RAST: highly distributed DB project.

Produced by NOA team

Contents

1. Coding patterns

1.1. Function specifications cascade merging

Let us assume we are to write some template class, which may be customized with any amount of template parameters. Then, of course, we have to use variadic templates. When using variadic templates, we are to define class through its older version with less amount of template arguments, thus forcing compiler to produce a code for several hierarchical classes. What is often used here is so called cascade inheritance. That is, new class which is specified by a larger amount of template parameters is being inherited from an older version.

Imagine that we want target class $X < A, B, C >$ to have a template function, which has concrete specifications for template parameters being equal to A, B or C. For instance, all classes A, B and C have *Output* method and we want class X to have *Output* $< T >$ method which for $[T = A]$ calls *Output* method from class A, for $[T = B]$ calls *Output* method of class B, etc.

First difficulty we would face considering such approach is that template specifications are not allowed in non-global scope (inside class X, for instance). That is due to the fact, that if those specifications depend on template parameters of X class itself, the more complicated version of template processing algorithm would be required to handle this. But, template processing is not that smart enough to guess what author meant here. So, we have to specialize functions another way.

Let us do specifications by means of input arguments. In order to do so let us introduce an empty template class $TypeSlecifier < T >$. We may now write three versions of function, which compiler would treat well:

```
class X {
    void Output(const TypeSlecifier<A>&) {
        A().Output();
    }
    void Output(const TypeSlecifier<B>&) {
        B().Ouptut();
    }
    void Output(const TypeSlecifier<C>&) {
        C().Ouptut();
    }
};
```

Such a function would be called as follows:

```
x.Output(TypeSlecifier<A>());
x.Output(TypeSlecifier<B>());
x.Output(TypeSlecifier<C>());
```

What remains is to represent this code using template processing:

```
template <typename ... Args>
class X {
    void Output() {}
};
template <typename T, typename ... Args>
class X<T, Args...>: public X<Args...> {
    using X<Args...>::Ouptut;
    void Output(const TypeSlecifier<T>&) {
        T().Ouptut();
    }
};
```

By means of “using” construction we add all versions of Output function from previous inheritance level to the next one, afterwards adding new version associated with current new template parameter T. As far as we put “using Output” inside the code, we have to define “Output” symbol inside very base class, which accepts zero template parameters. In order to do that, we firstly define version of class “X”, which accepts any amount of template arguments. The trick is that such implementation would only be considered if only “X” was zero template arguments, otherwise the declaration $X < T, Args... >$ would be chosen by compiler to process.

The only drawback here is that a user of the class has to know about some “TypeSpecifier” class to use “Output” function. But this can be fixed by introducing a final template function:

```
template <typename ... Args>
class X {
   void Output() {}
};
template <typename T, typename ... Args>
class X<T, Args...>: public X<Args...> {
   using X<Args...>::Ouptut;
   void Output(const TypeSlecifier<T>&) {
      T().Ouptut();
   }
   template <typename T2>
   void TemplateOutput() {
      Output(TypeSlecifier<T2>());
   }
};
```

Now a call to “Output” function looks as follows:

```
x.TemplateOutput<A>();
x.TemplateOutput<B>();
x.TemplateOutput<C>();
```

That is what is called “cascade merging of specifications” here.

The example on which technique is explained is rather simple and it would be weird to write such a complicated code, because one would just do something like this:

```
template <typename T>
void Output() {
   T().Output();
}
```

In that, in the example above there was no actual need in inheritance, the example is just used in order to demonstrate the pattern. But when is this pattern really needed then?

A situation where such a complicated technique starts to be useful is when custom function needs to know something about the moment template argument was being added to template variadic list. That is, when in such a “Output” function we have to access a specific inheritance level. Imagine, for instance, that in class $X < A, B, C >$ we want to have a function *GetIndex*, which would by class name A, B or C return an index 2, 1 or 0. To do it we have to store somewhere a correspondence between classes and indices. The best place for this is an inherited class:

```
template <typename ... Args>
```

```
class X {
   X()
   : max_index(0)
   {}

   int GetIndexImpl() {}

   void Output() {}
protected:
   int max_index;
};
template <typename T, typename ... Args>
class X<T, Args...>: public X<Args...> {
   using X<Args...>::GetIndexImpl;
   X()
   : inheritance_level_index(max_index++)
   {}

   int GetIndexImpl(const TypeSlecifier<T>&) {
      return inheritance_level_index;
   }

   template <typename T2>
   int GetIndex() {
      GetIndexImpl(TypeSlecifier<T2>());
   }
private:
   int inheritance_level_index;
};
```

That we have basically done here is incrementing “inheritance_level_index” in instantiation of each hierarchical class object, thus having increasing indices in the inheritance chain. Afterwards, while user is calling $GetIndex < T >$ function, the corresponding “GetIndexImpl” from the corresponding inheritance level would be called, and this function has a direct access to all the data stored in the level of inheritance it was defined. Thus, it can easily return the required index.

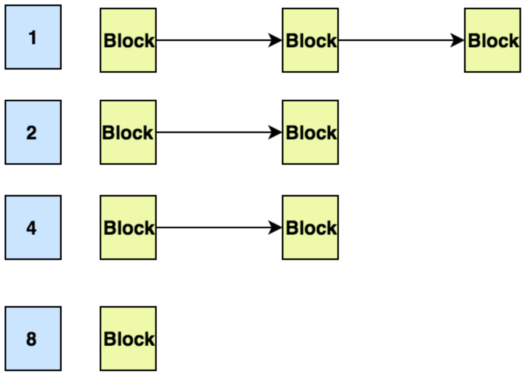
2. Algorithmic basis

2.1. Free-list multi-level allocator

On of the most important steps in order to make a low-latency system is to consider the allocation problem. The problem is that sometimes different components of the program require very small parts of memory to be allocated. Asking the system to allocate memory directly would result in slower performance. That’s why it is better to allocate large memory segment and distribute its parts among several consumers.

But here we face another difficulty, so called fragmentation problem. Imagine that an allocated segment is separated into several parts owned by consumers. Some time after, some of the consumers returned their own parts to allocator (calling deallocate method). Afterwards it appears that the returned memory in control of allocator is represented as distinct small fragments. Those fragments cannot be given to a consumer, who asks for a larger memory segment, than the size of the separate fragment. If there is a lot of such unusable fragments, we would have a lot of memory allocated, which will never be used. In order to avoid this problem, we have to force each small fragment to be used by a consumer who asks for a small amount of memory.

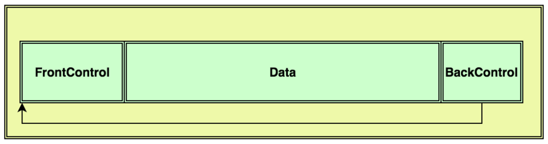
The main idea is to use a memory levels.



For each block size we calculate an integer part of $log(size)$ and put all blocks of memory in the corresponding list.

By means of the list, we can easily find the block of needed size and remove (detach) it from the list, giving ownership to the consumer, who asked to allocate memory. Likewise, when a consumer asks to deallocate the block, we may put (attach) it again in the list.

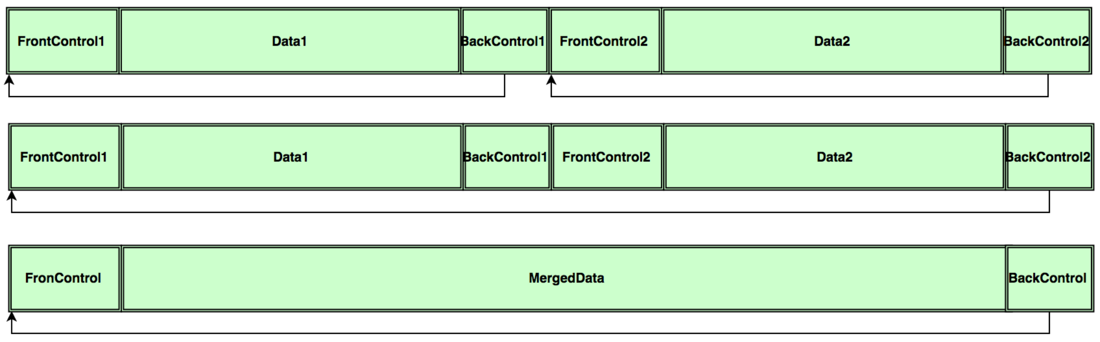
Also we want to join small segments of memory returned by consumers to one large segment in case if they go one after another in the global memory space. In order to do this, let us have the following structure of the memory block:



The block will consist of three logical parts:

- FrontControl, which stores the $data\_size$ which is size of the block, $localnext$ and $localprev$ pointers, $offset$ in external allocated segment, and $total\_size$ of external segment, and is_owned – whether the block is owned by the allocator itself or was it given away to the consumer.
- Data – raw byte space, which would be given an external consumer to control
- BackControl, which stores pointer to the FrontControl (pointer to the beginning of the entire block)

$localprev$ and $localnext$ pointers will help us to build freelists, which were described above. BackControl block will help us to find the block, previous to some specific Block, so that we could join two blocks together in case they both have been returned to the allocator. Here is how joining procedure works:



The steps are very simple:

- detach first block from its list
- detach second block from its list
- point BackControl2 to FrontControl1
- change FrontControl1 information about block size
- do nothing about BackControl1 and FrontControl2, because now we threat them as a garbage part of MergedData, and we do not have to care about their contents so far
- attach new merged block to its new list (based on $log(size)$ where $size$ is a sum of sizes of these two blocks) by changing $localprev$ and $localnext$ pointers of FrontControl.

Also, one can easily imagine how to split one large block into two smaller.

Blocks contained in the same external memory segment (which allocator asked from the system) may be joined and split so that we know exactly how many consequent free to use bytespace fragments do we have.

Finally, the allocation procedure is the following:

- calculate integral part $N = int(log(size))$ of the size of the memory consumer asks to allocate
- if the corresponding N-th list of blocks is empty, we allocate several external blocks of the sizes 2^N and build Block structure upon each of them, putting blocks to the N-th list
- detach first block from the list
- split the block into two blocks: first block has exactly needed size, second block consists of what is remained
- attach second block to its corresponding list
- mark the first block as not owned by allocator
- return a pointer to the Data section of the first block to a consumer

The deallocation procedure:

- receive from a consumer a pointer to deallocate data
- calculate the pointer to FrontControl (by subtracting $sizeof(FrontControl)$ from Data pointer)

- mark block as owned by the allocator
- read $offset$ from FrontControl, if $offset > 0$, than there is some previous block in the external memory segment.
- calculate the pointer to BackControl of the previous block (by subtracting $sizeof(BackControl)$ from the pointer to the current block)
- get the pointer to the FrontControl of the previous block via BackControl
- check whether the previous block is owned by the allocator and if it is, merge it with the current block
- if $offset < total\_size$ where $total\_size$ is the size of external allocated memory segment, we may want to check whether we can merge with the forthcoming block
- calculate the pointer to the FrontControl of the forthcoming block (by adding $data\_size + sizeof(FrontControl) + sizeof(BackControl)$ to the pointer to the current block)
- check if the forthcoming block is owned by allocator using its FrontControl and if it is, merge it with the current block.

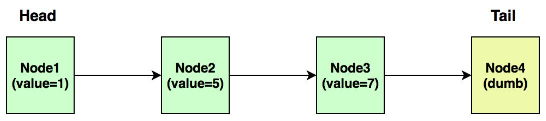
2.2. Lock-free queue (without deallocations)

Queue is a basic container, which is often desired to be used in more or less complicated systems. Although queue implementation is very simple, it is not thread-safe. In order to make it thread-safe, one may decide to use mutexes in Pop and Push operations. The idea is not good enough for several reasons:

- all threads would compete against each other for gaining control over mutex,
- all kernels would have to sync the mutex state via cache ping-pong, which would lead to plenty of time waisted on system calls,
- jobs of threads would be serialized, that is, the queue is 100% not scalable.

The main idea is that if we get down to the details of queue implementation, we may reorganize it in such a way, that we would not need to have mutually exclusive access to guarantee that each operation works fine. That is, we may obtain a lock-free algorithm on the queue.

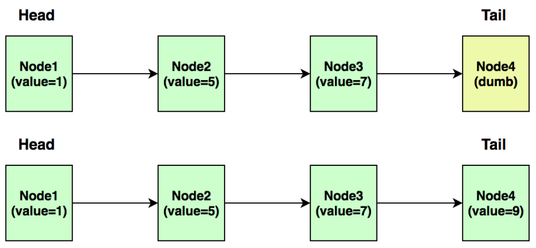
In order to do that, let us represent queue as a linked list, each node of each except for the last one, contains data elements. The last node, which is the tail of the queue would not contain any data at all. Let us call it a dumb node:



Head, Tail and Next pointers inside data structure would be atomic pointers, which would be changed using CAS operations. The main idea of Push is:

- put new data to the dumb node
- create another dumb node and mark it as a tail

But these are two distinct actions, which are not guarded by mutual exclusion and thus are not atomic. So, by choosing this strategy we should agree that it is OK for the queue to be one of the following two:



That is, we agree on purpose that queue may either have dumb element or not have dumb element. But if it is like that, we have to redesign Push:

- try CAS data on the tail (nullptr → our new data)
- if CAS succeeded, the queue actually had a dumb node and now we may proceed to adding a new dumb node
- if CAS failed, the queue does not have a dumb element, so we firstly create a dumb node as a tail, and retry the algorithm from the start

Changing tail pointer is also a CAS operation, but in case if it fails, we realize that another thread have already moved tail forward and thus we do not have to do anything, so we will not process tail CAS operation failure on purpose without harming algorithm invariants at all.

The detailed version of Push consists of the following steps:

- allocate new data
- allocate new dumb node
- L1: load tail
- L2: try CAS tail.data (nullptr → allocated new data)
- L3: if L2 succeeded, try CAS tail.next (nullptr → new dumb node)
- if L3 failed, somebody already moved tail.next to new node, so deallocate new dumb node, try CAS tail (old_tail → value of tail.next), ignore the results and finish Push
- if L3 succeeded, try CAS tail (old_tail → new dumb node), do not care about results and finish Push
- L4: if L2 failed, we do not have dumb node in the end of the queue, try CAS tail.next (nullptr → new dumb node)
- if L4 succeeded, we shall allocate new dumb node, try CAS tail (old_tail → value of tail.next), ignore the result of this operation and repeat algorithm from L1 point
- if L4 failed, try CAS tail (old_tail → value of tail.next), ignore the result of this operation and repeat algorithm from L1 point using the same already-allocated new dumb node

Note that the only action among these which may throw is an allocation, but all allocations happen-before successful L2 step. That is, in case if L2 succeeded, the rest of the code in Push in exception-free. Success of L2 operation means that new data is put to the queue, failure means that the data was not put to the queue yet. This implies we are having strong exception safety on Push operation: if it throws, it would surely mean that data was not put to the queue. This is the best available exception guarantee (after "noexcept") for a queue user. Of course, guaranteeing noexcept is impossible, because no one can imagine Push operation without memory allocations, which always may throw.

Now let us consider Pop operation. It would check if head is equal to tail. If it is, the queue consists of dumb node only, thus there is no data. In this case Pop would return empty unique_ptr. In the other case, we will switch the head to the next element, giving away first node data.

The detailed version of Pop consists of the following steps:

- read head, tail
- L1: check head == tail
- if L1 is true, return empty unique_ptr
- L2: if L1 is false, do CAS head (current → next)
- if L2 failed, retry from the start
- if L2 succeeded, store and exchange data from old head, return it to the caller

Notice that Pop is exception-free.

The implementation described here does not do deallocations, which poses a memory leak problem. The problem is handled here.

2.3. Multi-counters deallocation technique

Almost any algorithm underlaying some lock-free data structure requires careful deallocation technique. The algorithm of lock-free queue we have described above is not an exclusion. The implementation considered does not deallocate memory at all, thus suffering from a memory leak.

The simple but not correct approach is to deallocate node, which is being popped. The problem here is that we have to provide strict guarantee that no other thread is still working with data, which current thread is going to return to the memory heap.

More complicated, but still not correct idea is to count references to the node. We increase reference count if we are going to use a node, and decrease when it is no longer needed. And when counter hits zero, we are going to deallocate the data. Unfortunately, this attempt to write thread-safe deallocation code would also fail, because the counter being equal zero at some point of time does not guarantee that it would be zero the next moment of time. This implies, that checking whether the counter is zero is a useless operation: after checking conditional has passed, and we moved forward to deallocating data, some other thread could ask for one more copy of the object and increase reference count by 1. But, as far as in the current thread we have already decided to deallocate memory, we no longer care about reference counter value. The result is a race condition in the code, which again leads to the fact that some thread uses memory another thread is going to deallocate.

The most complicated, and this time, absolutely correct option, is to always bear in mind reference counter, but start to attempt comparing it with zero and deallocating memory only after the moment when we can guarantee that no other reference to the object is going to be created in the future. That is, we wait until the user-written code somehow manages to say "I will no longer create new references on this specific data". After this moment, we start to wait until user-written code stops to use existing (created before this moment) references. And after that, we may freely deallocate the memory, without any races.

In a more general case, there are several users, which may produce copies of the references to the object. Then reference counting mechanism should receive a promise not to create copies in the future from each one of them. In order to do that, we would use $external\_counters$ variable. At the initialization point, it stores the number of sources, which can produce copies of references to the object. Once a source tells that it would no longer produce copies, the $external\_counters$ decreases. Than it equals zero, we start trying to deallocate the data.

In order to store reference count, we introduce $internal\_counter$, which is stored in the data itself. Apart from that, each source, which can create reference copies, would contain $external\_counter$ – a number of reference copies it created on specific data.

Let us call source, which has not yet promised not to create copies of references, an *active* source. In our three-counter approach the following invariant would hold: for each data element sum of $internal\_counter$ and all $external\_counter$ values of all active sources is equal to reference count to this data.

To sum up, data element contains $internal\_counter$ and $external\_counters$ inside itself, while each reference to the data element consists of the pointer as well as $external\_counter$.

Now we have to define several procedures.

The access protocol:

- source, which want to copy reference, increases its $external\_counter$
- ... some operations over data element ...
- send data element a command to decrease its $internal\_counter$

Source nocopy promise:

- send a data element command to atomically decrease $external\_counters$ and to increment $internal\_counter$ by the value of $external\_counter$ of the current source

The cleaning condition:

- if *internal_counter* and *external_counters* of data element are both equal zero, deallocate the data

This condition is checked both on the finish of the access protocol and after receiving nocopy promise and if true, data is deallocated.

It is very simple to prove that all the operations described keep invariant that sum of *internal_counter* and all *external_counter* values of all active sources is equal to reference count. Because of that, cleaning condition is proven to be true. Indeed, if *external_counters* equals zero, all sources promised not to create copies in the future. Therefore, by that moment there are no active sources, that is, *internal_counter* (summed up with nothing) now stores the exact amount of references left. And, of course, if it also equals zero, then all access protocols are complete. That surely implies we can deallocate data with no problems.

There are also a few rather important details about the implementation:

- *internal_counter* and *external_counters* are stored in the data element itself and should mutate atomically and simultaneously. This can be lock-free only on processors, which support DWCAS.

- *external_counter* and pointer to the data element should also be packed to the one structure and mutate using DWCAS

- all operations described are exception-free, which means that the method can be used in any data structure without harming exception guarantees of the data structure itself.

- in order to simplify matters in the access protocol, we would incapsulate copying in the constructor of special CopyGuard class, and incapsulate decreasing *internal_counter* in its destructor.

2.4. Multi-counters lock-free queue (final algorithm)

Here we are to combine the ideas in the section about lock-free-queue and the section about multi-counters deallocation.

Now the node of the queue is going to contain its *internal_counter* and *external_counters*. In the queue we also need several atomic variables, which point to the nodes: head, tail and next. Let the head, tail and next be objects, which contain *external_counter* and a pointer to the node.

Also, head, tail and next would be associated with three sources, which may produce reference copies. It is required for the correct usage of access protocol, because we would have to access head, tail and next pointers in our algorithm, and the only way to access data element is to duplicate its reference.

While where to use an access protocol and a cleaning condition is a rather simple question, we should consider when do we send nocopy promises.

For tail, we should send nocopy promise when tail is moved further than current data element. This implies that the current data element is never going to be a tail in the future, thus we may guarantee that reference on it cannot be copied from tail.

For next and head, we will send nocopy promise only when the data element is popped out of the queue. This would surely guarantee that no other head and next pointers will ever point to current data element in the future, and the corresponding references will never be copied.

The final implementation of Push operation:

- allocate new data

- allocate new dumb node

- L1: copy tail using CopyGuard (forcing the increase of *external_counter*)

- L2: try CAS tail.data (nullptr $\rightarrow$ allocated new data)

- L3: if L2 succeeded, try CAS tail.next (nullptr → new dumb node)
- if L3 failed, somebody already moved tail.next to new node, so deallocate new dumb node, try CAS tail (old_tail → value of tail.next). If succeeded, send nocopy promise to the node. In any case finish Push.
- if L3 succeeded, try CAS tail (old_tail → new dumb node). If succeeded, send nocopy promise to the node. In any case finish Push.
- L4: if L2 failed, we do not have dumb node in the end of the queue, try CAS tail.next (nullptr → new dumb node)
- if L4 succeeded, we shall allocate new dumb node, try CAS tail (old_tail → value of tail.next). If succeeded, send nocopy promise to the node. In any case repeat algorithm from L1 point (CopyGuard destructor is called implicitly in the end of the cycle iteration, forcing the decrease of *internal_counter*)
- if L4 failed, try CAS tail (old_tail → value of tail.next). If succeeded, send nocopy promise to the node. In any case repeat algorithm from L1 point using the same already-allocated new dumb node (CopyGuard destructor will be called implicitly)

The final implementation of Pop operation:

- copy head using CopyGuard
- L1: check head == tail
- copy head.next using CopyGuard
- if L1 is true, return empty unique_ptr
- L2: if L1 is false, do CAS head (current → next)
- if L2 failed, retry from the start (CopyGuard destructors for head and next are called implicitly)
- if L2 succeeded, store and exchange data from old head
- trying until success to do CAS next (current_value → nullptr_special). It cannot hang for an infinite time, because, since the head have already been moved, no new calls to Pop will deal with this "next" pointer of popped head. This means, that we have to wait only finite amount of calls in another threads, which accidentally read same head to be popped, to finish. Threads, which have accidentally read the same head to be popped, would face L2 operation failure, and immediately after that would go to the start of algorithm, where they would reload another head. So, the threads, which may access the same popped head, would not be blocked, they would just go out of the way of the current thread and let it change "next" pointer. Threads, which do Pop, may also want to access current vertex in case if the queue is empty and tail == head. But they also cannot be blocked, because there is only one cycle in the Pop operation, which rereads data at the start, and the next iterations would not see already-popped head and would not interact with the "next" pointer in question. As all interactions with "next" would fade away after finite number of steps in finite number of threads, we would be able to do CAS successfully. That is why it is lock-free to wait until CAS next (current_value → nullptr_special) actually succeeds.
- after that, send nocopy promise of type "next" about the next element of the popped head (because it will never be someone's else next element, although may still be inside the queue)

- send nocopy promise of type “head” about the popped head (because it will never be a head again)
- return data from popped head to the caller (CopyGuard destructors are called implicitly)

What is specific about Pop operation is this weird nullptr_special we used in CAS next. The matter of the fact is that to guarantee strictly that the node, contained in the “next” pointer, would never be copied, we have to assign “next” pointer to something else. Exactly this assignment, as have already been explained, waits for other threads, which want to operate with the same “next” pointer, to stop making “next” reference copies. Still there is a field for bug in the code. If we assign next to nullptr, we may open the ability for Push operation to change this “next” pointer, because Push operation does CAS next (nullptr → new dumb node). The Push operation is based on the idea that the only node, which has “next” pointer being equal to nullptr, is the tail node. By assigning nullptr to “next” pointer of the popped head, we corrupt this invariant and Push may attach new node to the popped node, which is totally unacceptable. Thus, if we want to change “next” pointer value, we cannot use nullptr for this purpose. But, we may use another special pointer value, which we are sure that nobody else uses. This is what is called a nullptr_special here.

What is remained is to deallocate the rest of nodes of the queue in destructor. This is done by calling Pop a lot of times in the destructor (since Pop is noexcept, we may freely do so). Afterwards, the only dumb vertex remained, which cat be either deallocated directly, or we may send nocopy promises from head and tail and it would also be deallocated automatically by three-counters logic.

To summarize, we designed Push and Pop operations in such a way, that

- all data allocated will at some point be deallocated
- all accessed data in each thread is protected and would not be deleted while read
- Push can throw, strong exception safety is still guaranteed for the reason that reference counting itself is exception-free and all guarantees from lock-free queue are preserved
- Pop is exception-free

2.5. Multi-counters lock-free queue (productivity)

The algorithm of queue appears to be so complex, that one may wonder: is it really faster than the queue, protected by mutex? Let us do a couple of tests to show the difference between these two implementations.

Imagine we have 4-processor hardware. Imagine there are two threads, one of which wants to send 10 million Pop requests, another wants to send 10 million Push requests.

In case of lock-free queue the result is the following:

```
real    0m6.079s
user    0m11.314s
sys     0m0.077s
real RPS = 3.33 million
```

In case of mutex-protected queue the result is the following:

```
real    0m5.500s
user    0m6.980s
sys     0m3.146s
real RPS = 3.63 million
```

One may think that it is not so impressive, because lock-free implementation was 0.5 seconds slower (real time). But a closer look will reveal, that

- mutex-protected queue takes a lot of system time. Of course, it does, because a lot of time is required to synchronize very-fast-changing values of mutex bool flag. Because the threads compete for the ownership of the mutex, cache ping-pong is very likely to happen, forcing queue to be slower
- user time of lock-free queue is even bigger, 11.3 seconds, which is bad comparing with 6.9 seconds of mutex-protected version. But, from a scalability point of view, the lock-free implementation is far better. Using two threads, we gain almost ideal ratio of 11.3 seconds of user time and 6 seconds of real time. As for mutex-protected queue, we get almost no scalability, because each operation is protected with a mutex. Of course, this leads to a threads serialization, and we gain 6.9 seconds of user time resulting in 5.5 seconds of real time, which is a bad result for two-threaded program.

So, lock-free queue is slower, because it contains a lot more logic in its implementation. But, it does not spend so mush time on system calls and is ideally scalable.

Let us watch what is going to be if there are two threads, which try to Pop 10 million elements, and also two threads, which try to Push 10 million elements (4 threads in total).

In case of lock-free queue:

```
real    0m10.082s
user    0m37.448s
sys     0m0.949s
real RPS = 4.0 million
```

In case of mutex-protected queue:

```
real    0m17.316s
user    0m23.260s
sys     0m25.856s
real RPS = 2.3 million
```

So here we see the drawbacks of mutex-protected queue in action:

- the system time increased dramatically, because now 4 threads and competing for single mutex, and its value needs to be synchronized to 4 processors instead of 2
- the scalability is still poor: we get 17.3 seconds of real time having 23.2 seconds of user time.
- mutex-protected queue degrades in RPS after we increased threads count: from 3.6 million queries per second to 2.3 million. That is quite a bad degradation.

And again we clearly see the advantages of lock-free queue:

- it waists less than a second of system time, that is, almost 26 times less that mutex-protected queue. This means, it would not go slower if there would be a lot of another processes doing system calls
- it is still almost perfectly scalable: we get 10 seconds of real time having 37.4 seconds of user time, which is a good result for a 4-threaded application
- the user time of lock-free implementation still exceeds the user-time of mutex-protected queue (the fact that lock-free queue needs more time to execute its logic cannot be changed), but, due to scalability, real time shows that now, with 4 threads, it is almost 1.5 times faster than a mutex-protected implementation

- lock-free queue does not suffer from RPS degradation while we scale concurrency. It even became faster: from 3.33 million requests per second it increased up to 4 million requests per second. That is a very good result.

On the account of these simple measurements, we decided that it is better to use a lock-free algorithm.

3. Detailed Code Architecture (classes and methods)

3.1. TLS free-list multi-level allocator

The class implements the allocation logic described here.

```
class FreeListMultiLevelAllocator {
    FreeListMultiLevelAllocator()
    // Exceptions:
    //     may throw, strong exception safety

    FreeListMultiLevelAllocator(
        const FreeListMultiLevelAllocator&) = delete;

    FreeListMultiLevelAllocator(
        FreeListMultiLevelAllocator&&) = delete;

    FreeListMultiLevelAllocator& operator=(
        const FreeListMultiLevelAllocator&) = delete;
    // We will use object as s singleton, no copying and assigning

    template <typename T>
    T* Allocate(const size_t size);
    // Return value:
    //     a pointer to allocated block
    // Exceptions:
    //     may throw std::bad_alloc
    //     strong exception safety: no side effects in exception case

    template <typename T>
    void Deallocate(T* pointer, const size_t size) noexcept;
};

thread_local FreeListMultiLevelAllocator global_allocator;
// Memory allocations would be controlled by this TLS singleton

template <typename T>
class FixedFreeListMultiLevelAllocator<T> {
    FixedFreeListMultiLevelAllocator() noexcept;

    FixedFreeListMultiLevelAllocator(
        const FixedFreeListMultiLevelAllocator&) noexcept;

    template <class U>
    FixedFreeListMultiLevelAllocator(
        const FixedFreeListMultiLevelAllocator<U>&) noexcept;
    // These three methods are empty,
    //    they are needed for compatibility with std::allocator

    T* allocate(const size_t n);
    // Calls global_allocator.Allocate<T>(n)
    // Return value:
    //    a pointer to allocated block
    // Exceptions:
    //    may throw std::bad_alloc
    //    strong exception safety: no side effects in exception case

    void deallocate(T* p, const size_t n) noexcept;
    // Calls global_allocator.Deallocate<T>(p, n)
};
```

3.2. Lock-free queue

$LockFreeQueue < TElement >$ is an implementation of lock-free queue described here. It contains the following functions:

```
class LockFreeQueue<TElement> {
    void Push(TElement new_element);
    // Exceptions:
    //    Strong exception safety: in case of failure
    //    the element is not pushed to the queue,
    //    there are no visible side effects

    std::unqiue_ptr<TElement> Pop() noexcept;
    // Return value:
    //    nullptr unique_ptr if the queue was empty,
    //    unique_ptr pointing on TElement in case of successful pop
};
```