entry()`:

With the key provided, I would find a corresponding index of it first. The key and value of the node will be return as a struct pair if found and null will be return if not found.

- For `hashmap_remove_value()`:

With the key provided, I would find a corresponding index of it first. The value of the node will be return as a `void*` if found and null will be return if not found. At the same time, the key of the node will be freed using the `key_del()` and the current node will be place by the next node. This is because that node is useless already. If the deletion is successful, the size of the hashmap would be decremented by 1.

- For `hashmap_remove_entry()`:

With the key provided, I would find a corresponding index of it first. The key and value of the node will be return as a struct pair if found and null will be return if not found. At the same time, the key of the node will be freed using the `key_del()`, value of it will be freed using `val_del()` and the current node will be freed and its position will be placed by the next node. If the deletion is successful, the size of the hashmap would be decremented by 1.

- For `hashmap_size()`:

There would just return the current size of the hashmap.

- For `hashmap_destroy()`:

With the use of `arraysize` in the struct `hashmap`, I started a loop to free all the nodes in the hashmap, as well as the use of `key_del` and `val_del` before freeing a node. At this point, I created a condition to check if the node has a valid key or value, to prevent errors like double free error, free after use error or zero page error. When the loop finishes, I free the array itself. Lastly, `pthread_rwlock_destroy()` occurs here.

I shall not talk about other functions in this hashmap.c. This is because they are just about my own testing.

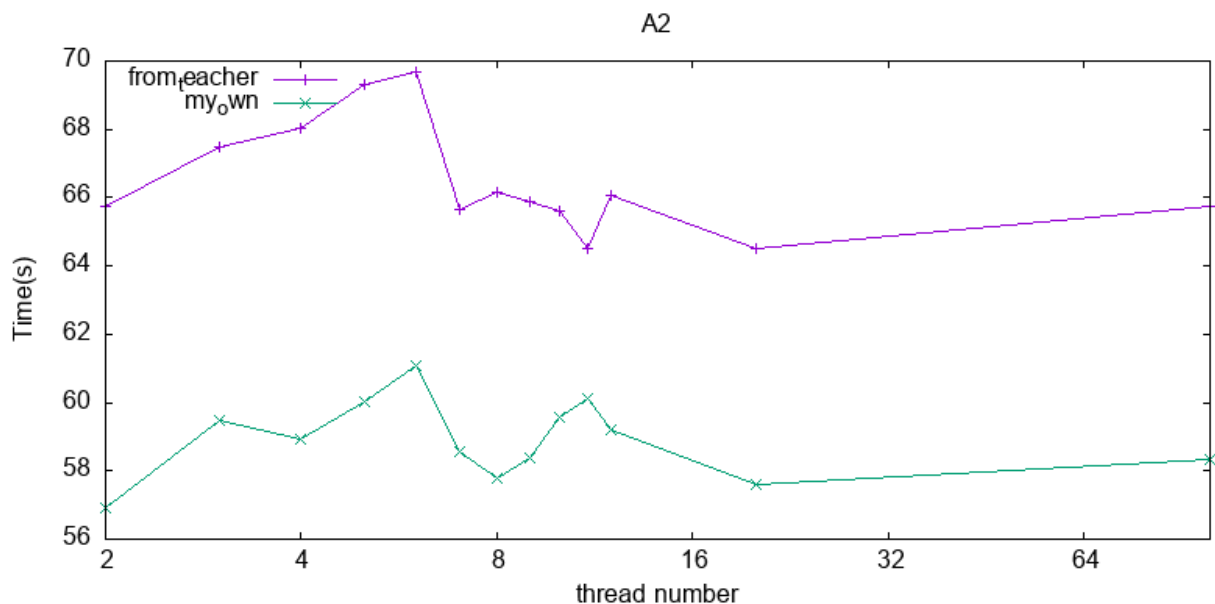
## Performance and report

### System environment / Setup:

- I am using a virtual machine for this assignment. A Linux system, with Ubuntu 64 bits, a ram of 2888MB, storage of 32.9gb and 6 cores. To be honest, after did some research [IEEE,2017], I cannot figure out the thread-to-core mapping for my program. Below is the explanation.

### Performance and Explain:

- After some researching, I understand that I need to prevent deadlocks from occurring in order to make my functions thread safe. [Balji]
- Below is my program, after using the maptest.o provided by the teacher, against the 'test' provided by the teacher:



For this, the average time (my program):  $766.08/13 = 58.929$  and the average time ("test" provided by teacher):  $864.368/13 = 66.490$ . The evidence of the timings is shown in Figures, from 1 to 14.

After calculation, my time taken to run the program used a roughly 7.561 seconds faster than the one provided by the teacher.

Just by looking at the graph, there are fluctuations in between the threads. In general, the time taken increase when the thread number increases from 2 to 5 (I have set the x-axis with log scale 2) and decrease from then. One of the reasons for this weird behavior I would think of is about the stability of the virtual machine. This is because I have tried to get the time of execution in different time during a day. The result I got differs in each trial. In overall, I think this kind of behavior is kind of normal.

In my opinion, the time taken should decrease gradually as the number of threads increases or stay almost evenly, according to Amdahl's Law. I feel that one of the main factors that affects the time taken would be the variable, such as the size of the hashmap and size of a node. By right, the x-axis should be the size of hashmap in my opinion. As the size grows, the time of execution would be easier to compare. However, in the maptest.o, I believe that those variable during the initialization are fixed, so the only changed variable I can think of is the thread number.

One way to improve the performance I can think of is the addition of register keyword. For the functions that have struct node variable, I placed the register in front. The overall time of execution do improve a bit. However, after did some research, I realized this is not the most pleasant way to do so. [GeeksforGeeks, 2019] This is because I cannot really control the compiler would really put all the nodes into the register.

Another factor that affects my performance would be the overhead in the functions I implemented. Even though all of them are thread-safe, the tasks for each node are relatively heavy. At the same time, despite the number of locks is very little, the tasks for the threads are relatively large, which consist of conditions of checking and iterations. For example, when inserting, the tasks for the threads are about initializing a node, check if the allocation of memory has succeeded, if the index of the key is unique and the iteration of the placement of the node if the index is duplicated in the hashmap. In addition, using read or write lock as an example, I may experience resource starvation or resource contention. [Bill Burns, 2020] This is because if some threads are utilizing with same key, that would be a waste of time for the checking of the validity of the key.

On top of these, if I allocate lesser tasks for the threads and put more locks, it may affect the performance too. Data race may occur if the position of locks is not adequately placed. Also, threads are queued when a lock occurs. As the threads are assigned tasks, in a function, which are decomposed into smaller parts, the overall time taken for achieving a goal may be longer than expected, due to the increased number of locks and unlocks in a function.

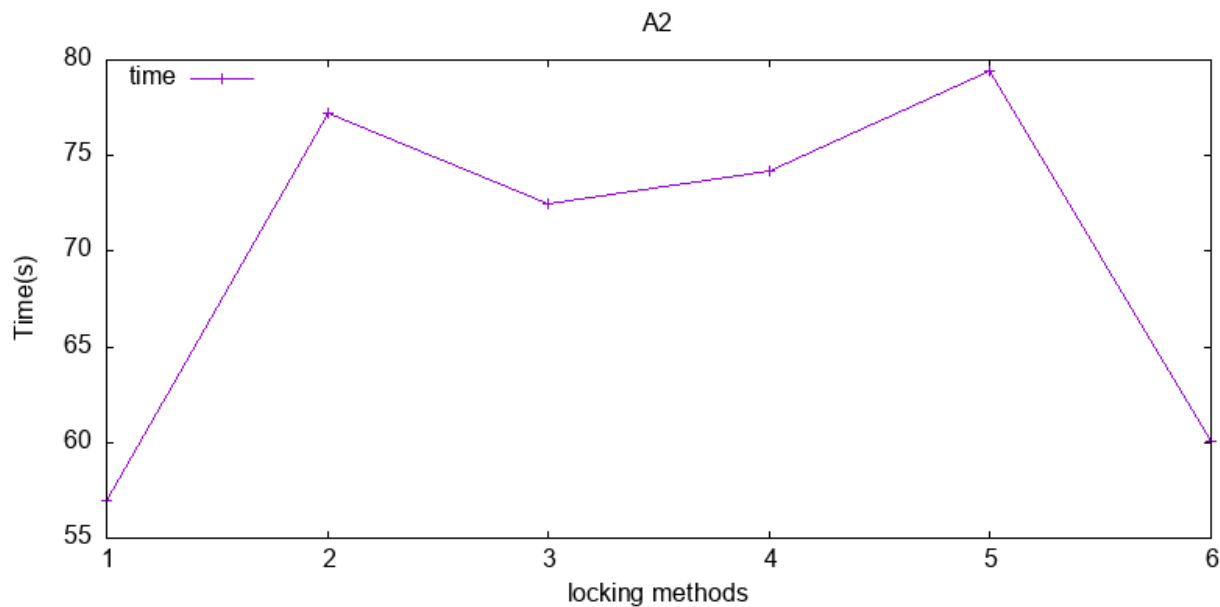
In general, since the design and flow of the idea is adequate for each function, I believe this is another reason that the time of execution is shorter. This is because I ensure

there is no unnecessary delay in between. This had been further elaborated above, in the heading, implementation.

### Testing methodology

- Firstly, I created my own tests in `hashmap_test.c`. Several functions are utilized in the `hashmap.c`. They are basically about the correctness of utilizing the (core) functions that I have implemented. For single thread, for example, the test on insertion, deleting, get and update. Some of the functions of the testing are a combination of the features. For concurrency, I created a number of 2 threads, with the input variables for `hashmap_init()`. I have tested them, such as the current size of the hashmap, whether a node is inserted or removed and whether is node can be retrieved or updated.
- The tests with the benchmark provided by the teacher haven been talked previously / above. And now, I shall talk about the reason why I chose `rwlock`.

### Optimistic locking / To improve performance:



- 1: coarse-grained of `rwlock` (used in `hashmap.c`)
- 2: fine-grained of `rwlock`
- 3: atomic operation

- 4: mutex
- 5: semaphore
- 6: spinlock

As I do not know how to put letters in the gnuplot, I replaced the locking methods with integer.

I used maptest.o to generate this graph, with the default (2) thread number. The evidence of the timings is in Fig.15 to 19. The only variable I can changed is the number of threads, and from the previous graph, I feel that the change of the thread number does not really affect the result. Hence, I simply compared and timed them with 2 threads only, via the maptest.o.

To be honest, I should create a new benchmark, and replace x-axis as the size of the input nodes or hashmap. As the number increases, the comparison would be more appealing to see.

To my surprise, the time taken for spinlock is nearly same as the 1. The reason of not using it is because, after reading from Wikipedia, <https://en.wikipedia.org/wiki/Spinlock>, I learnt that spinlock is only suitable to use when the duration of locking is short. However, my design of the locks has disobeyed this point. This is because the tasks of each thread are quite large. In simple words, the locks in the functions start at beginning and end before the return statement, and nothing else.

But, all in all, with the aid of maptest.o which is provided by the teacher, the initial locking method I have proved to be the most efficient among them.

For an extra information, I have implemented pthread\_barrier too. However, the program will be stuck forever. I guess is due to the deadlocking, or resource contention which would result a false sharing. And I cannot really figure out how to lock specific type of locks with barrier.

## **Safety Guarantees**

As mentioned earlier, the usage of locks and unlocks in the functions, as well as the flow of design, guaranteed me a thread safe application. (In hashmap.c from Implementation)

For example, as removed and inserted are rapidly performed, I have already ensured that each thread is mutually exclusive and perform own tasks without interfering the other. Each node is allocated with own memory. When iteration of finding the next node is occurring, I ensure that no extra memory is wasted when doing this. In addition, there are conditions to ensure the threads can exit the function safely.

All the other relevant details / explanations are included previously / above.

## Extension (Memory Optimisation)

- Even though I used functions in qalloc.c, I did not change/implement those functions. Due to lack of abilities, I tried but I will fail the testcases for the maptest.o. I get my learning resources from here, <http://tharikasblogs.blogspot.com/p/how-to-write-your-own-malloc-and-free.html>.
- Although I did not do this part, I did my research and learnt the benefits of creating own memory allocator. From this, <https://stackoverflow.com/questions/56167252/why-design-custom-memory-manager>, one of the comments is really insightful. For example, own memory allocate would always force the memory to be fully cleaned up. Since the codes have used the memory allocated to them, memory leaks will never be happening if the own memory allocator wants to free itself.
- Sometimes, I feel that an own memory allocator would have a relatively higher debug ability. This is because the allocated memory can be retrieved from the own memory allocator easier, if structs are created beforehand.

## Appendix:



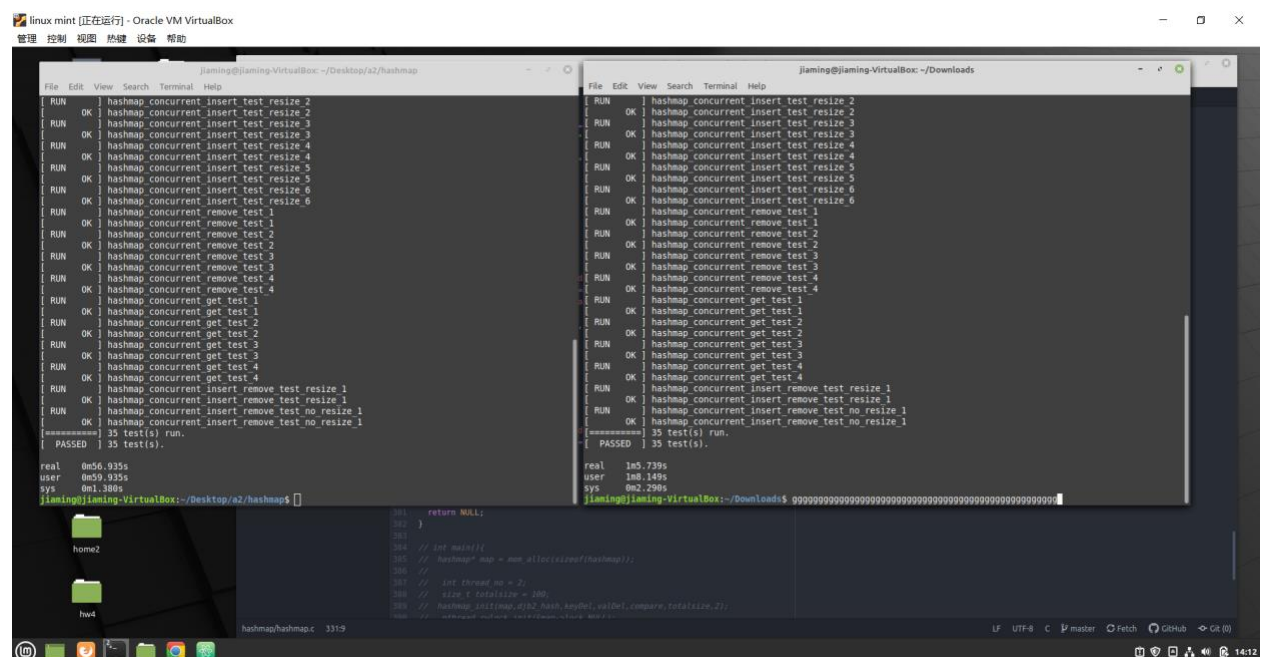
Inter-Cluster Thread-to-Core Mapping and DVFS on Heterogeneous Multi-Cores, 26 September 2017, <https://ieeexplore.ieee.org/document/8051086>

Understanding “register” keyword in C, 21 Aug, 2019 ,  
<https://www.geeksforgeeks.org/understanding-register-keyword/>

What Is Multithreading: A Guide to Multithreaded Applications by Bill Burns, September 4 2020, <https://totalview.io/blog/multithreading-multithreaded-applications>

Program to create Deadlock Using C in Linux, Dextutor, Baljit Singh Saini,  
<https://dextutor.com/program-to-create-deadlock-using-c-in-linux/>

Figures:



```
linux mint [正在运行] - Oracle VM VirtualBox
管理 控制 视图 热键 设备 帮助

jiaming@jiaming-VirtualBox: ~/Desktop/a2/hashmap
File Edit View Search Terminal Help
RUN OK hashmap_concurrent_insert_test_resize 2
RUN OK hashmap_concurrent_insert_test_resize 3
RUN OK hashmap_concurrent_insert_test_resize 4
RUN OK hashmap_concurrent_insert_test_resize 5
RUN OK hashmap_concurrent_insert_test_resize 6
RUN OK hashmap_concurrent_remove_test 1
RUN OK hashmap_concurrent_remove_test 2
RUN OK hashmap_concurrent_remove_test 3
RUN OK hashmap_concurrent_remove_test 4
RUN OK hashmap_concurrent_remove_test 4
RUN OK hashmap_concurrent_get_test 1
RUN OK hashmap_concurrent_get_test 2
RUN OK hashmap_concurrent_get_test 2
RUN OK hashmap_concurrent_get_test 3
RUN OK hashmap_concurrent_get_test 3
RUN OK hashmap_concurrent_get_test 4
RUN OK hashmap_concurrent_insert_remove_test_resize 1
RUN OK hashmap_concurrent_insert_remove_test_resize 1
RUN OK hashmap_concurrent_insert_remove_test_no_resize 1
===== 35 test(s) run.
PASSED 35 test(s).

real 0m56.935s
user 0m59.935s
sys 0m1.388s
jiaming@jiaming-VirtualBox:~/Desktop/a2/hashmap$

home2
hwt

hashmap/hashmap.c 331:9

201 return NULL;
202 }
203 // int main()
204 // hashmap_map = new hashmap(hashmap);
205 //
206 // int thread_no = 2;
207 // size_t total_size = 200;
208 // hashmap_init(hash_map, hash_key, valid, compare, total_size);
209 // hashmap_concurrent_insert_remove_test_resize 1;

jiaming@jiaming-VirtualBox: ~/Downloads
File Edit View Search Terminal Help
RUN OK hashmap_concurrent_insert_test_resize 2
RUN OK hashmap_concurrent_insert_test_resize 3
RUN OK hashmap_concurrent_insert_test_resize 4
RUN OK hashmap_concurrent_insert_test_resize 5
RUN OK hashmap_concurrent_insert_test_resize 6
RUN OK hashmap_concurrent_remove_test 1
RUN OK hashmap_concurrent_remove_test 2
RUN OK hashmap_concurrent_remove_test 3
RUN OK hashmap_concurrent_remove_test 4
RUN OK hashmap_concurrent_get_test 1
RUN OK hashmap_concurrent_get_test 2
RUN OK hashmap_concurrent_get_test 3
RUN OK hashmap_concurrent_get_test 4
RUN OK hashmap_concurrent_insert_remove_test_resize 1
RUN OK hashmap_concurrent_insert_remove_test_resize 1
RUN OK hashmap_concurrent_insert_remove_test_no_resize 1
===== 35 test(s) run.
PASSED 35 test(s).

real 1m5.739s
user 1m8.149s
sys 0m2.290s
jiaming@jiaming-VirtualBox:~/Downloads$
```

Fig.1. 2 threads.

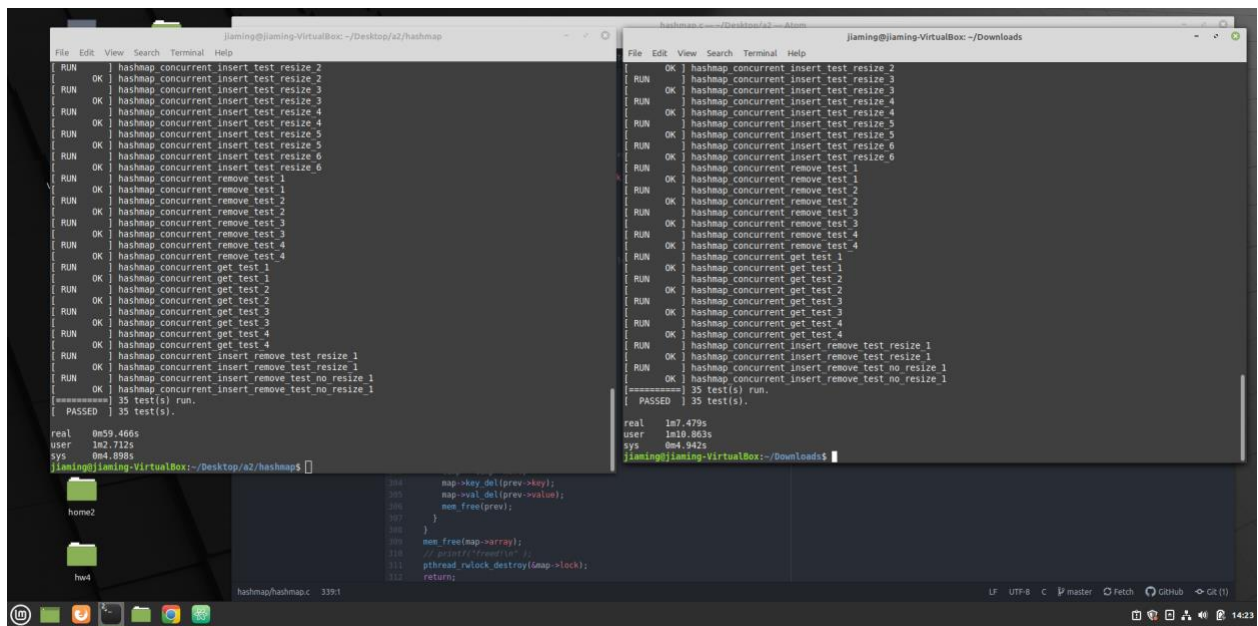


Fig.2. 3 threads

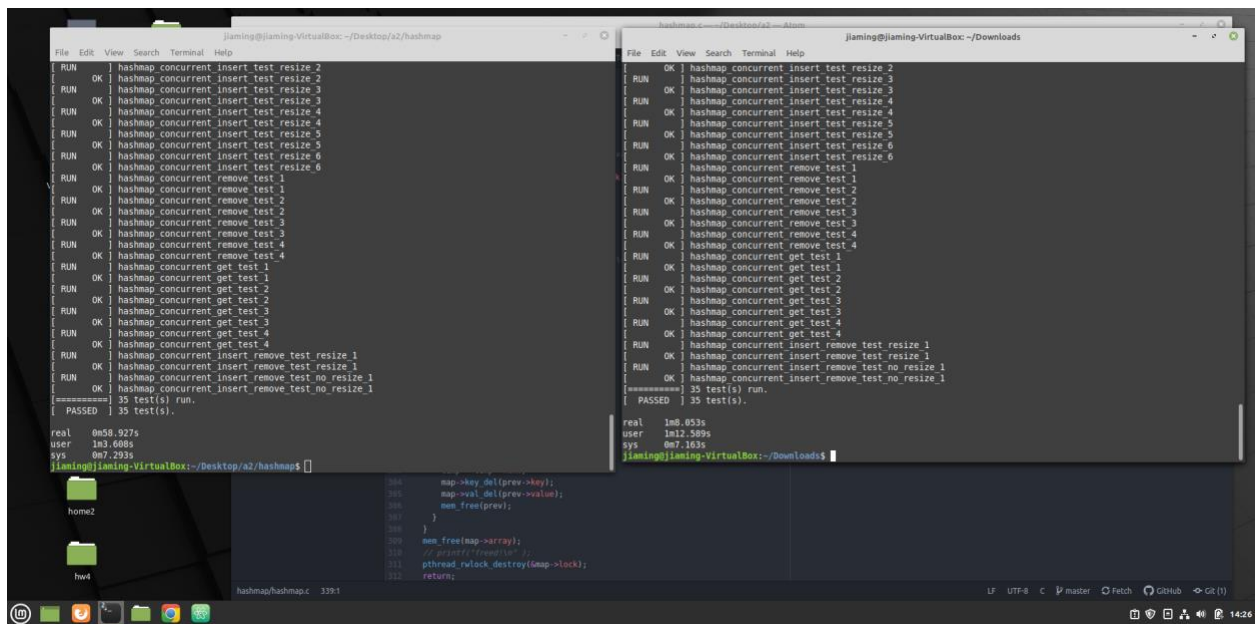


Fig.3. 4 threads.

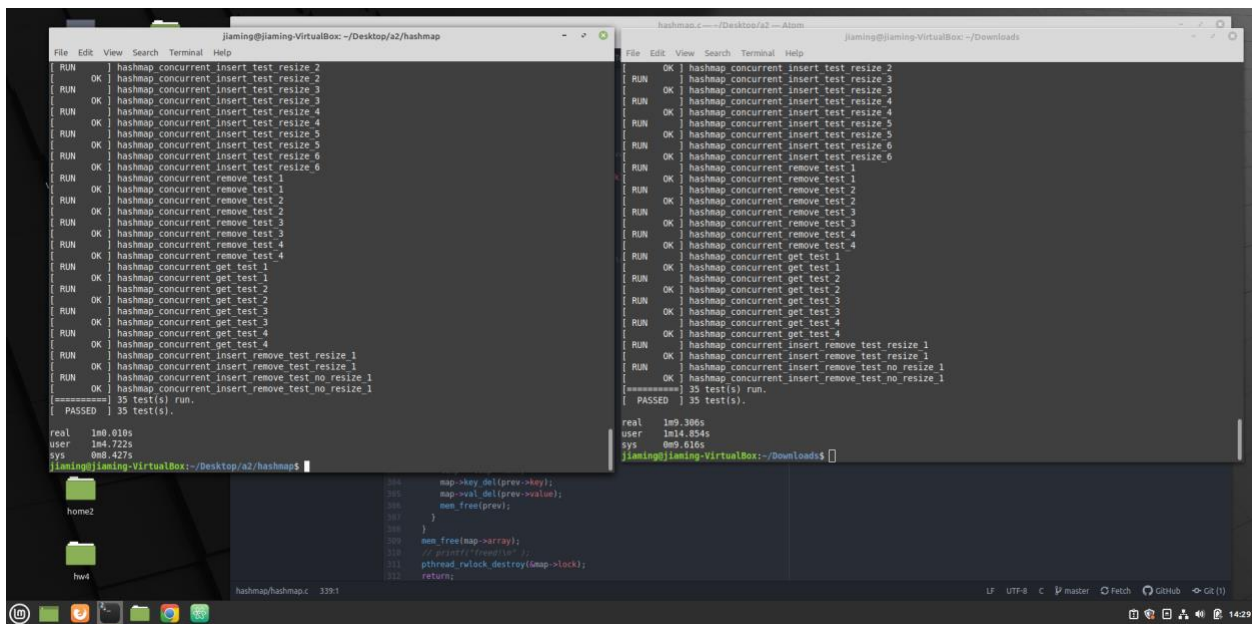


Fig.4. 5 threads

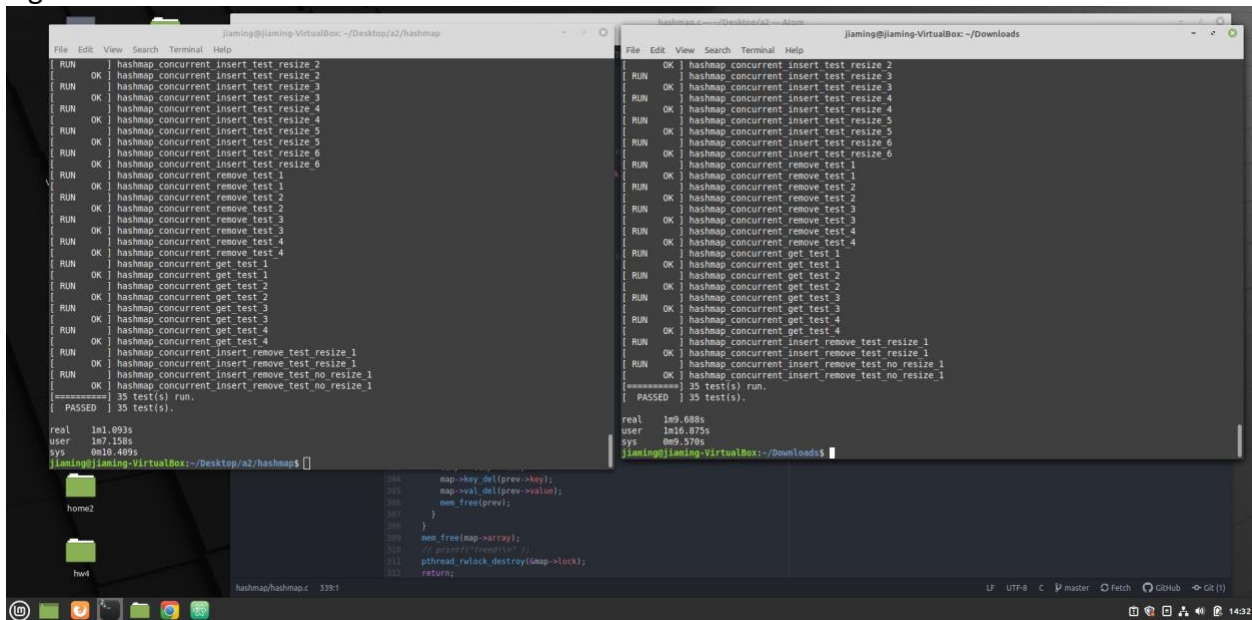


Fig.5. 6 threads

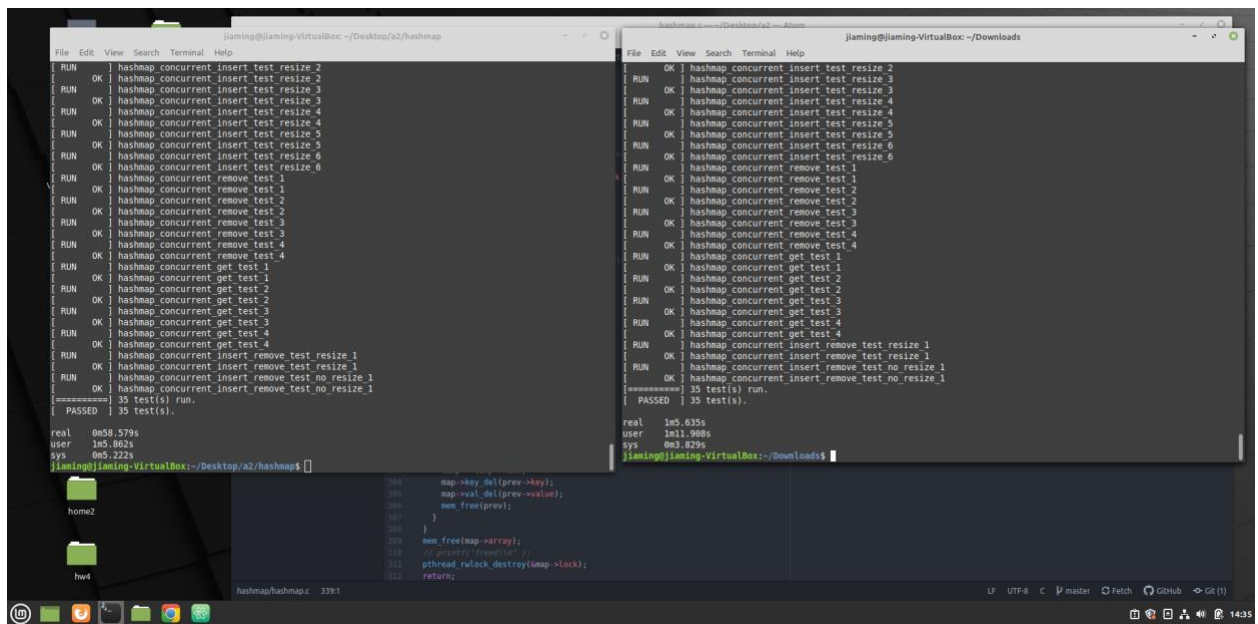


Fig.6. 7 threads

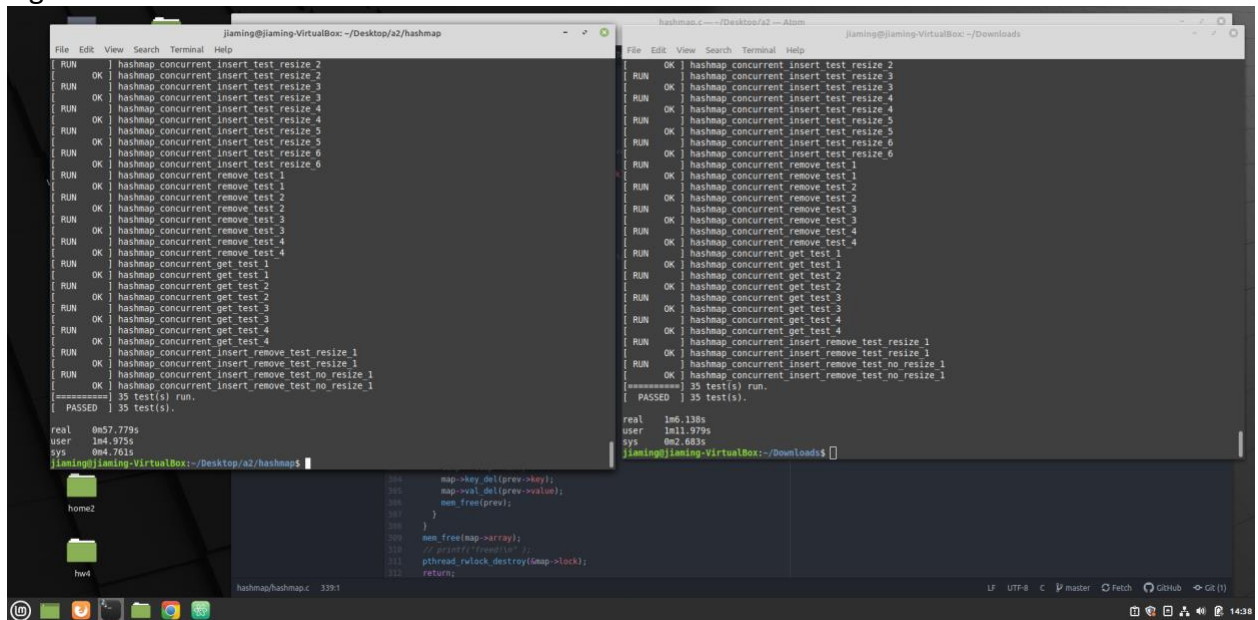


Fig.7. 8 threads



```
File Edit View Search Terminal Help
RUN OK | hashmap_concurrent insert test resize 2
RUN OK | hashmap_concurrent insert test resize 2
RUN OK | hashmap_concurrent insert test resize 3
RUN OK | hashmap_concurrent insert test resize 3
RUN OK | hashmap_concurrent insert test resize 4
RUN OK | hashmap_concurrent insert test resize 4
RUN OK | hashmap_concurrent insert test resize 5
RUN OK | hashmap_concurrent insert test resize 5
RUN OK | hashmap_concurrent insert test resize 6
RUN OK | hashmap_concurrent insert test resize 6
RUN OK | hashmap_concurrent remove test 1
RUN OK | hashmap_concurrent remove test 2
RUN OK | hashmap_concurrent remove test 2
RUN OK | hashmap_concurrent remove test 3
RUN OK | hashmap_concurrent remove test 4
RUN OK | hashmap_concurrent remove test 4
RUN OK | hashmap_concurrent get test 1
RUN OK | hashmap_concurrent get test 2
RUN OK | hashmap_concurrent get test 2
RUN OK | hashmap_concurrent get test 3
RUN OK | hashmap_concurrent get test 3
RUN OK | hashmap_concurrent get test 4
RUN OK | hashmap_concurrent get test 4
RUN OK | hashmap_concurrent insert_remove test resize 1
RUN OK | hashmap_concurrent insert_remove test resize 1
RUN OK | hashmap_concurrent insert_remove test_no_resize 1
RUN OK | hashmap_concurrent insert_remove test_no_resize 1
===== 35 test(s) run.
PASSED | 35 test(s).

real 0m58.396s
user 1m0.809s
sys 0m4.944s
jiaming@jiaming-VirtualBox:~/Desktop/x2/hashmap$
```

```
204 map->key.del(prev->key);
205 map->val.del(prev->val);
206 mem_free(prev);
207 }
208 }
209 mem_free(map->array);
210 if (pthread_rwlock_t)
211 pthread_rwlock_destroy(&map->lock);
212 return;
```

hashmap/hashmap.c 339:1

Fig.8. 9 threads

```
File Edit View Search Terminal Help
RUN OK | hashmap_concurrent insert test resize 2
RUN OK | hashmap_concurrent insert test resize 2
RUN OK | hashmap_concurrent insert test resize 3
RUN OK | hashmap_concurrent insert test resize 3
RUN OK | hashmap_concurrent insert test resize 4
RUN OK | hashmap_concurrent insert test resize 4
RUN OK | hashmap_concurrent insert test resize 5
RUN OK | hashmap_concurrent insert test resize 5
RUN OK | hashmap_concurrent insert test resize 6
RUN OK | hashmap_concurrent insert test resize 6
RUN OK | hashmap_concurrent remove test 1
RUN OK | hashmap_concurrent remove test 1
RUN OK | hashmap_concurrent remove test 2
RUN OK | hashmap_concurrent remove test 2
RUN OK | hashmap_concurrent remove test 3
RUN OK | hashmap_concurrent remove test 3
RUN OK | hashmap_concurrent remove test 4
RUN OK | hashmap_concurrent remove test 4
RUN OK | hashmap_concurrent get test 1
RUN OK | hashmap_concurrent get test 1
RUN OK | hashmap_concurrent get test 2
RUN OK | hashmap_concurrent get test 2
RUN OK | hashmap_concurrent get test 3
RUN OK | hashmap_concurrent get test 3
RUN OK | hashmap_concurrent get test 4
RUN OK | hashmap_concurrent get test 4
RUN OK | hashmap_concurrent insert_remove test resize 1
RUN OK | hashmap_concurrent insert_remove test resize 1
RUN OK | hashmap_concurrent insert_remove test_no_resize 1
RUN OK | hashmap_concurrent insert_remove test_no_resize 1
===== 35 test(s) run.
PASSED | 35 test(s).

real 0m59.572s
user 1m14.395s
sys 0m3.222s
jiaming@jiaming-VirtualBox:~/Desktop/x2/hashmap$
```

```
204 map->key.del(prev->key);
205 map->val.del(prev->val);
206 mem_free(prev);
207 }
208 }
209 mem_free(map->array);
210 if (pthread_rwlock_t)
211 pthread_rwlock_destroy(&map->lock);
212 return;
```

hashmap/hashmap.c 339:1

Fig.9. 10 threads

```
File Edit View Search Terminal Help
RUN OK | hashmap_concurrent insert test resize 2
RUN OK | hashmap_concurrent insert test resize 2
RUN OK | hashmap_concurrent insert test resize 3
RUN OK | hashmap_concurrent insert test resize 3
RUN OK | hashmap_concurrent insert test resize 4
RUN OK | hashmap_concurrent insert test resize 4
RUN OK | hashmap_concurrent insert test resize 5
RUN OK | hashmap_concurrent insert test resize 5
RUN OK | hashmap_concurrent insert test resize 6
RUN OK | hashmap_concurrent insert test resize 6
RUN OK | hashmap_concurrent remove test 1
RUN OK | hashmap_concurrent remove test 2
RUN OK | hashmap_concurrent remove test 2
RUN OK | hashmap_concurrent remove test 3
RUN OK | hashmap_concurrent remove test 4
RUN OK | hashmap_concurrent remove test 4
RUN OK | hashmap_concurrent get test 1
RUN OK | hashmap_concurrent get test 2
RUN OK | hashmap_concurrent get test 3
RUN OK | hashmap_concurrent get test 3
RUN OK | hashmap_concurrent get test 4
RUN OK | hashmap_concurrent get test 4
RUN OK | hashmap_concurrent insert_remove test resize 1
RUN OK | hashmap_concurrent insert_remove test resize 1
RUN OK | hashmap_concurrent insert_remove test_no_resize 1
RUN OK | hashmap_concurrent insert_remove test_no_resize 1
[=====] 35 test(s) run.
PASSED | 35 test(s).

real 1m0.139s
user 1m25.126s
sys 0m0.845s
jiaming@jiaming-VirtualBox: ~/Desktop/a2/hashmap

204 map_key_del(prev->key);
205 map_val_del(prev->val);
206 non_free(prev);
207 }
208 }
209 non_free(map->array);
210 // printf("found\n");
211 pthread_rwlock_destroy(&map->lock);
212 return;
```

Fig.10. 11 threads

```
File Edit View Search Terminal Help
RUN OK | hashmap_concurrent insert test resize 2
RUN OK | hashmap_concurrent insert test resize 2
RUN OK | hashmap_concurrent insert test resize 3
RUN OK | hashmap_concurrent insert test resize 3
RUN OK | hashmap_concurrent insert test resize 4
RUN OK | hashmap_concurrent insert test resize 4
RUN OK | hashmap_concurrent insert test resize 5
RUN OK | hashmap_concurrent insert test resize 5
RUN OK | hashmap_concurrent insert test resize 6
RUN OK | hashmap_concurrent insert test resize 6
RUN OK | hashmap_concurrent remove test 1
RUN OK | hashmap_concurrent remove test 1
RUN OK | hashmap_concurrent remove test 2
RUN OK | hashmap_concurrent remove test 2
RUN OK | hashmap_concurrent remove test 3
RUN OK | hashmap_concurrent remove test 3
RUN OK | hashmap_concurrent remove test 4
RUN OK | hashmap_concurrent remove test 4
RUN OK | hashmap_concurrent get test 1
RUN OK | hashmap_concurrent get test 1
RUN OK | hashmap_concurrent get test 2
RUN OK | hashmap_concurrent get test 2
RUN OK | hashmap_concurrent get test 3
RUN OK | hashmap_concurrent get test 3
RUN OK | hashmap_concurrent get test 4
RUN OK | hashmap_concurrent get test 4
RUN OK | hashmap_concurrent insert_remove test resize 1
RUN OK | hashmap_concurrent insert_remove test resize 1
RUN OK | hashmap_concurrent insert_remove test_no_resize 1
RUN OK | hashmap_concurrent insert_remove test_no_resize 1
[=====] 35 test(s) run.
PASSED | 35 test(s).

real 0m59.215s
user 1m14.927s
sys 0m2.968s
jiaming@jiaming-VirtualBox: ~/Desktop/a2/hashmap

204 map_key_del(prev->key);
205 map_val_del(prev->val);
206 non_free(prev);
207 }
208 }
209 non_free(map->array);
210 // printf("found\n");
211 pthread_rwlock_destroy(&map->lock);
212 return;
```

Fig.11. 12 threads

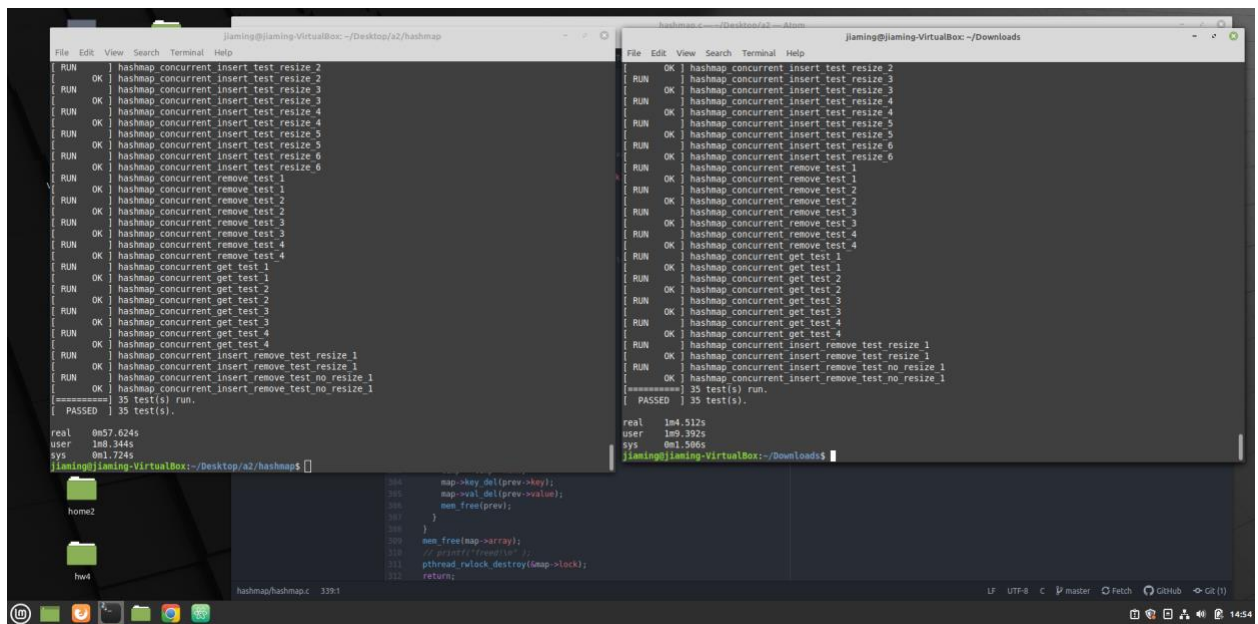


Fig.12. 20 threads

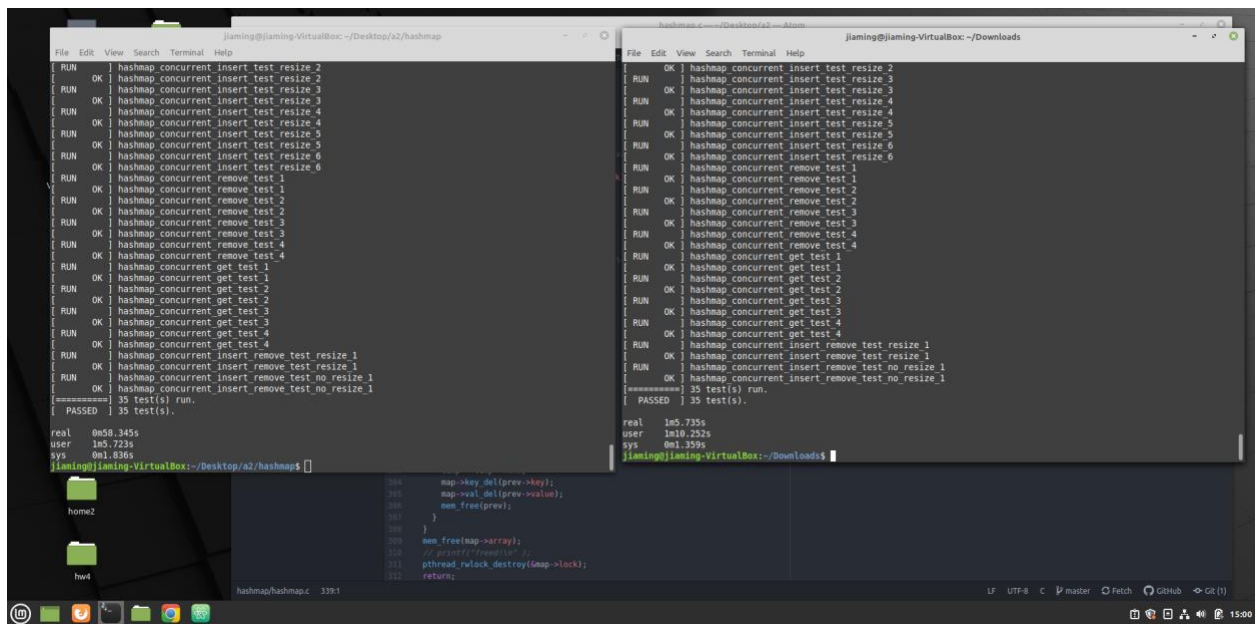


Fig.13. 100 threads



 **data.txt**

```
#Threads from_teacher my_own
2 65.739 56.935
3 67.479 59.466
4 68.053 58.927
5 69.306 60.010
6 69.688 61.093
7 65.635 58.579
8 66.138 57.779
9 65.899 58.396
10 65.623 59.572
11 64.507 60.139
12 66.054 59.215
20 64.512 57.624
100 65.735 58.345|
```

Fig.14. The data for comparing the time taken.



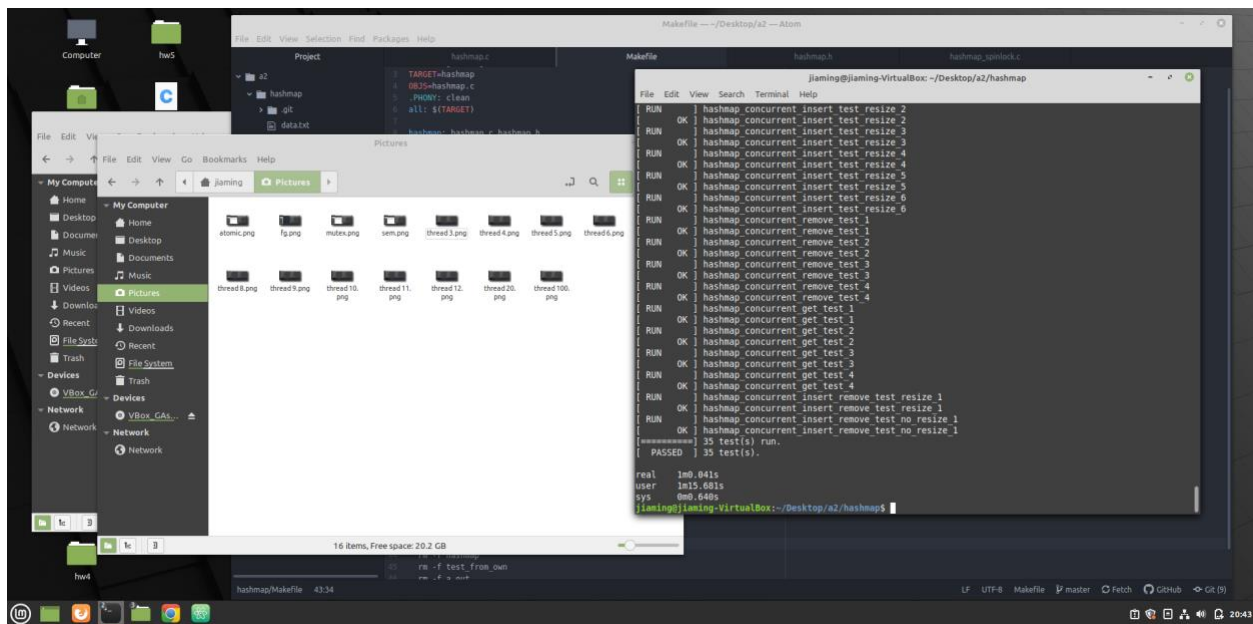


Fig.15. when using spinlock

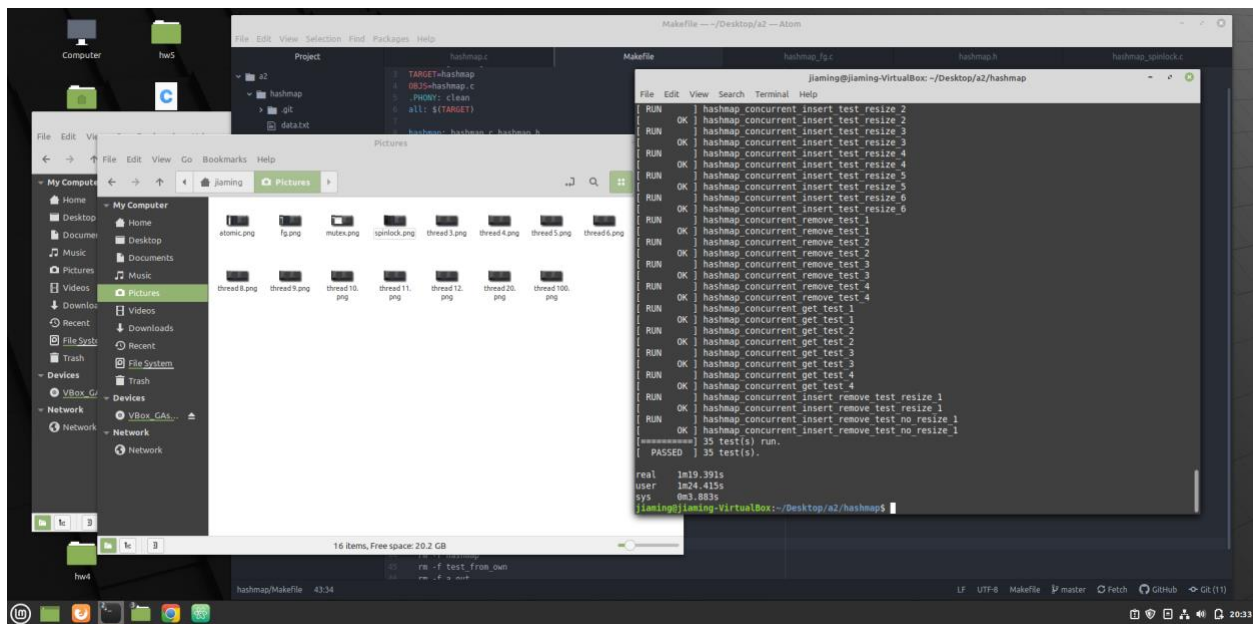


Fig.16. when using semaphore

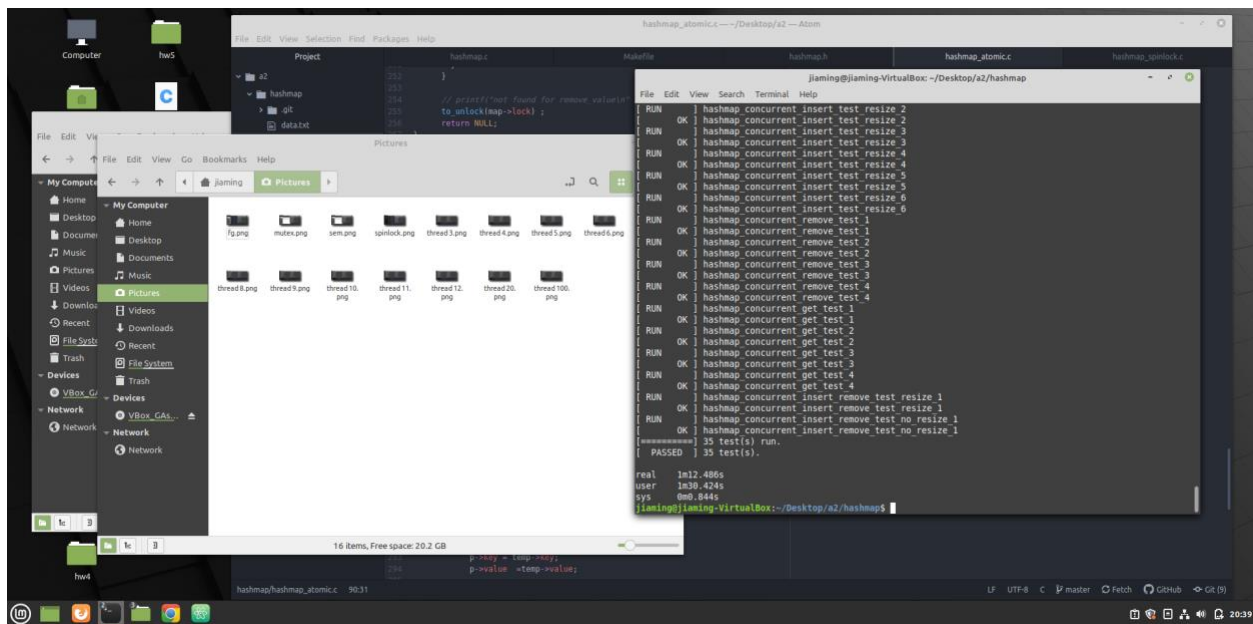


Fig.17. when using atomic operation

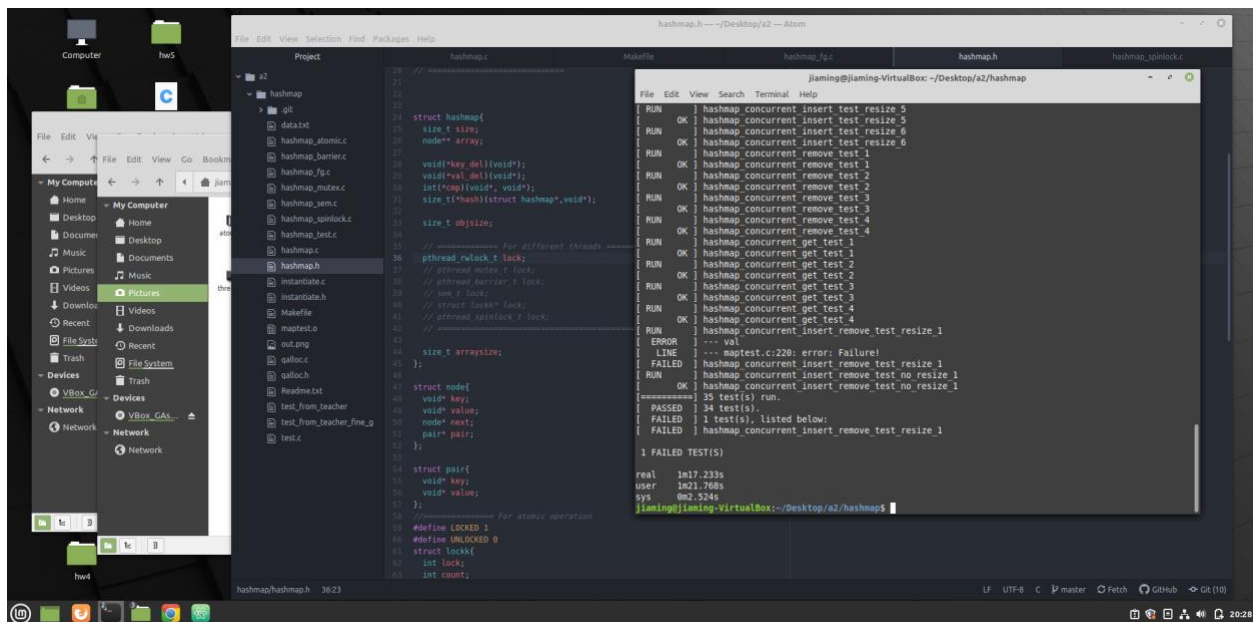


Fig.18. fine-grained version of rwlock

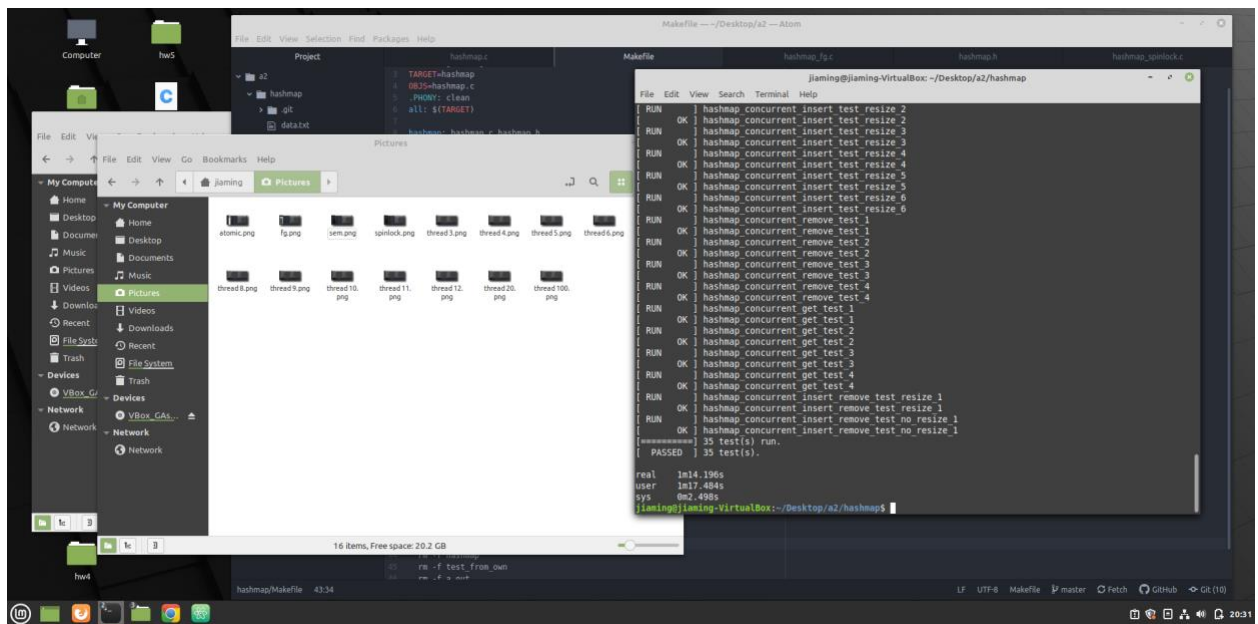


Fig.19. when using mutex