

Homework

Part 2: Concurrency –

7) More Locking, Condition Variables

master0d08e360 (20220915-171922)

P. Mainini / E. Benoist / C. Fuhrer / L. Ith

BTI1341 / Fall 2022-23

1 Concurrent List Data Structure

To gain some experience with concurrent programming, you will build a thread-safe list structure and perform some measurements.

1.1 Read and Understand


The “`list`” directory in Git/Moodle contains a basic implementation for a *singly linked* list. Study the source code:

1. Begin with the “`list.h`” header file. It contains the API for the list with a description of every function.
2. Study “`test_list.c`”, which contains some unit tests for the list. Learn how the different functions of the list API have to be used.

⚠ *Do not change “`test_list.c`” during the whole exercise! You may use it at any time to check if your modifications of the list still work correctly.*

3. Finally, have a look at the list implementation in “`list.c`”.

1.2 Multiple Threads and Concurrent Access

Now, a multi-threaded program shall be implemented, where each thread inserts a certain number of values to an instance of the list. *For this, only the file “threads.c” shall be modified!*  The code to be written is quite similar to the homework from 06-concurrency-1.

1. Create an instance of the list.
2. Create a thread function which inserts the required number of values into the list.
3. Write the necessary code to start and terminate the correct number of threads.
4. Run your program with different numbers of threads and values to insert. You should observe a *race condition*, due to which not enough values are inserted into the list. At which number of threads and/or values does it appear?

1.3 Time Measurement


In “threads.c”, add some code to measure the time it takes all threads to add all values to the list. Have a look at the solutions of the homework from 01-processes to see how this can be done.

Run your program with different numbers of threads and values to insert. Can you find some *sweet spot*, where performance is optimal? Are more threads better? What about the number of cores your CPU has?

1.4 Thread-Safe List Implementation

The list shall now be turned into a thread-safe list:

1. Use Valgrind’s **helgrind** tool to identify critical sections.¹
2. Modify the list implementation in “list.c”, such that the list becomes thread-safe. Use Pthread mutexes for this.
3. When you are done, run the program again with various numbers of threads and values. Does the race condition still occur? If so, fix it!
4. Re-check your program with **helgrind** to ensure that there are no race conditions anymore.
5. Run “test_list” to ensure that your modifications have not changed the list API.
6. Finally, use Valgrind to ensure that you have not introduced any memory leaks:
“valgrind --leak-check=yes ./threads T V”.

¹ Use “valgrind --tool=helgrind ./threads T V”. Note that some versions of **helgrind** may generate false positives in `printf()`. Ignore these and only look for warnings regarding the code from “list.c”/“threads.c”!

Is the thread-safe version slower than the initial version? If so, how much / by which factor?

1.5 ★ Improve Performance

In order to improve performance, an idea could be that every thread has its own list (which needs not to be thread-safe). It then adds some (or all) of its values to this list. When a thread is done adding a certain number of values (or all), its list is added to the global list. Try to implement this!

Did you manage to improve performance this way? If so, how faster does this version run compared to the thread-safe version from the previous exercise? Or is it slower?

2 ★ Linux Futexes

If interested, read [Dre04] (also provided as part of the course material) to gain a more in-depth understanding of how Linux futexes work and how they can be used to implement correct mutexes.

References

[Dre04] Ulrich Drepper, *Futexes Are Tricky*.