Exercises for Hands-on 3: Valgrind

Make sure to fill in questions in the space provided underneath the question text.

Question 1: Study helgrind's output and the ph.c program. Clearly something is wrong with put() on line 61.

- **a)** Identify a sequence of events that can lead to keys missing for 2 threads.
- **b)** Using that sequence, explain why there missing keys with 2 or more threads, but not with 1 thread. Your explanation should refer to the given code and make use of theory to justify your answer.

.

1a)

- Thread 1 can read ` if (!table[b][i].inuse)` and can a usable location , table[B][N]
- Thread 1 can assign the key1 and value1 to table at table[B][N]
- Thread 2 immediately interrupt and also read `if (!table[b][i].inuse)` and determine that table[B][N] is not in use.
- Thread 2 assigns key2 and value2 to table[B][N], overwriting key1 and value1
- Thread 2 then marks table[B][N] as inuse
- Thread 1 marks table[B][N] as inuse

1b)

The loss of keys happens because another thread can also determine that a block of the table is not inuse, as the (determining of inuse, and setting of the key,value, and inuse properties) are not atomic. If there are 2+ threads, another thread can interrupt during these operations, and both can believe that a table block is unused. If there is only one thread running, this set of operations is de facto atomic, as there is no other thread to interrupt it.

Fixing the error

To avoid this sequence of events, insert lock and unlock statements in put so that the number keys missing is always 0. The relevant pthread calls are (for more see the manual pages, man pthread):

```
pthread_mutex_t lock;     // declare a lock
pthread_mutex_init(&lock, NULL);   // initialize the lock
pthread_mutex_lock(&lock);  // acquire lock
pthread_mutex_unlock(&lock);  // release lock
```

The function get() has an example use of locks, and main initializes it.
Modify ph.c to make it correct and recompile with gcc. Test your code first with 1 thread, then test it with 2 threads. Is it correct (i.e. have you eliminated missing keys)? Check the correctness using helgrind. (Note that valgrind slows down ph a lot, so you may want to modify NKEYS to be some small number.)

Question 2: Describe your changes and argue why these changes ensure correctness. (Do not include the entire ph.c code, but do tell us which lines you have added and feel free to include the put code in your explanation.)

---

2)

```
static void put(int key, int value) {
  assert(pthread_mutex_lock(&lock) == 0);
  int b = key % NBUCKET;
  int i;
  // Loop up through the entries in the bucket to find an unused one:
  for (i = 0; i < NENTRY; i++) {
    if (!table[b][i].inuse) { // * A
      table[b][i].key = key;
      table[b][i].value = value;
      table[b][i].inuse = 1; // * B

      assert(pthread_mutex_unlock(&lock) == 0);
      return;
    }
  }
  assert(0);
}
```

As mentioned in question 1, the race condition comes from multiple threads evaluating the line with comment " * A" as true. Again, this can be prevent by making the critical code between points A and B atomic. For simplicity, I've encapsulated the entire function inside of a lock to make the whole function atomic. It acquires the lock in the first line of the function, and releases the lock in the line right before return.

Question 3: Is the two-threaded version faster than the single-threaded version in the put phase? Report the running times for the two-threaded version and the single-threaded version. (Make sure to undo your `NKEYS` change.)

Question 4: Most likely (depending on your exact changes) ph won't achieve any speedup. Why is that?

---

3) Single threaded is fastest.

Performance when locks are on PUT and GET		
#(thread)	PUT completion time	GET completion time
1	2.916412	2.876504
2	3.225117	3.401766
4	3.248693	3.581894

4) If anything, put may experience a slow down, similar to get() when using 2+ threads. Rather than exploiting the use of multiple cores to get a speed up, multithreading could waste time. For instance, say thread1 begins to execute put() and acquires the lock. Immediately after acquiring the lock, thread1 is interrupted. Thread2 begin to execute put() as well, but can not as it is locked. Thread 1 interrupts, and resumes its put() execution. Therefore all of the time that went into the interrupt + switching contexts / registers + attempting to execute put + being denied the lock , was just wasted time. Multithreading doesn't necessarily help when most of the tasks are atomic.

Living Dangerously

Remove the locks from get() in ph.c, recompile, and rerun the program.

Question 5: You should observe speedup on several cores for the get phase of ph. Why is that?

Question 6: Why does valgrind report no errors for get()?

Question 7: How long did this assignment take you?

Performance when locks are on PUT but not GET		
#(thread)	PUT completion time	GET completion time
1	2.987434	2.936057
2	3.115102	1.874828
4	3.142831	1.562860

5) Because, we have removed the locks on `get()`. Similar to my response to #4, we no longer have to worry about switching threads and not being able to acquire a lock. This allows us to exploit the multiple cores of our processors. When get() was atomic, if a thread was interrupted while it had the lock, none of the other threads to perform a get() call, and all of the time that went into switching threads was sunk.

6) Because `get()` does not mutate any data, only reads. Therefore, in this context get() is not invoking a race condition. Therefore, no keys were missing in the run, and valgrind reported no errors/race conditions.

7) 2 hours. But I took some time to learn what all the things in the C file did, as i've never written C before.