Density Internals

Index

1 Introduction

This document describes the implementation of the library density[1]. The exposition of the functionalities is bottom-up, from the internal functionalities to the public data structures. The trivial parts are not covered, while in the critical parts the exposition reaches the source-line level of detail. The document is at the same time an overview of the library, more technical than the one available in the documentation. The exposition assumes a basic knowledge of non-blocking programming. On Wikipedia there is an excellent introduction to this topic[2].

Density is a C++11 library that provides:

- Page-based memory management: rather than allocating many small blocks of heap memory, all the data structures of the library prefer allocating large memory pages from a page allocator. All the pages have the same size (by default around 64 kibibytes). In the unlikely case that a single object is too big to fit in a page, it is allocated on the legacy heap.

- Lifo ordered allocation of heterogeneous objects, useful for thread-local temporary data. The library introduces the data-stack, a stack (parallel to the call-stack) in which the user can allocate dynamic arrays or raw buffers. The lifo memory management is built upon the paged-memory management, so the data stack is actually composed by memory pages. Allo-

1 https://github.com/giucamp/density

2 https://en.wikipedia.org/wiki/Non-blocking_algorithm

cating on the data-stack is fast almost like allocating a on the call-stack with `_alloca`, but never results in a stack overflows. In case of out of system memory an `std::bad_alloc` is thrown.

- Fifo ordered allocation of heterogeneous objects, useful for exchanging messages between threads, or for asynchronous processing in both single-thread and multi-thread scenarios. The library provides a rich set of heterogeneous queues and function queues:

- Concurrency with parametric progress guarantee (blocking, obstruction-free, lock-free and wait-free). The user specifies a progress guarantee with a parameter, and the implementation does its best to successfully complete the operation respecting it. In case that the implementation can't respect the guarantee, the call fails and has no observable side effects.

This is the summary of the heterogeneous queues and function queues provided by the library:

Threading strategy	Heterogeneous queue	Function queue	Producers cardinality	Consumers cardinality
single-thread	`heter_queue`	`function_queue`	none	none
locking	`conc_heter_queue`	`conc_function_queue`	multiple	multiple
lock-free	`lf_heter_queue`	`lf_function_queue`	single or multiple	single or multiple
spin-locking	`sp_heter_queue`	`sp_function_queue`	single or multiple	single or multiple

The current implementation of density does not differentiate between obstruction-free and lock-free guarantees: when an obstruction-free operation is requested, the library tries to complete it in lock-freedom.

All the data structures of the library provide the strong exception guarantee on all the operations.

Paged memory management

2 Overview of paged memory management

The library implements a page-based memory management. Grouping heterogeneous objects in large memory pages (instead of allocating each object in a separate heap block) improves the access locality, and increases the granularity of every memory management based task, including safe indirections in lock-free algorithms.

Memory pages have an arbitrary constant size and a constant alignment. The alignment must be greater or equal to the size (and of course an integer power of 2). Given an address of a byte within a page, we are able to compute the start address of the page with a simple bitwise and. Since the size of pages can be less than the alignment, page allocators can store some metadata in a page footer struct. As example, by default the page allocator of density manages pages aligned to `2^16`, with a page size of `2^16 - sizeof(AnInternalPageFooter)`. The page allocator must guarantee that

the space between the end of a page and the next aligned page in the address space will not be used by any other allocator. This guarantee will show very useful for the implementation of lifo memory management.

The data structures allocate many objects of heterogeneous types linearly in the same page and in the same order in which the user will access the objects (in container order). When the space in a page is exhausted, another page is allocated. In this case we say that a page switch occurs. A *page switch* occurs also when destroying\consuming objects, and a page has no more objects allocated within. *In-page* allocation and deallocation are considered the fast path in all the data structures, while a page switch is the slow path. The data structures of the library never cache free pages. The page allocator may possibly do that.

There is no limit on the size of objects to allocate, so in case of objects too large all data structures fallback to legacy heap allocation. For this reason allocator types in the library are required to model two distinct concepts: *PagedAllocator* and *UntypedAllocator*. The first will be exposed in this paragraph, while the latter is just a legacy allocator supporting over-alignment with offset.

To allow a safe access to pages in a lock-free context, the page allocator supports pinning. Pinning is a kind of reference counting: while a page is pinned (that is the reference count is non-zero), even if the page is deallocated, the allocator will not recycle it for an allocation function, and will not alter or read its content in any way.

Pinning a page that has been already deallocated is legal. Accessing such page, or doing anything but unpinning it, triggers an undefined behavior. When encountering a pointer with value P in a data structure, a thread should pin it, and then it should check whether P is still linked to the data structure. If not, it has to unpin and retry.

Sometimes we need that the pages we link to a data structure have a zeroed content. Lock-free algorithms may use initially zeroed pages to make threads agree on an history of the data which begins with the zeroed state. We introduce in our allocator functions to allocate pages with zeroed (rather than undefined) content.

When deallocating a page, we may tell to the allocator that the page is already zeroed. If the page is still pinned, the deallocating thread guarantees that the page will be zeroed when unpinned. This allows the allocator to return the page to the user as zeroed without having to memset it. The set of zeroed pages is not distinct from the set of normal pages: being zeroed is a transient property of a page.

Here is the PageAllocator synopsis:

```
class PageAllocator
{
public:
     static constexpr size_t page_size = ...;
     static constexpr size_t page_alignment = ...;

     static_assert(is_power_of_2(page_alignment)
                && page_alignment >= page_size, "");
```

```
    void * allocate_page();
    void * try_allocate_page(progress_guarantee) noexcept;
    void * allocate_page_zeroed();
    void * try_allocate_page_zeroed(progress_guarantee) noexcept;
    void deallocate_page(void *) noexcept;
    void deallocate_page_zeroed(void *) noexcept;
    void pin_page(void *) noexcept;
    void unpin_page(void *) noexcept;
};
```

Allocation functions return a pointer to the first byte of the page. Functions taking a page as parameter always align the address, so the user can specify a pointer to any byte within the page. The page allocator and many data structures of the library expose a set of *try_* functions*. These functions have some common properties:

- they return a boolean or a type implicitly convertible to boolean. A true return value indicates success, while false return value indicates a failure with no observable side effects.

- they are noexcept or at least exception neutral. Allocation try_* function are always noexcept, while put try_* functions don't throw any exception, though they pass through any exception raised by the constructor of an user defined type.

- the first parameter has always the type:

```
enum progress_guarantee {
      progress_blocking, progress_obstruction_free,
      progress_lock_free, progress_wait_free };
```

  If the implementation can't guarantee the completion with the specified progress guarantee, the function fails. A failure with a blocking progress guarantee generally indicates an out of memory.

Deallocation functions are required to be always wait-free. The rationale for this is that any lock-free CAS-based deallocation algorithm, whenever it does not succeed to perform the operation in a finite number of steps, can push the free page in a thread-local queue, and try later (or satisfy a subsequent request of the same thread).

The current implementation of lock-free and spin-locking queues has a defect related to page pinning. Page pinning and unpinning has an obvious implementation with atomic increment and decrement, that have a lock-free implementation on most modern architectures. Anyway, even when such operations are translated to a single instruction, it's unlikely that they can ever be wait-free. For this reason wait-free try_* operations are forced to fail when they would perform a page switch, even if a thread executes them in isolation.

This is a defect, since any wait-free operation shouldn't fail indefinitely if tried by a thread in isolation. The planned solution for this problem is to extend the PageAllocator concept with two more functions:

```
bool try_pin_page(progress_guarantee, void *) noexcept;
```

```
void unpin_page(progress_guarantee, void *) noexcept;
```

The first function is expected to try a possibly successful CAS-based increment in case the first argument is `progress_wait_free`, and to do an always successful atomic increment in all the other cases.

The second one can't fail by design, because the caller is unlikely to be able to handle the failure. In case of failure of a wait-free unpin, the allocator may push the page on a thread-local list of pages, so that it can re-try the unpin operation later.

3 SingletonPtr

Starting from C++11 the initialization of static local objects is thread safe. Since global objects exhibit the initialization order fiasco, static locals are generally safer than globals to implement singletons. Anyway, unless zero or constant initialization is possible, the compiler introduces a hidden atomic initialization guard, so that only the first access will initialize the object. The cpu's branch predictor greatly reduces the cost of this branch, but in many cases it can be removed at all.

The implementation of density uses an internal class template that implements the singleton pattern:

```
template <typename SINGLETON> class SingletonPtr;
```

This class template has non-nullable pointer semantics. It is stateless (therefore immutable), copyable and thread safe. The actual singleton object is handled internally, allocated in the static storage at a fixed address. Since this address is constant there is no need to store it, and all the specialization of `SingletonPtr` are guaranteed to be empty (no non-static data members).

The singleton is actually initialized when the first `SingletonPtr` is constructed. When the last `SingletonPtr` is destroyed, the singleton is destroyed too (it uses an internal reference count). The cost of handling the lifetime of the singleton is paid only by the constructor and the destructor. This is an advantage when we can keep a stable pointer to the singleton, and use repeatedly. For example an allocator class may keep a `SingletonPtr` pointing to a memory manager. Everyone who has access to a `SingletonPtr` can safely access the singleton.

Internally every specialization of `SingletonPtr` declares an instance of itself in the static storage, so that the singleton is always constructed during the dynamic initialization. If another `SingletonPtr` is constructed and destroyed before the initialization of this internal instance, the singleton is constructed, destroyed, and then constructed again. Similarly if a `SingletonPtr` is constructed after the destruction of the internal instance, the singleton is created again.

So `SingletonPtr` does not guarantee that only one instance of the singleton is created, but rather that in any moment at most one instance of the singleton exists. In case of contention between threads in the first access, if that happens during the dynamic initialization, a thread may spin-lock waiting for another thread to complete the initialization if the singleton. This is a limitation of the current implementation, and will be probably fixed in the next releases.

4 Raw atomics

Sometimes in lock free algorithms we can't use `std::atomic`, because even if standard atomics are trivially default-constructible, a default constructed atomic requires a call to `std::atomic_init` to complete the initialization. So we introduce a minimal set of non-standard of functions for atomic operations on fundamental variables:

```
template <typename TYPE>
     TYPE raw_atomic_load(
          TYPE const volatile * i_atomic,
          std::memory_order i_memory_order) noexcept;

template <typename TYPE>
     void raw_atomic_store(
          TYPE volatile * i_atomic,
          TYPE i_value,
          std::memory_order i_memory_order) noexcept;

template <typename TYPE>
     bool raw_atomic_compare_exchange_weak(
          TYPE volatile * i_atomic,
          TYPE * i_expected,
          TYPE i_desired,
          std::memory_order i_success,
          std::memory_order i_failure) noexcept;

template <typename TYPE>
     bool raw_atomic_compare_exchange_strong(
          TYPE volatile * i_atomic,
          TYPE * i_expected,
          TYPE i_desired,
          std::memory_order i_success,
          std::memory_order i_failure) noexcept;
```

The default value for memory order parameters is omitted, and it is always `std::memory_order_seq_cst`. The current implementation deletes all the generic function templates, and specialize an implementation only for some integer types.

These functions behave like the standard counterparts. Anyway fundamental variables are fully trivially constructible, and can be zero-initialized. A second advantage of raw atomics is that in some (rare) cases we can mix non-atomic and atomic writes to the same variable, if we know for sure that the non-atomic access is synchronized in some way.

The implementation of `lf_heter_queue` we will describe requires raw atomics for `uintptr_t`. If for a given compiler or OS raw atomics aren't available, the lock-free queues can't be used.

5 WF_PageStack

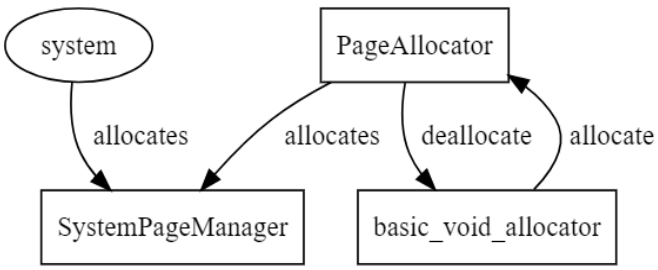
The implementation of the page allocator uses `WF_PageStack`, a wait-free queue specialized for pages. The stack is implemented as an intrusive linked list.

```
namespace density
{
    namespace detail
    {
        class WF_PageStack
        {
        private:
            std::atomic<PageFooter*> m_first{ nullptr };

        public:
            bool try_push(PageFooter * i_page) noexcept;
            bool try_push(PageStack & i_stack) noexcept;
            PageFooter * try_pop_unpinned() noexcept;
            PageStack try_remove_all() noexcept;
        };
    }
}
```

The class `PageStack` (used for some member functions) is a non-concurrent stack of pages. All the operations on `WF_PageStack` can fail in case of contention between threads. The function `try_pop_unpinned` temporary steals the whole stack, then search for the first page with zero pin count, and then re-links the pages (with possibly one less) to the stack. Stealing the whole content is an easy and elegant way to avoid the ABA problem.

6 Implementation of the page allocator



The page management is composed by 3 layers. The lowest layer is the `SystemPageManager`, that exposes a public function to allocate a memory page, but no function to deallocate. So it provides an irreversible allocation service.

Internally the `SystemPageManager` allocates large memory regions from the system and slices them into pages. Every region has a pointer to the next page to allocate. When a page is to be allocated, this pointer is advanced by the size of a page, and then its previous value is returned. When the current region is exhausted, another region is created. If the `SystemPageManager` gets an out of memory from the system when allocating a region, it tries to allocate a smaller region. After a number of failed tries with decreasing sizes, it reports the failure to the caller. Memory regions are deallocated when the `SystemPageManager` is destroyed, that is when the program exits.

The function of `SystemPageManager` that allocates a page is try_allocate_page:

```
void * SystemPageManager::try_allocate_page(
     progress_guarantee i_progress_guarantee) noexcept;
```

On success it returns a pointer to the first byte of the page, while in case of failure it returns null. The conditions under which this function fails depends on the requested progress guarantee, as illustrated in this table:

Progress guarantee	**Condition failure**
`progress_blocking`	Fails only in case of out of system memory
`progress_obstruction_free`	Fails if there is no free space in the memory regions already allocated, as the allocation of system memory is likely to be blocking.
`progress_lock_free`	
`progress_wait_free`	Fails if there is no free space in the memory regions already allocated or in case of contention between threads

The user of `SystemPageManager` may reserve in advance a capacity of memory to usable in lock-freedom with the following function:

```
uintptr_t SystemPageManager::try_reserve_region_memory(
     progress_guarantee i_progress_guarantee,
     uintptr_t const i_size) noexcept;
```

The return value is the total memory (in bytes) allocated from the system after the call. If this memory would be less than the one specified by the second argument, and the caller has specified the blocking progress guarantee, it tries to allocate memory regions from the system until the total size is equal or greater the the second parameter. Note: the signature of this function does not respect the convention on try_ functions. Anyway it is an internal function, and it may change in the future.

The second layer is the `PageAllocator`, that provides reversible page allocation, with an interface similar to the PageAllocator concept. In this layer there are a (currently fixed) number of slot, each containing a `WF_PageStack` of free pages and a `WF_PageStack` of free and zeroed pages.

Furthermore the `PageAllocator` associates to every thread some local data:

- A pointer to the *current slot*
- A pointer to the *victim slot*
- A non-concurrent stack of free pages
- A non-concurrent stack of free and zeroed pages

When a new page is requested, the `PageAllocator` tries to satisfy the request executing an ordered sequence of steps until one of them is successful. More precisely the `PageAllocator`:

- Peeks the first non-pinned page from the local stack
- Peeks the first non-pinned page from the current slot
- Steals all the pages of the victim slot, pushing them to the current slot. Then peeks the first non-pinned page from the stolen stack
- Tries to allocate from the `SystemPageManager` in wait-freedom (no new system memory is allocated)
- Loops the pointer to the victim slot through all the slot, stealing all the pages from each of them. If a non-pinned page in found in a stolen stack, the loop is interrupted.
- Now, if the progress guarantee specified by the caller is not `progress_blocking`, the `PageAllocator` returns null to signal a failure. Otherwise it forwards the request to the `SystemPageManager`, that may try to allocate a new memory region.

To deallocate a page the `PageAllocator` loops the pointer to the current slot through all the slot. At every iteration it tries to push the page on the `WF_PageStack`. The push may fail only because of contention with another thread. If the push fails on all the slots, the page is pushed on the thread-local stack.

The last layer of the page management is the `void_allocator`, that just forwards the requests to the `PageAllocator`. The layers before the `void_allocator` don't throw exceptions: they report a failure in allocating a page with a null return value. The `void_allocator` is the layer that, in case of failure of a non-try allocation function, throws a `std::bad_alloc`.

The user can use two functions of `void_allocator` e to reserve paged-memory to be used in lock-freedom:

```
static void reserve_lockfree_page_memory(size_t i_size,
     size_t * o_reserved_size = nullptr);

static bool try_reserve_lockfree_page_memory(
     progress_guarantee i_progress_guarantee,
     size_t i_size, size_t * o_reserved_size = nullptr) noexcept;
```

These functions make sure that the specified size is already been allocated from the system. In case of out of memory the first function throws an `std::bad_alloc`, while the second one, in case of failure, just returns false. The parameter `o_reserved_size` is to retrieve the total memory allocated from the system after the call.

Lifo memory management

7 The lifo allocator

The lifo memory management provided by the library is not thread safe, the reason being that it is supposed to be used mostly for thread-local data. It is built upon paged-memory management: the class template `lifo_allocator` adapts an allocator satisfying both he the PagedAllocator and UntypedAllocator concepts to expose a LIFO constrained allocation service.

The LIFO order imposes that only the most recently allocated block can be resized or deallocated. If this constraint is violated the behavior is undefined. Dealing with the LIFO order without the help of the RAII idiom is extremely bug prone, and it's highly discouraged.

This is the signature of the class template:

```
class lifo_allocator {
     template < typename UNDERLYING_ALLOCATOR = void_allocator,
     size_t ALIGNMENT = alignof(void*)>
          class lifo_allocator;
```

To simplify the implementation of the allocator the alignment is constant and the same for all the blocks.

The only non-static data member of is the a pointer to the end of the stack, that is the *top pointer*:

```
static constexpr uintptr_t s_virgin_top =
     uint_lower_align(page_alignment - 1, alignment);
uintptr_t m_top = s_virgin_top;
```

The top pointer usually points to the beginning of next block that will be allocated. It is an `uintptr_t` rather than a pointer to allow the default constructor to be constexpr (`reinterpret_cast` is not allowed in constant expressions).

The allocator is designed so that in the fast path, with the same conditional branch, it can redirect to the same slow execution path these 3 cases:

- the block to allocate is not too big, but the current page has not enough free space
- the block to allocate is too big, and it will not fit in a memory page
- the allocator is virgin, that is it has just been constructed and never used (after allocating a page the allocator will never return to the virgin state)

To allocate a block the allocator just adds the input size to the top pointer. If it detects that the updated top would lie past the end of the current page, it enters in the slow execution path. In the slow

path it decides between a new page or an external block (that is a block allocated in the legacy heap). When allocating or deallocating external blocks the state of the allocator is not altered.

When a new page is allocated, a header is added at the beginning of the page. This header contains only a pointer to the previous page, and it is necessary for the deallocation.

```
void * allocate(size_t i_size)
{
     DENSITY_ASSERT(i_size % alignment == 0);

     auto const new_top = m_top + i_size;
     auto const new_offset = new_top -
          uint_lower_align(m_top, page_alignment);
     if (!DENSITY_LIKELY(new_offset < page_size))
     {
          // page overflow
          return allocate_slow_path(i_size);
     }
     else
     {
          // advance m_top
          DENSITY_ASSERT_INTERNAL(i_size <=page_size);
          auto const new_block = reinterpret_cast<void*>(m_top);
          m_top = new_top;
          return new_block;
     }
}

DENSITY_NO_INLINE void * allocate_slow_path(size_t i_size)
{
     DENSITY_ASSERT_INTERNAL(i_size % alignment == 0);
     if (i_size < page_size / 2)
     {
          // allocate a new page
          auto const new_page =
               UNDERLYING_ALLOCATOR::allocate_page();
          DENSITY_ASSERT_INTERNAL(new_page != nullptr);
          auto const new_header = new(new_page) PageHeader;
          new_header->m_prev_page = reinterpret_cast<void*>(
               m_top);
          m_top = reinterpret_cast<uintptr_t>(
               new_header + 1) + i_size;
          return new_header + 1;
     }
     else
     {
          // external block
          return UNDERLYING_ALLOCATOR::allocate(
               i_size, alignment);
     }
}
```

The function allocate requires that the size of the block is aligned, otherwise the behavior is undefined. `allocate` can allocate a block with size zero. Anyway there is a special allocation function that can be used when the size of the block is always zero:

```
void * allocate_empty() noexcept
{
     return reinterpret_cast<void*>(m_top);
}
```

This function does not alter the state of the queue, is faster and never throws.

The allocator uses a delayed deallocation strategy for the pages: a page is not deallocated when it owns no alive blocks, but rather when a block of the previous page is deallocated. With this strategy a user that occasionally allocates and deallocates a block will enter in the slow path (and will allocate a page) only the first time.

To deallocate a block, if the address is in the same page of the top pointer, the `lifo_allocator` just assigns the block to deallocate to the top pointer. Otherwise it enters the slow path, in which the allocator detects whether the top pointer is going to leave an empty page, or it has to deallocate an external block.

```
void deallocate(void * i_block, size_t i_size) noexcept
{
     DENSITY_ASSERT(i_block != nullptr && i_size % alignment == 0);

     // this check detects page switches and external blocks
     if (!DENSITY_LIKELY(same_page(i_block,
               reinterpret_cast<void*>(m_top))))
     {
          deallocate_slow_path(i_block, i_size);
     }
     else
     {
          m_top = reinterpret_cast<uintptr_t>(i_block);
     }
}

DENSITY_NO_INLINE void deallocate_slow_path(
     void * i_block, size_t i_size) noexcept
{
     DENSITY_ASSERT_INTERNAL(i_size % alignment == 0);

     if ((m_top & (page_alignment - 1)) == sizeof(PageHeader) &&
               same_page(reinterpret_cast<PageHeader*>(
               m_top)[-1].m_prev_page, i_block) )
     {
          // deallocate the top page
          auto const page_to_deallocate = reinterpret_cast<void*>(
               m_top);
          UNDERLYING_ALLOCATOR::deallocate_page(
               page_to_deallocate);
```

```
            m_top = reinterpret_cast<uintptr_t>(i_block);
      }
      else
      {
            // external block
            UNDERLYING_ALLOCATOR::deallocate(
                  i_block, i_size, alignment);
      }
}
```

The allocator allows resizing a block, preserving its content up to the previous size, with the following function:

```
void * reallocate(void * i_block, size_t i_old_size, size_t i_new_size);
```

The implementation of this function is slightly more complex, but it is basically an exception safe mixing of `allocate` and `deallocate`, so it is not listed here.

8 The data-stack

The library keeps a private thread-local instance of `lifo_allocator`, called the *data-stack*. By design the constructor of `lifo_allocator` is trivial and constexpr. Threads may allocate the first page only when they uses the data-stack for the first time. Direct access to the data-stack is not allowed. Two data structures are provided instead to use indirectly the data-stack: `lifo_array` and `lifo_buffer`.

The class template `lifo_array` is very similar to an array. It has an immutable size specified at construction time. If no initializer is provided, `lifo_array` default-constructs the elements. This is a big difference with to `std::vector`, that uses value-initialization.

```
    // uninitialized array of doubles
    lifo_array<double> numbers(7);

    // initialize the array
    for (auto & num : numbers)
        num = 1.;

    // compute the sum
    auto const sum = std::accumulate(
           numbers.begin(), numbers.end(), 0.);
    assert(sum == 7.);

    // initialized array
    lifo_array<double> other_numbers(7, 1.);
    auto const other_sum = std::accumulate(
           other_numbers.begin(), other_numbers.end(), 0.);
    assert(other_sum == 7.);

    // array of class objects (default-constructed)
    lifo_array<std::string> strings(10);
```

```
bool all_empty = std::all_of(strings.begin(), strings.end(),
  [](const std::string & i_str) {
              return i_str.empty(); });
assert(all_empty);
```

It's highly recommended to use `lifo_array` only on the automatic storage. Doing so there is no way to break the LIFO constraint. Any other used should be handled with caution.

The class `lifo_buffer` is very different from `lifo_array` mainly for 2 reasons:

- it handles raw memory rather than elements of a type known at compile-time
- it allows resizing (with content preservation) of the most recently constructed instance.

Resizing a `lifo_buffer` when another most recent `lifo_buffer` or `lifo_array` is alive causes undefined behavior.

Internally the default-contructor of `lifo_buffer` uses the function `allocate_empty` to allocate a block of size 0, so it is noexcept and very fast.

```
void func(size_t i_size)
{
    using namespace density;

    lifo_buffer buffer_1(i_size);
    assert(buffer_1.size() == i_size);

    lifo_buffer buffer_2; /* now buffer_1 can't be
            resized until buffer_2 is destroyed */
    assert(buffer_2.size() == 0);

    auto mem = buffer_2.resize(sizeof(int));
    assert(mem == buffer_2.data());
    *static_cast<int*>(mem) = 5;

    mem = buffer_2.resize(sizeof(int) * 20);
    assert(*static_cast<int*>(mem) == 5);

    lifo_array<int> other_numbers(7);
    // buffer_2.resize(20); ← other_numbers is more recent, so this
                        would be a violation of the lifo constraint!
}
```

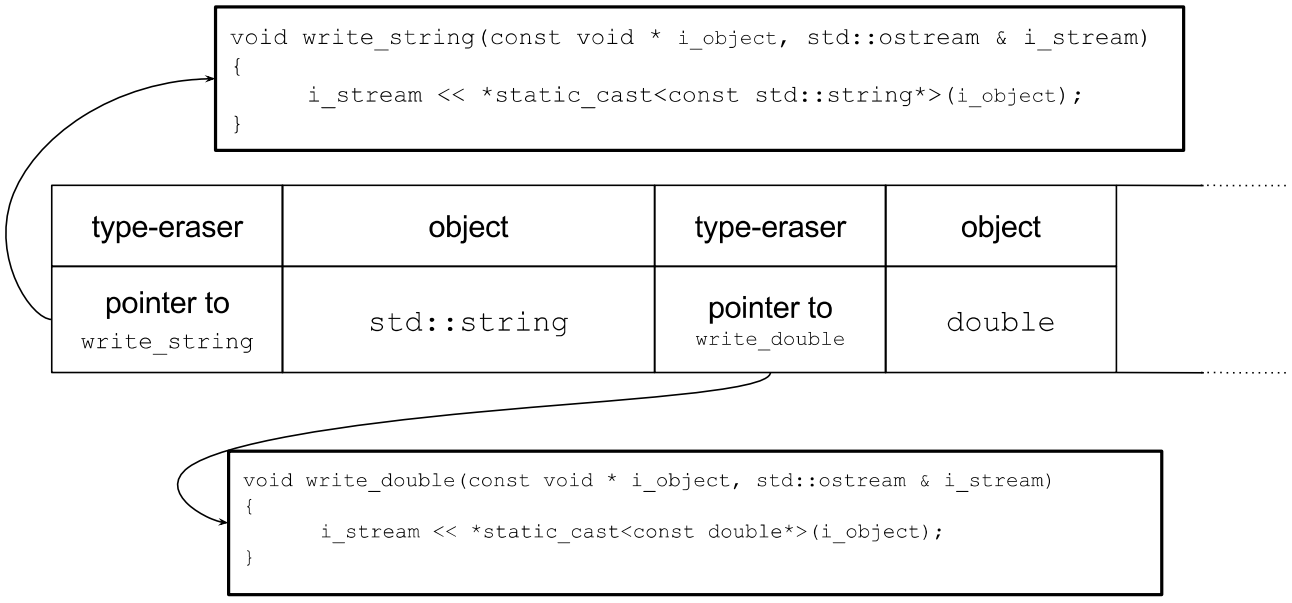
Heterogeneous and function queues

9 Introduction to type erasure

Type erasure is a well know technique to decouple actions on objects from their compile-time type. Let's consider an implementation of a heterogeneous sequence of objects. When we insert an object in the sequence we may know the type at compile-time, so we can just call the constructor of the element. Anyway, when we later iterate the sequence, we have no knowledge of the types of the objects. Type erasure solves this problem.

The type erasure used by density requires declaring in advance which features should be captured the types. Let's suppose for example that we need to be able to write the objects of the sequence to an `std::ostream`. A very simple way to do that is to prepend to every element a pointer to a function that is able to handle the specific object.

The memory layout of this sequence may be something like this:



In this example the type eraser is a pointer to a function with this signature:

```
void (*)(const void * i_object, std::ostream & i_stream);
```

A trivial consideration is that the implementation of the write function for any type may be provided by a function template:

```
template <typename T>
	void write_string(const void * i_object, std::ostream & i_stream)
{
	i_stream << *static_cast<const T*>(i_object);
}
```

In a more realistic case we would need to add at least a pointer to a destroy function, so in this case our type erasure would be a struct containing two pointers to functions. We may even add a variable with the size of the type, and one with the alignment. If our type erasure has a considerable size, we may consider to add a level of indirection: the type eraser becomes a pointer to a static consteprx struct with the actual pointer to functions and data.

The above example is simplified, but is very similar to the type erasure used by density. Note that this technique is similar to the v-table used by the C++ compilers to implement virtual functions. The main differences are:

- it is easily customizable. For example one may add data members (for example for the size and alignment), and even pointer to construction functions, while virtual static data and virtual constructors are currently not supported by the standard.

- it is not intrusive, while virtual function are a feature embedded in the type. A class can be type-erased without being modified.

Here follows a digression on some benefits of using type erasure. The reader not interested may skip to the next paragraph.

Let's suppose we have a set of classes representing 2d shapes:

```
class Circle
{
public:

    bool IsPointInside(const Vector2d & i_point) const;

private:
    Vector2d m_center;
    float m_radius = -1.f;
};

class Box
{
public:

    bool IsPointInside(const Vector2d & i_point) const;

private:
    Vector2d m_center, m_size{-1.f, -1.f};
};
```

Now we want to implement a class `ComplexShape`, that represent a union of other shapes. `ComplexShape` provides a function `IsPointInside` that checks if the point is inside any of the child shapes. The classic OOP solution would be introducing a base class or an interface representing a shape, with a virtual function `IsPointInside`. Anyway interfaces and base classes have impact on both the design and the performances. Any instance of `Box` would have a pointer to a v-table as hidden member, even if polymorphous is not necessary for all the uses.

Instead of using polymorphism, we may implement `ComplexShape` using type erasure:

```
class ShapeReference
{
public:

    // ...

    bool IsPointInside(const Vector2d & i_point) const
    {
        return (*m_type->m_is_point_inside)(m_object, i_point);
    }

private:
    struct Type
    {
        bool (*m_is_point_inside)(void * const i_object,
                      const Vector2d & i_point);
        void (*m_destroy_func)(void * i_object);
    };
    void * m_object;
    Type * m_type;
};

class ComplexShape
{
    // ...
private:
    std::vector<ShapeReference> m_children;
};
```

With type-erasure, polymorphism is paid only when it's actually used.

10 The RuntimeType pseudo-concept

The heterogeneous queues provided by the library are not bound to a particular type-erasure. They have instead a type template-parameter that allows the user to specify a custom type erasure, provided that it models *RuntimeType*.

Here is the RuntimeType synopsis:

```
class RuntimeType
{
public:

    using common_type = ...;

    template <typename TYPE>
        static RuntimeType make() noexcept;
```

```
    template <typename TYPE>
        bool is() const noexcept;

    RuntimeType(const RuntimeType &);

    size_t size() const noexcept;

    size_t alignment() const noexcept;

    common_type * default_construct(void * i_dest) const;

    common_type * copy_construct(void * i_dest,
           const common_type * i_source) const;

    common_type * move_construct(void * i_dest,
           common_type * i_source) const;

    void * destroy(common_type * i_dest) const noexcept;

    const std::type_info & type_info() const noexcept;

    bool are_equal(const common_type * i_first,
           const common_type * i_second) const;

    bool operator == (
           const RuntimeType & i_other) const noexcept;

    bool operator != (
           const RuntimeType & i_other) const noexcept;
};
```

Most of the functions are optional, so RuntimeType strictly can't be a concept.

The static function template make makes the transition from a type known at compile-time, to an instance of RuntimeType that has no compile-time dependence from the type. The type bound to a RuntimeType is the *target type*. If the type alias `common_type` is void, the target type can be any type. Otherwise the target type must be a user-defined type deriving from `common_type`.

The `*_contruct` functions take as parameter a pointer to the beginning of the storage of the object to construct (`i_dest`). If `common_type` is `void`, they return the same pointer. Otherwise they return a pointer to the `common_type` sub-object of the complete object. Symmetrically the function `destroy` takes a pointer to the `common_type` sub-object of the object, and returns a pointer to the complete object.

By default all the heterogeneous queues of the library use the class template `runtime_type` for type erasure. A `runtime_type` internally is just a pointer to a custom v-table containing data and pointer to functions to capture a set of features on the target type. This is the signature of the class template:

```
template <typename COMMON_TYPE = void,
  typename FEATURE_LIST =
```

```
     type_features::default_type_features_t<COMMON_TYPE> >
          class runtime_type;
```

`runtime_type` is highly customizable, as it allows to specify the set of feature to capture with the second template argument, which is a type list. The namespace `density::type_features` provides many common features that can be used. This table enumerates the features that a `runtime_type` captures with the default feature list:

Feature	What allows on **`runtime_type`**
`type_features::size`	Allows to use the function `runtime_type::size` to get the size of the target type
`type_features::alignment`	Allows to use the function `runtime_type::alignment` to get the size of the target type
`type_features::rtti`	Allows to use the function `runtime_type::type_info` to get a `std::type_info` for the target type
`type_features::destroy`	Allows to use the function `runtime_type::destroy` to destroy an instance of the target type
`type_features::move_construct` (only if the `comon_type` is `void` or is move constructible)	Allows to use the function `runtime_type::move_construct` to move-construct an instance of the target type from an instance of the same type
`type_features::copy_construct` (only if the `comon_type` is `void` or is copy constructible)	Allows to use the function `runtime_type::move_construct` to copy-construct an instance of the target type from an instance of the same type

This example shows how to define a `runtime_type` customizing the captured features and how to use it:

```
using namespace type_features;
using MyRTType = runtime_type<void,
     feature_list<default_construct, destroy, size> >;
MyRTType type = MyRTType::make<std::string>();
void * buff = malloc(type.size());
type.default_construct(buff);
// now buff points to a valid std::string
*static_cast<std::string*>(buff) = "hello world!";
type.destroy(buff);
free(buff);
```

11 Overview of the heterogeneous queues

The library provides 4 heterogeneous queues:

- *heter_queue*: not thread safe

- *conc_heter_queue*: thread safe, blocking
- *lf_heter_queue*: thread safe, lock-free
- *sp_heter_queue*: thread safe, spin-locking

The first 3 template parameters of all the heterogeneous queues are:

- `COMMON_TYPE`. type parameter, by default `void`. An element of type `T` can be put in the queue only if `T*` is implicitly convertible to `COMMON_TYPE*`.
- `RUNTIME_TYPE`. type parameter, by default `runtime_type<>`. It's the type-eraser. It must have the member type alias `common_type` that matches `COMMON_TYPE`. A static assert checks that this requirement is met.
- `ALLOCATOR_TYPE`. type parameter, by default `void_allocator`. It must model both PageAllocator and UntypedAllocator.

The class template `lf_heter_queue` has these additional template parameters:

- `PRODUCER_CARDINALITY`. parameter of type `concurrency_cardinality`, the default value is `concurrency_multiple`. Specifies whether multiple threads are allowed to put in the queue concurrently.
- `CONSUMER_CARDINALITY`. parameter of type `concurrency_cardinality`, the default value is `concurrency_multiple`. Specifies whether multiple threads are allowed to consume from the queue concurrently.
- `CONSISTENCY_MODEL`. parameter of type `consistency_model`, the default value is `consistency_sequential`. Specifies whether the queue is linearizable, that is all the threads agree on a total ordering of all the operations on the queue.

The class template `sp_heter_queue` has these additional template parameters:

- `PRODUCER_CARDINALITY`. Like the homonymous parameter of `lf_heter_queue`.
- `CONSUMER_CARDINALITY`. Like the homonymous parameter of `lf_heter_queue`.
- `BUSY_WAIT_FUNC`. Type callable with the signature `void ()` that is called in the body of a busy wait. The default value is a function type that calls `std::this_thread::yield`.

An action on the queue is an *operations*. Some operation is executed by a single function call, while some others span more than one call. Every operation belongs to one of these classes:

- *put operations*: operations that inserts an element to the end of the queue.

- *consume operations*: operations that consume an element at the beginning of the queue, or that check if the queue is empty.

- *lifetime operations*: all the other functions, including constructors, destructors, assignments and swap.

Depending on the queue and on the template arguments, put and consume operations can be used concurrently. All the queues define a set of member constants that describe the concurrent capabilities of the queue:

Member constant	**Semantics**
`static constexpr bool concurrent_puts;`	Whether multiple threads are allowed to do put operations concurrently.
`static constexpr bool concurrent_consumes;`	Whether multiple threads are allowed to do consume operations concurrently.
`static constexpr bool concurrent_put_consumes;`	Whether put operations can be executed concurrently with consume operations.
`static constexpr bool is_seq_cst;`	Whether the queue is linearizable, that is all the threads agree on a total ordering of all the operations on the queue.

`heter_queue` is not concurrent, so all the above constants are `false`. `conc_heter_queue` is always fully concurrent, so all the above constants are `true`.

`lf_heter_queue` and `sp_heter_queue` always allows a single producer to run concurrently with a single consumer, so `concurrent_put_consumes` is always `true`. The values of `concurrent_puts` and `concurrent_consumes` depends on the template arguments `PRODUCER_CARDINALITY` and `CONSUMER_CARDINALITY`. The constant `is_seq_cst` is always true for `sp_heter_queue`, and depends on the template argument `CONSISTENCY_MODEL` for `lf_heter_queue`.

The simplest way to put an element in a queue is using `push` or `emplace`:

```
queue.push(19); // the parameter can be an l-value or an r-value
queue.emplace<std::string>(8, '*'); // pushes "********"
```

The type of the element is deduced from the argument in the case of `push`, but needs to be specified explicitly in the case of `emplace`.

Put functions containing `start_` in their name are *transactional*. They start a put, and return an object of type `put_transaction`, that can be used to *commit* or *cancel* the put.

```
auto put = queue.start_push(12);
```

```
put.element() += 2;
put.commit(); // commits a 14
```

A put transaction becomes observable only when it's committed. If it is canceled, it will never be observable.

An instance of `put_transaction` in any moment either is bound to a transaction, or is empty. When the functions `commit` or `cancel` are called, the `put_transaction` becomes empty. If the `put_transaction` is already empty, calling `commit` or `cancel` causes undefined behavior. When a non-empty `put_transaction` is destroyed, the transaction is canceled. `put_transaction`'s are movable but not copyable.

Transactional puts allow to associate a *raw block* with the element:

```
heter_queue<> queue;
struct MessageInABottle
{
    const char * m_text = nullptr;
};
auto transaction = queue.start_emplace<MessageInABottle>();
transaction.element().m_text = transaction.raw_allocate_copy("Hello!");
transaction.commit();
```

A raw block is an untyped and uninitialized range of contiguous memory, in which the user may store additional data associated to an element. An element may be associated to more than one block, but it should keep a pointer to every of them, because otherwise consumers would have no way to get these addresses.

Raw blocks can be allocated specifying a size and an alignment, a pair of iterators, or a range (like in the example). A raw block is automatically deallocated when the put of the associated element is canceled, or when the element is consumed. Accessing a deallocated raw block causes undefined behavior.

By default operations on queues are non-reentrant. During a non-re-entrant operation the queue is not in a consistent state for the calling thread (it is in a consistent state for the other threads, provided that it is concurrency-enabled). So, in the example above, calling any member function on the queue between the `start_emplace` and the `commit` would cause undefined behavior. Even a single-call put may introduce re-entrancy:

```
// buggy code:
queue.emplace<ClassThatUsesTheQueueInTheConstructor>(12); <- UB!!!!
```

The operations that starts with a function containing `reentrant_` in their name support *re-entrancy*. Re-entrant operations can overlap each other.

```
auto put_1 = queue.start_reentrant_push(12);
auto put_2 = queue.start_reentrant_push(std::string("Hello "));
auto put_3 = queue.start_reentrant_push(3.14f);
put_3.commit();
put_1.commit();
put_2.element() += "world!!!!";
```

```
put_2.cancel();
```

Non-re-entrancy allows to the implementation some optimizations and may alter thread synchronization. For example `heter_queue` starts non-re-entrant puts in the committed state, so that the actual `commit` becomes a no-operation (so the observable state of the queue is temporary wrong). A notable further example is `conc_heter_queue`, that keeps its internal non-recursive mutex locked during a non-re-entrant operation, and locks it twice for re-entrant operations. If the user starts an operation while a non-re-entrant one is still in progress, the mutex would be locked twice by the same thread, causing undefined behavior.

Put functions containing `dyn_` in their name are *dynamic*. Dynamic puts inserts elements of a type unknown at compile-time. The type of the element is rather specified by an instance of the type-eraser, passed as first argument. The element can be default-constructed, copy-constructed or move-constructed, provided that the type-eraser supports it.

```
using namespace type_features;
using MyRunTimeType = runtime_type<void, feature_list<default_construct,
destroy, size, alignment>>;
heter_queue<void, MyRunTimeType> queue;
auto const type = MyRunTimeType::make<int>();
queue.dyn_push(type); // appends 0
```

One possible use of dynamic puts is for transferring one element from one queue to another, as shown in the following example:

```
using namespace type_features;
using MyRunTimeType = runtime_type<void, feature_list<
     move_construct, destroy, size, alignment>>;

heter_queue<void, MyRunTimeType> queue_1;
queue_1.emplace<std::string>("Hello!");

heter_queue<void, MyRunTimeType> queue_2;
auto consume = queue_1.try_start_consume();
queue_2.dyn_push_move(consume.complete_type(), consume.element_ptr());
consume.commit();

consume = queue_2.try_start_consume();
assert(consume && consume.complete_type().is<std::string>());
assert(consume.element<std::string>() == "Hello!");
consume.commit();

assert(queue_1.empty() && queue_2.empty());
```

Put functions containing `try_` in their name allows to specify the progression guarantee as first parameter, and returns a (possibly empty) `put_transaction`. Try put functions are supported only by `lf_heter_queue` and `sp_heter_queue`.

The set of put functions is orthogonal: every combination of transnational, re-entrant, dynamic and try_* is supported. This is a summary of the complete set:

- `[try_][start_][reentrant_]push`
- `[try_][start_][reentrant_]emplace`
- `[try_][start_][reentrant_]dyn_push`
- `[try_][start_][reentrant_]dyn_push_copy`
- `[try_][start_][reentrant_]dyn_push_move`

A *consume operation* is very similar to a put-transaction, but it's not a transaction. When the consume starts, the element is immediately removed by the queue, so that other consumers will not interfere. Whenever the consume operation is committed, the element is destroyed, and it's gone forever. Anyway if the consume is canceled the element will reappear in the queue, so that the consume can be retried. This is the reason why consumes are not transactions: even when canceled, they have the observable effect of temporary removing the element.

```
    heter_queue<> queue;
    auto consume_1 = queue.try_start_consume();
    assert(!consume_1);
    queue.push(42);
    auto consume_2 = queue.try_start_consume();
    assert(consume_2);
    assert(consume_2.element<int>() == 42);
    consume_2.commit();
```

The following example shows how consumers can analyze the element and its type. Note that if the same action is needed for all the elements regardless of they type, a type feature would be a much better solution (there also is a built-in `type_feature::ostream` that can be used with `runtime_type`).

```
heter_queue<> queue;
queue.push(42);
queue.emplace<std::string>("Hello world!");
queue.push(42.);

while (auto consume = queue.try_start_consume())
{
    if(consume.complete_type().is<int>())
        std::cout << "Found an int: "
           << consume.element<int>() << std::endl;
    else if(consume.complete_type().is<std::string>())
        std::cout << "Found a string: "
           << consume.element<std::string>() << std::endl;
    consume.commit();
}
```

We may rewrite the consume loop in a similar way that may allow a more efficient implementation:

```
heter_queue<>::consume_operation consume;
while (queue.try_start_consume(consume))
{
```

```
    ...
    consume.commit();
}
```

12 Anatomy of a queue

The storage of a heterogeneous queue is an ordered set of pages and possibly a set of legacy memory blocks. The first page is the *head page*, while the last is the *tail page*. The queue has two pointers: the *head pointer*, that points to a byte of the head page, and the *tail pointer*, that point to a byte of the tail page.

A *value* is alive if it contains a valid element. The element of an alive value must be destroyed soon or later. A value is dead if it does not contain a valid element. Dead values arise from:

- elements whose put was canceled (including those whose constructor threw an exception).
- elements that have been consumed, but still have a storage
- raw blocks

The layout of a value is composed by:

• an instance of *ControlBlock*, an internal structure of the library. It is always present, and holds the pointer '*next*' and the state flags of the value.

• optionally the run-time type object, not present for raw allocations.

• optionally the element or the raw block

If the queue is not empty, the head pointer points to the first value of the queue. Values are stored as a forward linked-list, and are allocated linearly, in container order, with an implementation-defined granularity (which is the alignment guaranteed for the storage of the value).

If the queue is fully heterogeneous (that is the template argument `COMMON_TYPE` is `void`), a control block is composed only by an `uintptr_t`, that stores the pointer 'next' and some control flags in the least significant bits. If the queue is partially heterogeneous (that is the template argument `COMMON_TYPE` is not `void`), the control block includes a pointer to the element. This pointer is necessary because up-casting a pointer in C++ is a non-trivial operation.

From now on we will describe the memory layout and representation of an `lf_heter_queue`. The details of the other queues are not covered because:

- `heter_queue` has an extremely similar layout, but a much simpler implementation.
- `conc_heter_queue` is just a wrapper of an internal `heter_queue` protected by an `std::mutex`.

- `sp_heter_queue` shares much of it implementation with `lf_heter_queue`. The part of implementation not in common will described at the end.

The bottom of every page is reserved to the *end-of-page* control block. This block is allocated always at the same offset from the beginning of the page. Excluding this block and the first of the page, all control blocks are allocated at variable offsets.

In the end-of-page control block, the pointer to the next control block always points to another page. In all other cases, the pointer to the next control block points to the same page.

There is an implementation defined limitation on the size and alignment of elements that can be allocated in a page. Whenever an element (or raw block) is too big, the queue inserts in the page just a pointer to a legacy heap memory block used as storage for the element.

The least significant bits of `m_next` are used to store the control flags:

```
namespace detail
{
    template<typename COMMON_TYPE> struct LfQueueControl
    {
        uintptr_t m_next; // raw atomic
        COMMON_TYPE * m_element;
    };

    template<> struct LfQueueControl<void>
    {
        uintptr_t m_next; // raw atomic
    };

    enum NbQueue_Flags : uintptr_t
    {
        NbQueue_Busy = 1,
        NbQueue_Dead = 2,
        NbQueue_External = 4,
        NbQueue_InvalidNextPage = 8,
        NbQueue_AllFlags = NbQueue_Busy | NbQueue_Dead |
NbQueue_External | NbQueue_InvalidNextPage
    };
    ...
```

The flag ‘Busy’ is set to a value while a producer is producing it, or while a consumer is consuming it. The flag ‘Dead’ is set on values not alive. Consumers search for m_next that are not zeroed, and don’t have heither ‘Busy’ or ‘Dead’.

While an element is being produced (before commit is called on the put transaction), the element is busy, and not observable. If commit is not called, no one will ever observe any part of that element.

When a thread start consuming an element, it sets the flag ‘Busy’ on the element. If the consume operation is committed, the element is gone forever. Otherwise, if the consume operation is canceled, the element reappears in the queue (the busy flag is cleared). Since other consumers have ob-

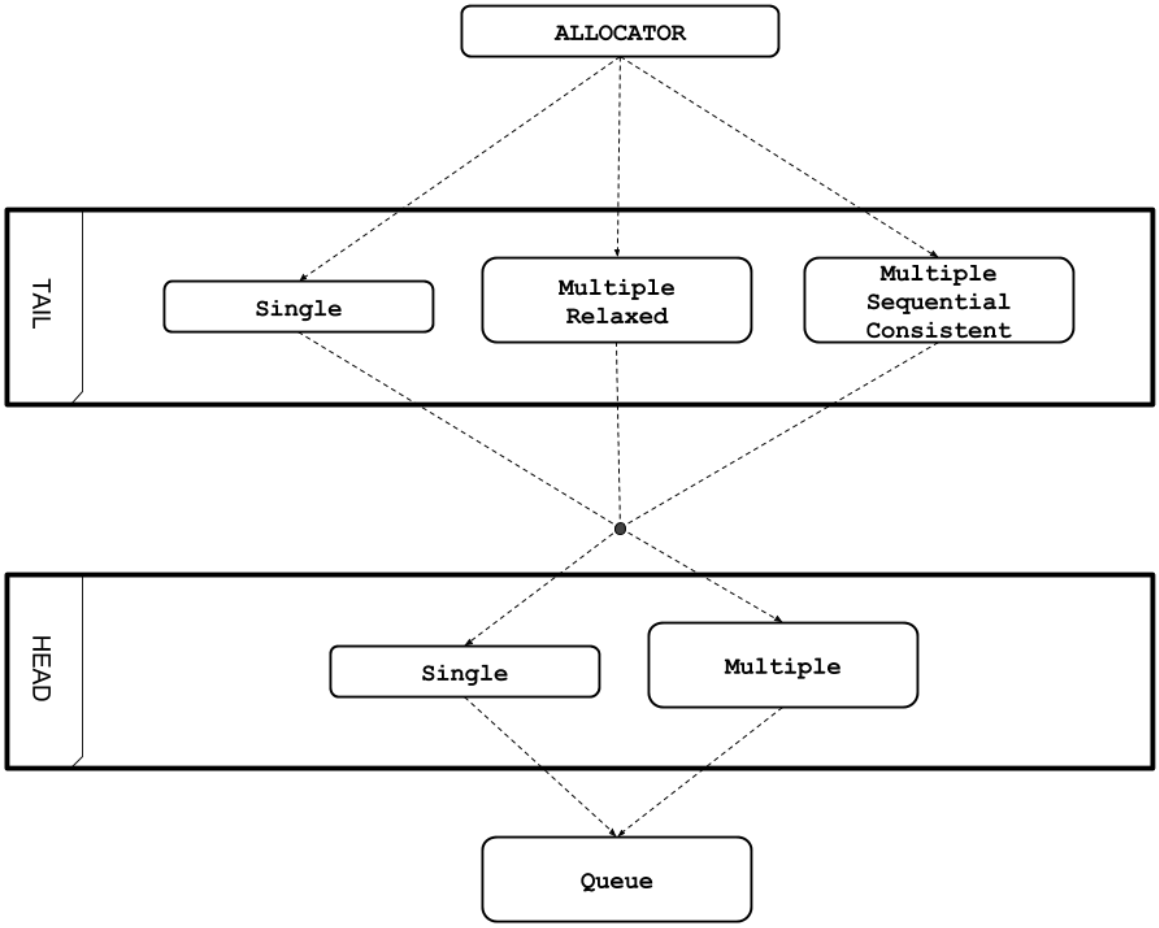
served the absence of the element between the beginning of the consume and the cancel, the consume is not a transaction.

1 Derivation chain

We are implementing the following class template:

```
template < typename COMMON_TYPE = void,
    typename RUNTIME_TYPE = runtime_type<COMMON_TYPE>,
    typename ALLOCATOR_TYPE = void_allocator,
    concurrent_cardinality PROD_CARDINALITY = concurrency_multiple,
    concurrent_cardinality CONSUMER_CARDINALITY = concurrency_multiple,
    consistency_model CONSISTENCY_MODEL = consistency_sequential>
        class lf_heter_queue;
```

To handle such complexity, `lf_heter_queue` has a four-level private hierarchy.



The queue is internally a null-terminated linked-list, and consumers can iterate it without accessing the tail for the termination condition. So the tail pointer is visible only to producers, and the head pointer is visible only to consumers. Beyond slightly less contention, this simplifies the implementation, and allows to better exploiting reduced cardinalities: if `PROD_CARDINALITY` is `concurrency_single`, the tail pointer is not an atomic variable. Similarly, if `CONSUMER_CARDINALITY` is `concurrency_single`, the head pointer is not an atomic variable.

2 Put operations

The first level is the allocator. In the second level we implement the tail pointer and the put operations in a class template:

```
namespace detail
{
   template < typename COMMON_TYPE, typename RUNTIME_TYPE,
      typename ALLOCATOR_TYPE,
      concurrent_cardinality PROD_CARDINALITY,
      consistency_model CONSISTENCY_MODEL >
         class LFQueue_Tail : protected ALLOCATOR_TYPE;
```

The general class template is not defined. The actual implementation is split in 3 template specializations:

One for PROD_CARDINALITY = concurrency_single. This specialization allocates non-zeroed pages, and does not demand zeroing-before-deallocation to consumers.

One for PROD_CARDINALITY = concurrency_multiple and CONSISTENCY_MODEL = consistency_relaxed. This specialization allocates zeroed pages, and does allow (but does not require) zeroing the memory at consume time. This is an optimization: consumers probably have that memory in their cache memories.

One for PROD_CARDINALITY = concurrency_multiple and CONSISTENCY_MODEL = consistency_sequential. This specialization allocates zeroed pages, but does not allow zeroing the memory at consume time, because this would break the algorithm. In this case the cost of zeroing is up to the allocator, since zeroed pages are allocated, but non-zeroed pages are deallocated.

Every specialization of LFQueue_Tail provides this const boolean to communicate to the subsequent layers whether pages should be zeroed at consume time:

constexpr static bool s_deallocate_zeroed_pages = ...;

Queues that require zeroed pages return zeroed pages to the allocator. While this is not mandatory, it helps the allocator, that would have to write the whole content of the page before reusing it. In contrast the queue can zero the memory while it is probably still in the cache.

The constructor of LFQueue_Tail looks the same in all 3 specializations:

```
LFQueue_Tail() noexcept
: m_tail(invalid_control_block()), m_initial_page(nullptr)
{
}
```

We delay the allocation of the first page, so that the default constructor and the move constructor can be noexcept. The value invalid_control_block() is such that it will always cause a page overflow in the first put. When the first page is allocated, it is set to m_initial_page, so that the consume layer can read it during its delayed initialization. After being set to the first allocated page, m_initial_page does not change anymore, even when the page it points to is deallocated.

The code below is the allocation function for an element or raw block in the case of single producer (and non-atomic m_tail).

```
// LFQueue_Tail<..., concurrency_single>::inplace_allocate
Block inplace_allocate(uintptr_t i_control_bits,
bool i_include_type, size_t i_size, size_t i_alignment)
{
     if (i_alignment < min_alignment)
     {
          i_alignment = min_alignment;
          i_size = uint_upper_align(i_size, min_alignment);
     }

     auto tail = m_tail;
     for (;;)
     {
          // allocate space for the control block
          void * address = address_add(tail,
i_include_type ? s_element_min_offset : s_rawblock_min_offset);

          // allocate space for the element
          address = address_upper_align(address, i_alignment);
          void * const user_storage = address;
          address = address_add(address, i_size);
          address = address_upper_align(address, s_alloc_granularity);
          auto const new_tail = static_cast<ControlBlock*>(address);

          // check for page overflow
          auto const new_tail_offset = address_diff(new_tail,
address_lower_align(tail, ALLOCATOR_TYPE::page_alignment) );
          if (DENSITY_LIKELY(new_tail_offset <= s_end_control_offset))
          {
               // Note: not an atomic store
               new_tail->m_next = 0;
```

```
                auto const control_block = tail;
                auto const next_ptr =
reinterpret_cast<uintptr_t>(new_tail) + i_control_bits;

                raw_atomic_store(&control_block->m_next,
next_ptr, detail::mem_release);

                m_tail = new_tail;
                return { control_block, next_ptr, user_storage };
           }
           else if (i_size + (i_alignment - min_alignment) <=
s_max_size_inpage)
           {
                tail = page_overflow(tail);
           }
           else
           {
                /* this allocation would never fit in a page,
allocate an external block */
                return external_allocate(i_control_bits, i_size,
i_alignment);
           }
     }
}
```

The function LFQueue_Tail::inplace_allocate is the core of all put operations, and is the interface for subsequent layers. It is still a private and low-level function.

This is a good time to make 2 considerations valid for all 3 kinds of tail we are implementing:

We test for the page overflow in the domain of page offset to avoid having to handle the pointer arithmetic overflows.

Since most times all the arguments are compile time constants, an alternate overload template is always provided, to make sure all the possible simplifications are done:

```
template < uintptr_t CONTROL_BITS, bool INCLUDE_TYPE,
size_t SIZE, size_t ALIGNMENT > Block inplace_allocate();
```

This is inplace_allocate for the case of multiple producers, but with relaxed consistency:

```
/* LFQueue_Tail<..., concurrency_multiple,
consistency_relaxed>::inplace_allocate */
Block inplace_allocate(uintptr_t i_control_bits,
bool i_include_type, size_t i_size, size_t i_alignment)
{
     if (i_alignment < min_alignment)
     {
          i_alignment = min_alignment;
          i_size = uint_upper_align(i_size, min_alignment);
     }
```

```
     auto tail = m_tail.load(detail::mem_relaxed);
     for (;;)
     {
          // allocate space for the control block
          void * new_tail = address_add(tail,
i_include_type ? s_element_min_offset : s_rawblock_min_offset);

          // allocate space for the element
          new_tail = address_upper_align(new_tail, i_alignment);
          void * const user_storage = new_tail;
          new_tail = address_add(new_tail, i_size);
          new_tail = address_upper_align(new_tail,
s_alloc_granularity);

          // check for page overflow
          auto const new_tail_offset = address_diff(new_tail,
address_lower_align(tail, ALLOCATOR_TYPE::page_alignment));
          if (DENSITY_LIKELY(new_tail_offset <= s_end_control_offset))
          {
               if (m_tail.compare_exchange_weak(tail,
static_cast<ControlBlock*>(new_tail),
                    detail::mem_acquire, detail::mem_relaxed))
               {
                    auto const next_ptr = reinterpret_cast<uintptr_t>(
new_tail) + i_control_bits;

                    raw_atomic_store(&tail->m_next, next_ptr,
detail::mem_release);

                    return { tail, next_ptr, user_storage };
               }
          }
          else if (i_size + (i_alignment - min_alignment) <=
s_max_size_inpage)
          {
               tail = page_overflow(tail);
          }
          else
          {
               /* this allocation would never fit in a page,
allocate an external block */
               return external_allocate(i_control_bits, i_size,
i_alignment);
          }
     }
}
```

Unless a page overflow occurs, no page pinning is necessary on the producer side. The reason is that after that a producer allocates space (updating m_tail), that space can never be consumed until it unzeroes it and then removes the busy flag.

The are two reasons why this tail provides relaxed consistency instead of sequential consistency:

Every put updates the tail, and then writes the member m_next of the control-block it has just allocated (m_next is zero before the latter write). In the middle of these two writes, other producers may succesfull do other puts (they are not blocked), but for the consumers the queue is truncated to the first zeroed m_next. A put may be temporary not observable even to the thread that successfully has carried it.

Consumers are requested to zero the memory while consuming elements. In some cases of high contention a consumer may see an m_next zeroed by the other consumers, and incorrectly consider it an end-of-queue marker.

The last case of LFQueue_Tail::inplace_allocate is for multiple producers, and sequential consistent queue. Basically we have to solve the two problems above.

In this specialization of LFQueue_Tail we allocate in the pages only values requiring at most a number of bytes equal to the square of s_alloc_granularity. All other values are allocated with a legacy heap allocation.

To solve the problem 1 we adopt a two phases tail update. To solve the problem 2 we just ask to consumers to not zero the memory. Here is the code of the put for sequential consistent queues:

Block inplace_allocate(uintptr_t i_control_bits, bool i_include_type,

```
size_t i_size, size_t i_alignment)
{
     if (i_alignment < min_alignment)
     {
          i_alignment = min_alignment;
          i_size = uint_upper_align(i_size, min_alignment);
     }

     auto const overhead = i_include_type ? s_element_min_offset :
s_rawblock_min_offset;
     auto const required_size = overhead + i_size + (i_alignment -
min_alignment);
     auto const required_units = (required_size +
(s_alloc_granularity - 1)) / s_alloc_granularity;

     detail::ScopedPin<ALLOCATOR_TYPE> scoped_pin(this);

     bool const fits_in_page = required_units <
size_min(s_alloc_granularity, s_end_control_offset /
s_alloc_granularity);
     if (fits_in_page)
     {
          auto tail = m_tail.load(mem_relaxed);
          for (;;)
          {
```

```
                auto const rest = tail & (s_alloc_granularity - 1);
                if (rest == 0)
                {
                      // we can try the allocation
                      auto const new_control =
reinterpret_cast<ControlBlock*>(tail);
                      auto const future_tail = tail +
required_units * s_alloc_granularity;
                      auto const future_tail_offset = future_tail -
uint_lower_align(tail, page_alignment);
                      auto transient_tail = tail + required_units;
                      if (DENSITY_LIKELY(future_tail_offset <=
s_end_control_offset))
                      {
                            if (m_tail.compare_exchange_weak(tail,
transient_tail, mem_relaxed))
                            {
                                  raw_atomic_store(&new_control->m_next,
future_tail + i_control_bits, mem_relaxed);


     m_tail.compare_exchange_strong(transient_tail,
future_tail, mem_relaxed);

                                  auto const user_storage =
address_upper_align(
address_add(new_control, overhead), i_alignment);

return { new_control, future_tail +
 i_control_bits, user_storage };
                            }
                      }
                      else
                      {
                            tail = page_overflow(tail);
                      }
                }
                else
                {
                      // an allocation is in progress, we help it
                      auto const clean_tail = tail - rest;
                      auto const incomplete_control =
reinterpret_cast<ControlBlock*>(clean_tail);
                      auto const next = clean_tail + rest *
s_alloc_granularity;

                      if (scoped_pin.pin_new(incomplete_control))
                      {
                            auto updated_tail = m_tail.load(mem_relaxed);
                            if (updated_tail != tail)
                            {
                                  tail = updated_tail;
```

```
                                    continue;
                               }
                          }

                          uintptr_t expected_next = 0;

     raw_atomic_compare_exchange_weak(&incomplete_control->m_next,
&expected_next, next + detail::NbQueue_Busy, mem_relaxed);
                          if (m_tail.compare_exchange_weak(tail, next,
mem_relaxed))
                               tail = next;
                     }
               }
     }
     else
     {
          return external_allocate(i_control_bits, i_size,
i_alignment);
     }
}
```

A producer starts analyzing the value of m_tail. If it is multiple of s_alloc_granularity, then there is no other put in progress. So it:

- Adds to m_tail the required size in bytes divided by s_alloc_granularity. This is enough to make other consumers realize that a put is in progress, and how much memory this put is allocating.
- Setups the control block (that is sets m_next)
- Sets m_tail to point after the allocation
- Otherwise, if the tail is not multiple of s_alloc_granularity, the thread first completes the put in progress, and then try to do its own put.