**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:

Chart, line chart

Description automatically generated

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 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:**

Chart, line chart

Description automatically generated

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:**

A picture containing text, electronics, screenshot, computer

Description automatically generated

Fig.1. 2 threads.

A screenshot of a computer

Description automatically generated with medium confidence

Fig.2. 3 threads

A screenshot of a computer

Description automatically generated with medium confidence

Fig.3. 4 threads.

A screenshot of a computer

Description automatically generated with medium confidence

Fig.4. 5 threads

A screenshot of a computer

Description automatically generated with medium confidence

Fig.5. 6 threads

A screenshot of a computer

Description automatically generated with medium confidence

Fig.6. 7 threads

A screenshot of a computer

Description automatically generated with medium confidence

Fig.7. 8 threads

A screenshot of a computer

Description automatically generated with medium confidence

Fig.8. 9 threads

A screenshot of a computer

Description automatically generated with medium confidence

Fig.9. 10 threads

A screenshot of a computer

Description automatically generated with medium confidence

Fig.10. 11 threads

A screenshot of a computer

Description automatically generated with medium confidence

Fig.11. 12 threads

A screenshot of a computer

Description automatically generated with medium confidence

Fig.12. 20 threads

A screenshot of a computer

Description automatically generated with medium confidence

Fig.13. 100 threads

Graphical user interface, text, application

Description automatically generated

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

A screenshot of a computer

Description automatically generated with medium confidence

Fig.15. when using spinlock

A screenshot of a computer

Description automatically generated with medium confidence

Fig.16. when using semaphore

A screenshot of a computer

Description automatically generated with medium confidence

Fig.17. when using atomic operation

A screenshot of a computer

Description automatically generated with medium confidence

Fig.18. fine-grained version of rwlock

A screenshot of a computer

Description automatically generated with medium confidence

Fig.19. when using mutex