

OVERVIEW

This library contains the skeleton code for implementing and evaluating two concurrent linked lists: a lock-free and a lock-based. The implementations should work on any Linux-based x86 environment.

Both lists are sorted and provide three main operations:
(i) adding an element to the list (if not already in the list)
(ii) removing an element from the list (if already in the list)
(iii) looking for an element in the list

In our case, a node of the list contains at least an integer key.

The lock-based implementation will use a technique called "hand-over-hand locking", while the lock-free will be based on Harris' algorithm (reference below).

REFERENCES

A thorough explanation of the algorithms we will implement:
<http://www.cs.nyu.edu/courses/fall05/G22.2631-001/lists.slides2.pdf>

Lock-free linkedlist implementation of Harris' algorithm
"A Pragmatic Implementation of Non-Blocking Linked Lists"
T. Harris, p. 300-314, DISC 2001.

DEADLINE

You should provide us with the deliverables (see below) until Tuesday, December 9th, 23:59. The deadline is strict: late solutions will NOT be considered.

INSTALL

You can compile the code (in Linux) by calling:

```
make
in the base directory (./ca14_linkedlist)
```

If the number of cores on your processor is not recognized properly, fix it in include/Utils.h file under the "#if defined(DEFAULT)" definitions.

RUN

The two executables are in the ./bin folder: lb-ll and lf-ll, for the lock-based and lock-free implementations respectively.

```
./bin/lb-ll -h
./bin/lf-ll -h
```

will print the options that the benchmarks accept.
Notice you can compile and execute these benchmarks, but they do not provide the intended functionality, i.e., the implementations of the linked lists are empty.

SCRIPTS

You can find several useful scripts that will help you test and evaluate your implementations.

In details:

- * ./scripts/test_correctness.sh : test the correctness of an implementation, by stressing it
 - * ./scripts/scalability1.sh : benchmark 1 application and get its throughput and scalability
E.g., ./scripts/scalability1.sh all ./bin/lb-ll -i128
 - * ./scripts/scalability2.sh : benchmark 2 applications and get their throughput and scalability
E.g., ./scripts/scalability2.sh all ./bin/lb-ll ./bin/lf-ll -i100
 - * ./scripts/run_ll.sh : execute the workloads that will be part of the deliverable
 - * ./scripts/create_plots_ll.sh : generate the plots (int plots folder) of the data generated with ./scripts/run_ll.sh
- Notice!! You need gnuplot (<http://gnuplot.info/>) installed

Notice!! You need to execute the scripts from the base folder.

IMPLEMENT

You need to change the: (i) linkedlist.h and (ii) linkedlist.c files in the src/linkedList/ src/linkedList-lock/ folders in order to implement the lists.

You can find an easy-to-use interface for atomic operations in include/atomic_ops.h.

(i) linkedlist.h :: contains the interface and the structures of the list.
You only need to change the llist_t and node_t structures to reflect the list and a node of a list of your implementations respectively.

Notice!! These two structures might be different for the lock-free and lock-based versions of the list.

(ii) linkedlist.c :: contains the implementations of the operations of the list, i.e., creating a new list and a new bucket, freeing the list, and, of course, adding, removing, and looking for an element in the list.

Additionally, for the lock-based version, you need to implement and use some locks. You can find the skeletons for initializing, freeing, locking, and unlocking a lock in include/lock_if.h

DETAILS

Memory management is one of most cumbersome problems on lock-free data structures. In other words, when a thread removes an element (a node) from the structure, it cannot always free the memory for that node, because other threads might be holding a reference to this memory.

In this project, we will disregard the memory management part. This means that on a remove operation the memory of the removed node will not be freed. Additionally, this approach entails to every insertion allocating a new node (new memory).

When using locks, memory management is rather straightforward, because of the mutual exclusion property of locks. You can optionally implement memory management on the lock-based version.

DELIVERABLES

You will provide us with:

- * a zip archive of your code, named ca14_lists_Surname.zip, e.g., ca14_lists_Trigonakis.zip
- * a short pdf report, named ca14_lists_Surname.pdf with:
 - (1) a description of your implementations
 - (2) the details of the hardware platform you used for the evaluation
 - (3) evaluation (graphs and explanation) for the following workloads (it is recommended to

- use the scripts in the scripts folder to perform your evaluation)
- (a) Initial size: 128, range 256, update rates: 0, 10, 50 %, #threads: 1, 2, ..., max
 - (b) Initial size: 1024, range 2048, update rates: 0, 10, 50 % #threads: 1, 2, ..., max
 - (c) Initial size: 8192, range 16384, update rates: 0, 10, 50 %, #threads: 1, 2, ..., max

where "max" is the number of cores (or hardware threads) of your target processor.
You should use a test duration that is greater (equal) to 2000ms.

An "easy" way to get the necessary results is to execute the experiments using:

```
./scripts/run_ll.sh
```

and then plots the graph using:

```
./scripts/create_plots_ll.sh
```

You will email us your solutions to both: vasileios.trigonakis@epfl.ch
and georgios.chatzopoulos@epfl.ch

with a title: [CA14] Programming assignment

Notice!! If you do not comply with the naming convention above, your solution will NOT be considered.

GRADING

The points of this programming assignment are bonus, i.e., you can still get a 6 in the final exam, even if you do not get any points from this assignment. You can get up to 0.7 bonus points from this assignment.

We will compile and execute your code on a 48-core AMD Opteron processor with Ubuntu 12.04 Linux. Only the 5 fastest implementations for each data structures will get bonus points. In particular, the fastest implementation (for each of the two data structures) will get a 0.5, the second fastest 0.4, and so on so forth. The sum of the bonuses from both implementations cannot exceed 0.7. For example, a student with 0.4 bonus on the lock-free list and 0.5 on the lock-based one will get a bonus of 0.7, not 0.9.

Breakdown of points:

- * lock-free implementation : (1st) 0.5, (2nd) 0.4, (3rd) 0.3, (4th) 0.2, (5th) 0.1
- * lock-based implementation: (1st) 0.5, (2nd) 0.4, (3rd) 0.3, (4th) 0.2, (5th) 0.1
- * total maximum: 0.7 points

COPYING

There are many more resources out there. However, there is no tolerance for academic dishonesty. Please refer to the University Policy on cheating and plagiarism. Discussion and group studies are encouraged. However, all submitted material must be the student's individual work.

Example behavior that is considered academic dishonesty:

- * Writing code together
- * Copying code from any online resources or previous solutions.

You can talk to the professor or TAs for clarification if you have any questions.

You might be called by the professor or the TAs for explaining your code.