Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

System Construction Course 2019,

Assignment 7

Felix Friedrich, ETH Zürich

Variations of a Lock-Free Stack and the ABA Problem

Lessons to Learn

- Learn how to implement lock-free data structures.
- Experience the difference between data structures that are linked in-place and data structures that use separate items as placeholders from the point of view of lock-free programming.
- Get a deeper insight into the cause and remedy of the ABA problem.

Preparation

1. For this exercise you can start working on your current operating system. As in the previous exercises, you can use the command shell environment and the compiler.
2. Open a console in directory assignments/assignment07
3. In a second step you can continue using Bochs as used in the previous exercise or - even better for this exercise – use qemu. If you feel uncomfortable with this, you can use any virtualisation environment that emulates modern x86 hardware and that has a support for booting from raw disk images as IDE devices. [1]
4. Extract the files for your operating system in directory assignments/assignment07. For example, execute `unzip linux64.zip`.

Implement a Thread-Safe (blocking) Stack

For this lab, we have prepared a module Stack.Mod that contains an abstract Stack object and a concrete ("unprotected") implementation that is vulnerable to race conditions.

Moreover, we provide a module TestStack.Mod that can be used in order to test the Stack. It implements a simple scenario: Initially, a stack is filled with n nodes carrying values 0 to $n-1$, then t Threads are started that repeatedly pop and push elements. Each thread performs m such operations concurrently.

Compile the module and the driver with the following commands:

```
./oberon compile Stack.Mod
./oberon compile TestStack.Mod
```

The test setup can be driven on the unprotected stack using the following command

```
./oberon TestStack.Test unprotected <t> <m> <n>
```

(replace `<t>`, `<m>` and `<n>` by the number of threads, operations and nodes, respectively).

In order to run this command in a virtual environment, you can choose between two operating system kernels: (blocking) A2 kernel and lock-free A2 kernel. With regards to the actual problems

[1] If you like, you can also later on try to use the graphical A2 OS in your system (that you can get from here: http://a2.inf.ethz.ch)

of this task it is irrelevant which one you need. You might notice performance differences, though. They can be created using one of the following commands:

```
./oberon execute makeA2
./oberon execute makeLockFreeA2
```

Run them using Bochs (as in the previous exercise by double clicking the `a2.bxrc` file in windows or executing `bochs -f linuxBochSettings` in Linux). Alternatively, use qemu as follows

```
qemu-system-i386 -serial stdio a2.img
```

Qemu has the advantage that you can set the number of processors and even use virtualization hardware. You can change the memory size with the `m` switch, for example `-m 512`. You can change the numbers of processors using, for example `-smp 8` and you can switch on the virtualization using flag `-enable-kvm`.

1. Shortly experiment with the unprotected stack and find numbers for t, m and n where the stack seems to always work, where it sometimes works and where it does never work (on your computer).
2. Implement a blocking version of the stack (by completing the implementation of the `BlockingStack` in module `Stack.Mod`). Convince yourself that it works using the command

   ```
   TestStack.Test blocking <t> <m> <n>
   ```

Task 2

Having implemented a blocking version of the (thread-safe) stack, now implement a lock-free version of it. You can use the `CAS` construct that is available as built-in function in Active Oberon. The syntax of `CAS` is

```
CAS(variable, old, new)
```

where `old` and `new` need to be values that are assignment compatible to the content of the variable `variable` (which needs to represent a memory location). The semantics of `CAS` are: in one atomic action (that can nevertheless occupy a lot of processor cycles), the value of `old` is compared to the value stored in the memory of `variable`. If the values coincide, the value of `variable` is set to `new`.

`CAS` returns the previous value of the variable. If you want to ignore that return value, you can use the following construct

```
IGNORE CAS(variable, old, new);
```

that effectively turns the `CAS` expression into a statement.

1. Do not (yet) try to find a solution for the ABA problem but implement the lock-free stack according to first principles of lock-free programming by complementing object `LockfreeStack` in module `Stack`.
2. Try out the capabilities of your implementation by calling

   ```
   TestStack.Test lockfree <t> <m> <n>
   ```

   Again, try to find values of t, m, and n where it works. Since you have not yet taken into consideration the `ABA` problem, it will fail for some values of t, m and n.

Task 3

Now implement a lock-free stack that avoids improper reuse of the nodes internally used by the stack. In order to do so,

1. create a new class of container nodes that serve as the nodes of the linked list of the stack and use them as placeholders that refer to what we previously called the nodes. For simplicity (and such that the interface of the stack does not need to be changed), you can keep using the `Stack.Node` (ignoring the `next` field) and simply add another container node with a reference to a `Stack.Node`.
2. For each `Push` operation, you can now allocate a new placeholder node referring a `Stack.Node`. During each `Pop` operation, you may discard the placeholder node.
3. Test your implementation by calling

   ```
   TestStack.Test placeholder <t> <m> <n>
   ```

   with appropriate values of t, m and n and explain to your colleague in detail, why the ABA problem cannot occur with the (garbage collected) placeholder nodes.

Think what you would have to do in order to reuse placeholder nodes (i.e. avoiding the re-allocation of a new object for each `Push`).

Optional

Try to come up with a node cache that could be used in order to avoid reallocation of new objects for each `Push`. Think about where the ABA problem would occur there.

Documents

- System Construction Lecture 7 slides from the course-homepage
  http://lec.inf.ethz.ch/syscon
- Oberon Language Report (Draft 2019) from the A2 repository
- $\mathcal{A}_2$ Programming Quickstart Guide. File A2QuickStartGuide.pdf in folder documents/oberon