RAST: highly distributed DB project.

Produced by NOA team

Contents

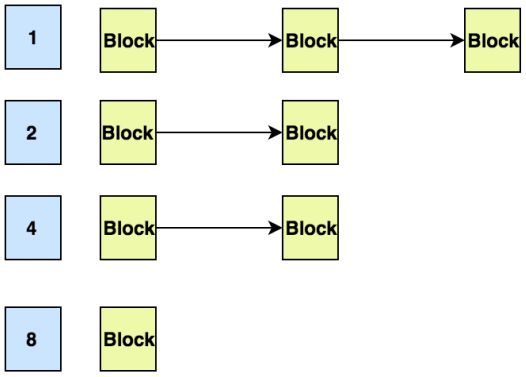
1. Algorithmic basis

1.1. 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 can 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.

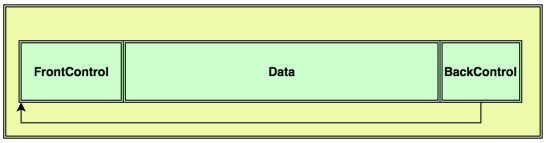
The main idea is to use a memory levels.



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:



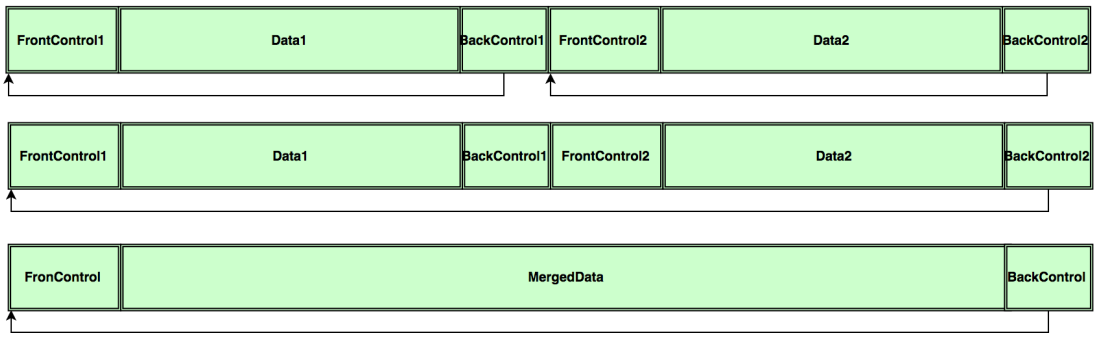
The block will consist of three logical parts:

• 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. Detailed Code Architecture (classes and methods)

2.1. TLS free-list multi-level allocator

The class implements the 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)
};
```

2.2. Lock-free queue

$LockFreeQueue < TElement >$ is obviously an implementation of lock-free queue. 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
};
```