You do not have to submit this file unless you do Q3 (bonus)

Scalability assignment (cont'd)

Code:

https://github.com/fxlin/p2-concurrency/tree/master/exp2

Description:

https://fxlin.github.io/p2-concurrency/exp2/

2. A scalable hashtable

Your task: Take an existing hashtable implementation in `glib` and make it scalable.

Context: `glib` is a widely used C library that implements common data structures, such as linked lists, trees, and hashtables.

- A minimum example (saved as hashmapEx.c). Source

```
// required packages
// sudo apt-get install libglib2.0-dev libgtk2.0-dev

#include <stdio.h>
#include <glib.h>

int main(int argc, char **argv) {

        // load string hash table
        GHashTable* sm = g_hash_table_new(g_str_hash, g_str_equal);
        g_hash_table_insert(sm, "a", "alfa");

        // lookup key
        printf("a => %s\n", (char *)g_hash_table_lookup(sm, "a"));

        // replace a value
        g_hash_table_replace(sm, "a", "ALFA");
        printf("a => %s\n", (char *)g_hash_table_lookup(sm, "a"));

        // free memory
        g_hash_table_destroy(sm);
}
```

To use integers as keys/values, see here ("glib2 hash table: GINT_TO_POINTER macro")

To compile:

p2exp2b

```
gcc hashmapEx.c `pkg-config --cflags --libs gtk+-2.0` -Werror -Wall -O2
```

To run:

```
xzl@granger1[~]$ ./a.out
a => alfa
a => ALFA
```

More (reference only when needed):

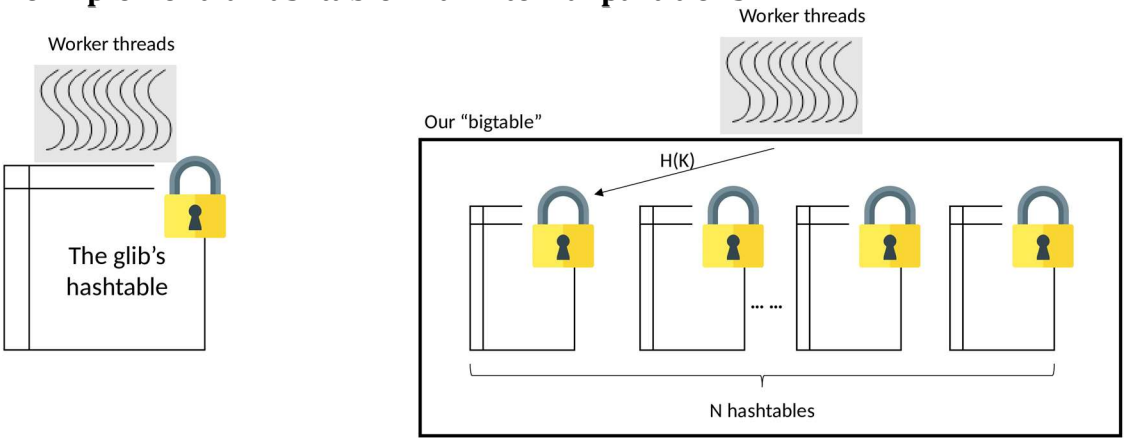
- The glib hashtable API is described here.
- For the API's example usage, see the glib's test code here.
- To add the required header/lib of glib to CMakeList.txt, you can manually add the output of pkg-config, or see here.

Goal: write a benchmark program, which spawns multiple threads for inserting 32-bit (integers) keys to a hashtable in parallel.

> Note 1: the hashtable API expects keys/values as **pointers** to the actual keys/values. It's possible to store 32-bit integer keys/values in the hashtable by casting them to pointers (c.f. GINT_TO_POINTER() and GPOINTER_TO_INT())
>
> Note 2: to generate keys, it is okay just to use monotonically ascending integers so that they are unique.

At the end of the benchmark, validate the correctness by checking all the keys in the hashtable, e.g. no missing or surplus keys.

To implement: a hashtable with internal partitions



The "big lock" approach ensures correctness but cannot scale. We can build our own hashtable (called "bigtable") by wrapping around the glib's hashtable. A bigtable consists of *N* hashtables internally, where *N* is a parameter much larger than the number of threads. Each hashtable has its own lock which must be acquired by a worker thread before the thread inserts keys in this hashtable.

To insert a key K, a worker thread first computes a hash function H(K) to determine which of the N hashtables the key should go. Then the worker thread acquires the lock for the hashtable, does the insertion, and unlock. Since N>>numThreads, the chance of lock contention is low and our bigtable should scale better than the "big lock" approach.

To look for a key K, compute H(K) to find the smaller hashtable, lock it, and access it.

There are other details to take care of. Please read the project description (webpage) carefully.

Deliverables

A tarball containing:

- All the source code. None of the .git/ or binary files. **(40 pts)**
- A short README.txt on how to build and run your code **(5 pts)**
- A short PDF file (around half a page, no more than one page) discussing what has been attempted/achieved and the results. In that file, include scalability plots as you see fit. **(20 pts)**

Only include the above in your tarball! If your tarball is larger than 10MB, we will apply 50% penalty to your score.

Recommended implementation steps

Version 1. Multi-threaded with a big lock. Transform the above version by adding pthread, mutex, etc. Design a test to validate the correctness of the resultant hashtable.

We give students a boilerplate code: exp2b-assignment/hashtable-biglock.c. It is optional for you to use it. If you use it, make sure to understand EVERY line, including its CMakeLists.txt.

It is the student's responsibility to ensure their code is free of bugs.

Version 2. Add internal partitions. Test & profile.

Only submit the final version.

3. (Bonus, 20pts) double-checked locking (0 pts)

```
newP() {
  p = malloc(sizeof(p));
  p->field1 = …
  p->field2 = …
  return p;
```

Thread 1

```
if (p == NULL) {
  lock_acquire(lock);
  if (p == NULL) {
    p = newP();
  }
  release_lock(lock);
}
use p->field1
```

Thread 2

```
if (p == NULL) {
  lock_acquire(lock);
  if (p == NULL) {
    p = newP();
  }
  release_lock(lock);
}
use p->field1
```

In recent lectures, we talked about double-checked locking.

In your own words:

(a) What was the motivation to check if p is NULL without acquiring the lock?

The motivation for checking if p is NULL without acquiring the lock is to make sure that the thread does not create a new p object when a p object has already been created.

(b) Why could a race condition happen between thread 1 and 2? When it happens, what could go wrong?

The race condition can still happen because one thread could attempt to access a field in p without p actually being created by another thread beforehand.

(c) How to eliminate the race condition?

You could a add a read fence before the second check if p == NULL and a write fence in the end of the constructor for p.

Changelog. 3/20/22 added a small code example