ABA

This lesson explains the ABA problem: An analogy for a value being changed in between two successive reads for that value giving the same result.

WE'LL COVER THE FOLLOWING

Programming concurrent applications are inherently complicated. This still holds true if you use C++11 and C++14 features, and that is before I mention the memory model.

ABA

ABA means you read a value twice and it returns the same value A each time. Therefore, you conclude that nothing changed in between, but you missed the fact that the value was updated to B somewhere in between.

Let me first use a simple scenario to introduce the problem.

An Analogy

The scenario consists of you sitting in a car and waiting for the traffic light to become green. Green stands for B in our case, and red stands for A. What's happening?

1. You look at the traffic light and it is red (A).
2. Because you are bored, you begin to check the news on your smartphone and forget the time.
3. You look once more at the traffic light. Damn, it is still red (A).

Of course, the traffic light became green (B) between your two checks. Therefore, what seems to be one red phase was actually a full cycle.

What does this mean for threads (processes)? Here is a more formal explanation:

1. Thread 1 reads the variable `var` with value A.
2. Thread 1 is preempted and thread 2 runs.
3. Thread 2 changes the variable `var` from A to B to A.
4. Thread 1 continues to run and checks the value of variable `var` and gets A. Because of the value `A`, thread 1 proceeds.

Often that is not a problem and you can simply ignore it.

Non-critical ABA

The functions `compare_exchange_strong` and `compare_exchange_weak` suffer the ABA problem that can be observed in the `fetch_mult` (line 7). Here, it is non-critical; `fetch_mult` multiplies a `std::atomic<T>& shared` by `mult`.

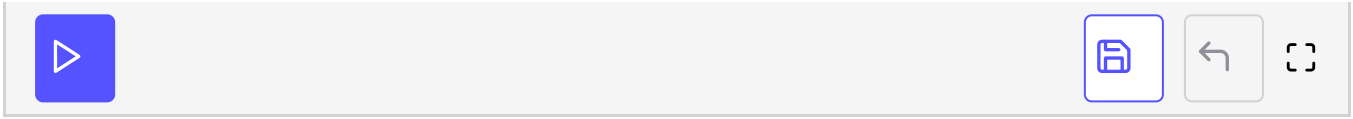
```
// fetch_mult.cpp

#include <atomic>
#include <iostream>

template <typename T>
T fetch_mult(std::atomic<T>& shared, T mult){
  T oldValue = shared.load();
  while (!shared.compare_exchange_strong(oldValue, oldValue * mult));
```

```
  return oldValue;
}


int main(){
  std::atomic<int> myInt{5};
  std::cout << myInt << std::endl;
  fetch_mult(myInt,5);
  std::cout << myInt << std::endl;
}
```



The key observation is that there is a small time window between the reading of the old value `T oldValue = shared.load` in line 8 and the comparison with the new value in line 9. Therefore, another thread can kick in and change the `oldValue` from `oldValue` to another value, then back to `oldValue`. The `oldValue` is the A, another value is the B in ABA.

Often it makes no difference if the first value is from the second read operation as long as the value is the original one. But in a lock-free concurrent data structure, ABA may have a great impact.

A lock-free data structure

I will not present a lock-free data structure in detail here. I will use a lock-free stack that is implemented as a singly linked list. The stack supports only two operations.

1. Pop the top object and return a pointer to it.
2. Push the specified object to stack.

Let me describe the pop operation in pseudo-code to give you an idea of the ABA problem. The pop operation executes the following steps until the operation is successful.

1. Get the head node: `head`
2. Get the subsequent node: `headNext`
3. Make `headNext` to the new head if `head` is still the head of the stack

Here are the first two nodes of the stack:

```
Stack: TOP -> head -> headNext -> ...
```

Let's construct the ABA problem.

ABA in Action

Let's start with the following stack:

```
Stack: TOP -> A -> B -> C
```

Thread 1 is active and wants to pop the head of the stack.

- Thread 1 stores

```
head = A
headNext = B
```

Before thread 1 finishes the pop step, thread 2 kicks in.

- Thread 2 pops A

```
Stack: TOP -> B -> C
```

- Thread 2 pops B and deletes B

```
Stack: TOP -> C
```

- Thread 2 pushed A back

```
Stack: TOP -> A -> C
```

Thread 1 is rescheduled and checks if `A == head`. Because of `A == head`, `headNext` (B) becomes the new head - but B was already deleted; therefore, the program has undefined behavior.

There are a few remedies to the ABA problem.

Remedies

The conceptional problem of ABA is quite easy to understand. A node such as `B == headNext` was deleted although another node `A == head` was referring to it. The solution to our problem is to get rid of the premature deletion of the node. Here are a few remedies.

Tagged state reference

You can add a tag to each node indicating how often the node has been successfully modified. The result is that the compare and swap (CAS) method will eventually fail although the check returns true.

The next three techniques are based on the idea of deferred reclamation.

Garbage Collection

Garbage collection guarantees that the variables will only be deleted if it is not needed anymore. This sounds promising but has a big drawback. Most garbage collectors are not lock-free, therefore, even if you have a lock-free data structure, the overall system won't be lock-free.

Hazard Pointers

From Wikpedia: Hazard Pointers:

In a hazard-pointer system, each thread keeps a list of hazard pointers indicating which nodes the thread is currently accessing. (In many systems this "list" may be provably limited to only one or two elements.) Nodes on the hazard pointer list must not be modified or deallocated by any other thread. (...) When a thread wishes to remove a node, it places it on a list of nodes "to be freed later", but does not actually deallocate the node's memory until no other thread's hazard list contains the pointer. This manual garbage collection

can be done by a dedicated garbage-collection thread (if the list “to be freed later” is shared by all the threads); alternatively, cleaning up the “to be freed” list can be done by each worker thread as part of an operation such as “pop”.

RCU

RCU stands for **R**ead **C**opy **U**pdate and is a synchronization technique for almost all read-only data structures. RCU was created by Paul McKenney and has been used in the Linux Kernel since 2002.

The idea is quite simple and follows the acronym. In order to modify data, you make a copy of the data and modify that copy. In contrast, all readers work with the original data. If there is no reader, you can safely replace the data structure with its copy.

For more details about RCU, read the article What is RCU, Fundamentally? by Paul McKenney.

> **Two new proposals**
>
> As part of a concurrency toolkit, there are two proposals for upcoming C++ standards. The proposal P0233r0 for hazard pointers and the proposal P0461R0 for RCU.