Final Project Write-Up

Concurrent Tree

1. A description of your algorithm

I have implemented a key-value store implemented as a Binary Search Tree. Each node of this tree has a structure which consists of the key, value, pointer to a left node structure variable, pointer to the right node structure variable and a lock. We also have a pointer to a global structure variable which acts as the main root of the tree. Additionally, we also have a global lock which is used if and when the tree is empty. The locks used here is the "pthread_mutex_t" lock. Hand-over-hand locking is used to avoid data races.

In the reader-writer lock implementation of this tree, the locks used is the "pthread_rwlock_t" lock.

Put
The put function accepts a root node, a parent node (which will always be NULL when called from a non-recursive context), the key and the value. If the parent node is NULL, the global lock is locked. If the tree does not exist, the root node is created, the global lock is unlocked and the function returns. Otherwise, the main root node is locked, and then the tree lock is unlocked. Now the key value to be inserted is compared with key of the current root node. If it is lesser than the key of the current root node, it is checked if the left node exists. If it does not, a node is created, the lock on the current root node is unlocked and the function returns. Otherwise, if the left node exists, this left node is locked, the current root is unlocked, and the same function is called in a recursive manner passing the left node as the root and current root node as the parent node.
Similarly, if the key is greater than the key of current root node, the above commands are implemented with respect to the right node from the current root node.
If the key to be inserted is equal to the key of the current root node, the value is corresponding to that key is overwritten, the root node is unlocked and the function returns.
Therefore, the algorithm proceeds in a similar manner calling the "put" function in a recursive manner until a left/right NULL pointer is encountered or if the key matches the key of the current root node.

In case of the reader-writer lock implementation of this tree, all the locks used in the above implementation are replaced by a writer lock.

Get
The get function accepts a root node, a parent node (which will always be NULL when called from a non-recursive context) and the key. This function has a similar flow as compared to the put function. The only change is that if the main root node, the left node or the right node at any instance is NULL, a message is printed to indicate that the key does not exist in the tree (instead of creating a new node) after which the current root node is unlocked and the function returns. Additionally, if the key matches the key of the current root node, the node is found, and the corresponding key and value is printed.

In case of the reader-writer lock implementation of this tree, all the locks used in the above implementation are replaced by a reader lock.

Range Queries

This function accepts a root node, a parent node (which will always be NULL when called from a non-recursive context), the low key, the high key, and the TID of the thread executing this function for printing purposes. The algorithm works by making use of inorder traversal. Therefore, the algorithm traverses the tree and checks if the key falls within the range in a sorted order. If the low key is lesser than the key of the root node, then the left subtree is recursively called. If the root's key falls within the range, then the key and value of the root node is printed. If the high key is greater than the key of the root node, then the right subtree is recursively called. Hand-over-hand locking is used by locking the next node before releasing the current node.

2. Experimental results

The code was tested based on a combination of the following three criteria.

- Lock type – "pthread_mutex_lock" vs Reader-Writer locks
- Contention – high contention vs low contention
- Number of threads
- Number of iterations

Based on the number of threads, 1/3rd of threads are involved in the put operation, 1/3rd in get and the rest in the range queries operation. Each thread involved with the get and put operation implement the get and put operation equal to the number of iterations. The number of operations for range queries was equal to one-third of the number of total threads. For example, if the user inputs 12 threads with 5 iterations, we will have 20 put operations, 20 get operations and 4 range query operations.

A different random value is generated for each iteration. In the low contention scenario, the key values can range from 0 to 999,999. This ensures that the keys are spread out and there is less contention among different threads. In the high contention scenario, the thread with id 1 first constructs a skewed tree with number of elements equal to 1/3rd of the total number of threads multiplied by number of iterations. These operations are not included in the timing measurement. Once this tree has been constructed, the put and get operations can only work with key values ranging from 0-4. This ensures that all the operations access a small range of key resulting in a higher chance of threads conflicting for the same key.

Note: The following results take all the print statements into consideration. Therefore, the following timings do not match those values which result when running the unit test bash script or if command is run without specifying the print argument (see execution instructions for more details).

"pthread_mutex_lock"

Number of Threads	Number of iterations	Total number of get/put operations	High Contention	Low Contention	Diff (high contention – low contention)
12	100	400	.008188 s	0.008471 s	-0.000283 s
12	10000	40000	5.335862 s	0.199742 s	5.13612 s

24	10000	80000	35.683229 s	0.904462 s	34.778767 s
36	10000	120000	88.751777 s	1.162768 s	87.589009 s
48	10000	480000	168.651478 s	1.979585 s	166.671893 s

Table 1

<u>Reader-Writer locks</u>

Number of Threads	Number of iterations	Total number of get/put operations	High Contention	Low Contention	Diff (high contention – low contention)
12	100	400	.005433 s	0.00280 s	0.00263 s
12	10000	40000	23.502044 s	0.167138 s	23.3349 s
24	10000	80000	71.545331 s	0.496125 s	71.0492 s
36	10000	120000	149.475260 s	0.809013 s	148.6662 s
48	10000	480000	241.424249 s	1.224759 s	240.1994 s

Table 2

3. Analysis of results using perf as necessary to support explanations

The first table compares the time taken to execute the 3 operations using normal pthread locks for high contention and low contention cases for different number of threads and iterations. On observing the table, we notice that the high contention case takes more time than the low contention case, as expected, when the number of operations is much greater. However, when the number of operations is small, the there is not much difference between the high contention and low contention cases. Depending on that particular test run, either the high contention case takes more time or the low contention case. But either way, the difference is small. A significant difference is observed when the number of operations is greater than 5000. We really notice the impact of high contention on the timing with higher number of operations. This data is backed by the number of CPU cycles taken for different number of operations and the page-faults which is highlighted in the table below.

<u>CPU Cycles</u>

Number of Threads	Number of iterations	Total number of get/put operations	High Contention	Low Contention
12	100	400	49,863,050	21,866,366
12	10000	40000	23.502044 s	2,035,181,829
48	10000	480000	7,884,378,207,687	8,239,565,502

Table 3

Page-Faults

Number of Threads	Number of iterations	Total number of get/put operations	High Contention	Low Contention
12	100	400	128	102
12	10000	40000	3680	581
48	10000	480000	48,299	871

Table 4

We observe that the CPU cycles are significantly more in the high contention case as compared to the low contention case which explains the greater time taken by the higher contention case.

The second table compares the time taken to execute the 3 operations using reader-writer locks for high contention and low contention cases for different number of threads and iterations. We similarly observe that the high contention case takes more time than the low contention case. Additionally, if we compare table 1 and table 2, we observe that for the low contention case, reader writer locks takes lesser time as compared to normal pthread lock, which is expected as there can be multiple reads as compared to only a single read in the case of normal pthread locks. We, however, observe a peculiar behaviour when comparing the time taken between normal pthread locks and reader-writer locks in the high contention case, where the reader-writer locks take more time than normal pthread mutex locks. This is probably due to the manner in which a skewed tree is pre-constructed in the high contention case.

The number of CPU cycles and the page faults statistics for reader-writer locks is similar for the case with normal pthread locks as highlighted in table 3 and table 4.

Therefore, we observe that low contention scenarios give better performance. When using reader-writer locks, we see an improved performance only in low contention cases. In the high contention cases, the performance reduces when using reader-writer locks.

4. Description of code organisation

- The number of threads, lock type and test type are accepted as arguments from the user.
- Mutex, barriers and threads are initialized.
- Threads are launched.
- If the TID is one and test type is high contention, the initial tree is constructed. TID one also notes the time.
- If the TID is lesser than one thirds of the number of threads, each thread is looped based on the number of iterations and a random key is inserted into the tree.
- If the TID is greater than one thirds of the number of threads and lesser than two thirds of the number of threads, each thread is looped based on the number of iterations and a random key is searched from the tree.
- If the TID is greater than two thirds of the number of threads, each thread performs a range query.

- A barrier is used to ensure all threads are completed before the TID one takes note of the end time.
- The mutex, barriers and threads are freed from memory.
- An inorder traversal prints out all the elements in sorted order.
- The number of elements in the tree is compared against the number expected to declare success or failure of put operation
- It is checked if all the get operations are executed successfully and accordingly, the result is printed.
- The time taken to execute the three operations is printed.

5. Description of every file submitted

main.c – Contains the main function, This file is responsible for taking in the command line arguments, creating threads, calling the put, get and range query function from the thread function and determining the time needed to carry out the task.
fine_grained_lock_BST.c – This file contains the put, get, range query and inorder traversal functions for hand-over-hand locking using normal pthread mutex.

fine_grained_lock_BST.h – This file contains the tree structure, the global lock, the main root node and function declarations for hand-over-hand locking using normal pthread mutex.

rw_lock_BST.c – This file contains the put, get, range query and inorder traversal functions for hand-over-hand locking using reader-writer locks.

rw_lock_BST.h – This file contains the tree structure, the global lock, the main root node and function declarations for hand-over-hand locking using reader-writer locks.

Makefile – File to build and generate the executable.

config.h – Contains variables needed by both types of locks implementations

config.c – Contains function to print execution instructions.

6. Compilation instructions

Just enter “make” to compile.

7. Execution Instructions

The executable is “concurrent_tree”. The command line takes in 5 arguments.

- --test=high – testing high contention scenario (This case takes time depending on the number of threads and iterations)
  --test=low – testing low contention scenario
- --lock=fg – Using fine grained locking with normal pthread locks
  --lock=rw – Using fine grained locking with reader-writer locks

- -t – Number of threads. Minimum number of threads is 3. If less than 3 threads are specified, then the program automatically assumes 3 threads.
- -i – Number of iterations for each thread.
- --print=no – Print statements stating output of get operations and range query operations are omitted. Only the end results are displayed
  --print=yes – All the print statements stating output of get and range query operations are included.

Additionally, enter "./concurrent_tree -h" to get help on the execution instructions.

In case no arguments are specified, the default values assumed are

- --test=low
- --lock=fg
- --t12
- -i1
- --print=yes

Run "./unit_tests.sh" to run 12 tests with different arguments. None of these commands include print statements, therefore only the end summary results are displayed. The high contention commands take some time; therefore, patience is highly advised. If you wish to see the print outputs for each run, run the commands individually with no print argument. Examples are provided below. These test cases present the following three results

- Verifying if the number of nodes inserted correspond to the number of nodes returned by an inorder traversal.
- Verifying that each get operation either successfully returns the correct value corresponding to the key to be searched or successfully returns null if the key specified is not present. This operation is a success if number of successful get operations is equal to the number of get operations carried out.
- Verifying that a range query function between the minimum key possible and maximum key possible returns as many number of nodes as that returned by an inorder traversal operation.

<u>Test cases corresponding to the unit test script</u>

<u>Test Case 1:</u> **Lock** - Fine Grained pthread mutex; **Test** - low contention; **Threads** - 12; **Iterations** - 100
<u>Test Case 2:</u> **Lock** - Fine Grained pthread mutex; **Test** - high contention; **Threads** - 12; **Iterations** - 100
<u>Test Case 3:</u> **Lock** - Fine Grained pthread mutex; **Test** - low contention; **Threads** - 12; **Iterations** - 5000
<u>Test Case 4:</u> **Lock** - Fine Grained pthread mutex; **Test** - high contention; **Threads** - 12; **Iterations** - 5000
<u>Test Case 5:</u> **Lock** - Fine Grained pthread mutex; **Test** - low contention; **Threads** - 30; **Iterations** - 5000
<u>Test Case 6:</u> **Lock** - Fine Grained pthread mutex; **Test** - high contention; **Threads** - 30; **Iterations** - 5000
<u>Test Case 7:</u> **Lock** - Reader-Writer lock; **Test** - low contention; **Threads** - 12; **Iterations** - 100

Test Case 8: **Lock** - Reader-Writer lock; **Test** - high contention; **Threads** - 12; **Iterations** - 100
Test Case 9: **Lock** - Reader-Writer lock; **Test** - low contention; **Threads** - 12; **Iterations** - 5000
Test Case 10: **Lock** - Reader-Writer lock; **Test** - high contention; **Threads** - 12; **Iterations** - 5000
Test Case 11: **Lock** - Reader-Writer lock; **Test** - low contention; **Threads** - 30; **Iterations** - 5000
Test Case 12: **Lock** - Reader-Writer lock; **Test** - high contention; **Threads** - 30; **Iterations** - 5000

Commands corresponding to above experimental results

"pthread_mutex_lock"

Case 1:

Lock – Hand-over-hand locking using normal locks. **Test** – High contention. **Threads** – 12 **Iterations** - 100

```
./concurrent_tree --lock=fg --test=high -t12 -i100
```

Case 2:
Lock – Hand-over-hand locking using normal locks. **Test** – Low contention. **Threads** - 12 **Iterations** – 100

```
./concurrent_tree --lock=fg --test=low -t12 -i100
```

Case 3:
Lock – Hand-over-hand locking using normal locks. **Test** – High contention. **Threads** - 12 **Iterations** - 10000

```
./concurrent_tree --lock=fg --test=high -t12 -i10000
```

Case 4:
Lock – Hand-over-hand locking using normal locks. **Test** – Low contention. **Threads** – 12 **Iterations** - 10000

```
./concurrent_tree --lock=fg --test=low -t12 -i10000
```

Case 5:

Lock – Hand-over-hand locking using normal locks. **Test** – High contention. **Threads** – 24 **Iterations** - 10000

```
./concurrent_tree --lock=fg --test=high -t24 -i10000
```

Case 6:

Lock – Hand-over-hand locking using normal locks. **Test** – Low contention. **Threads** – 24 **Iterations -** 10000

```
./concurrent_tree --lock=fg --test=low -t12 -i10000
```

Case 7:

Lock – Hand-over-hand locking using normal locks. **Test** – High contention. **Threads** – 36 **Iterations -** 10000

```
./concurrent_tree --lock=fg --test=high -t36 -i10000
```

Case 8:

Lock – Hand-over-hand locking using normal locks. **Test** – Low contention. **Threads** – 36 **Iterations -** 10000

```
./concurrent_tree --lock=fg --test=low -t36 -i10000
```

Case 9:

Lock – Hand-over-hand locking using normal locks. **Test** – High contention. **Threads** – 48 **Iterations -** 10000

```
./concurrent_tree --lock=fg --test=high -t48 -i10000
```

Case 10:

Lock – Hand-over-hand locking using normal locks. **Test** – Low contention. **Threads** – 48 **Iterations -** 10000

```
./concurrent_tree --lock=fg --test=low -t48 -i10000
```

Reader-Writer locks

Case 1:

Lock – Hand-over-hand locking using reader-writer locks. **Test** – High contention. **Threads** – 12 **Iterations -** 100

```
./concurrent_tree --lock=rw --test=high -t12 -i100
```

Case 2:
Lock – Hand-over-hand locking using reader-writer locks. **Test** – Low contention. **Threads** – 12 **Iterations** – 100

```
./concurrent_tree --lock=rw --test=low -t12 -i100
```

Case 3:
Lock – Hand-over-hand locking using reader-writer locks. **Test** – High contention. **Threads** – 12 **Iterations -** 10000

```
./concurrent_tree --lock=rw --test=high -t12 -i10000
```

Case 4:
Lock – Hand-over-hand locking using reader-writer locks. **Test** – Low contention. **Threads** – 12 **Iterations -** 10000

```
./concurrent_tree --lock=rw --test=low -t12 -i10000
```

Case 5:

Lock – Hand-over-hand locking using reader-writer locks. **Test** – High contention. **Threads** – 24 **Iterations -** 10000

```
./concurrent_tree --lock=rw --test=high -t24 -i10000
```

Case 6:

Lock – Hand-over-hand locking using reader-writer locks. **Test** – Low contention. **Threads** – 24 **Iterations -** 10000

```
./concurrent_tree --lock=rw --test=low -t24 -i10000
```

Case 7:

Lock – Hand-over-hand locking using reader-writer locks. **Test** – High contention. **Threads** – 36 **Iterations -** 10000

```
./concurrent_tree --lock=rw --test=high -t36 -i10000
```

Case 8:

Lock – Hand-over-hand locking using reader-writer locks. **Test** – Low contention. **Threads** – 36 **Iterations -** 10000

```
./concurrent_tree --lock=rw --test=low -t36 -i10000
```

Case 9:

Lock – Hand-over-hand locking using reader-writer locks. **Test** – High contention. **Threads** – 48 **Iterations -** 10000

```
./concurrent_tree --lock=rw --test=high -t48 -i10000
```

Case 10:

Lock – Hand-over-hand locking using reader-writer locks. **Test** – Low contention. **Threads** – 48 **Iterations -** 10000

```
./concurrent_tree --lock=rw --test=low -t48 -i10000
```

8. **Any Extant Bugs**
   No.