LINGI2251

Software Quality Assurance

Assignment 2

Vincent Tessier - Nathan Rullier – William Glazer

20th April 2020

Question 1:

The invalid read and write operations were caused due to the malloc which only made for one `pthread_t` with `malloc(size(pthread_t))`. However, the program uses an array of `pthread_t`. Thus we changed:

`malloc(size(pthread_t))`

←→

`malloc(size(pthread_t[n_thread]))`

When the program was called with a 1 as an argument it worked because the malloc offers enough space for one pthread_t. However when we had more threads, the program could not run since it tried to read and write the thread ID's to memory locations which were not allocated.

Question 2 :

The memory leak was caused because a malloc was called but no free was there to delete the memory. So to fix it we just freed the memory allocated at the end of the program.

`free(malloc_var)`

Question 3 :

A single thread will always sequentially call the `put()` method which means there are no concurrent accesses since only one instance of put runs at a time. However, 2 threads could end up creating a missing key if the first thread and the second thread both go through the `if(!table.inuse)` statement at the same time and after this modify the `.inuse` attribute. We need a mutex to disable concurrent access.

Question 4 :

The changes made to assure no missing keys were :

line 52 : `assert(pthread_mutex_lock(&lock) == 0);`

line 61 & 65 : `assert(pthread_mutex_unlock(&lock) == 0);`

With those changes we assure that everytime a new value is put into the array it is locked so that no other thread can use it and override a value before the attribute `.inuse` is set to 1. We thus correct concurrent access.

Question 5 :

In the put phase, the two-threaded version is slower than the single-threaded one.

Running time for single-threaded version: 3.035559

Running time for the two-threaded version: 3.421515

Question 6 :

The program won't achieve any speed up because it is still only making one insert at a time since all threads have a bottleneck in the mutex. Having a single thread running sequential inserts or many threads running sequentials inserts due to the mutex wait makes little difference.

Question 7 :

We observe a speedup because it is much more efficient since multithreading has a purpose due to no mutex blocking the threads allowing for proper concurrent access.

Question 8 :

Valgrind reports no errors for `get()` because concurrent access errors are only in place when there are concurrent operations with a write. Concurrent reads are no problem unless someone modifies the values.

Question 9 :

To accelerate the program we divide the array in sections. Since we have 5 buckets we create a mutex per bucket. This allows up to 5 concurrent writes at the same time given that they are in different buckets. We get a write time of 0.67 instead of 2s with sufficient threads.

We could implements more mutexes to speed up further the process by dividing the index entries per bucket. We would then have, for example, 10 mutexes per bucket and 5 buckets leading to 50 mutexes allowing for more concurrent operations. However we would need to deal with indexes going into the next mutex if they are already in use. (E.G. mutex 1 → index [0,10[and we have index=10 in use so we would have to go to mutex 2 and free up mutex 1). The limit to this speedup would be the time and memory required to create and store these mutexes.