This article describes the implementation of `lf_heter_queue`, the lock-free heterogeneous queue implemented in density.

Memory management

Whenever possible, we use a page based memory management. Grouping user data in large memory pages increases the granularity of every memory management based task, including safe indirections. We promote locality in the allocation of the user data.

Memory pages have a fixed size, which can be arbitrary, and a fixed alignment, which 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.

To allow a safe access to pages in a concurrent context, the allocator supports page pinning. Pinning is a kind of reference counting: while a page is pinned, even if the page is deallocated, the allocator will not recycle it in an allocation function, and will not alter its content in any way.
Pinning a page that has been already deallocated is legal. Accessing such page, or doing anything that is not unpinning it, triggers an undefined behaviour.
When encountering a pointer with value P in a data structure, a thread should pin it, and then check if P is still present in the data structure. If not, it has to unpin and retry.

Sometimes we need the pages we allocate to have a zeroed content. Lock-free algorithms may exploit this constraint to make threads agree on an history of the allocated memory, which begins with the zero state. When deallocating a page, we may tell to the allocator that the page is 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 characteristic of a page.

```
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 * allocate_page_zeroed();
      void deallocate_page(void *) noexcept;
      void deallocate_page_zeroed(void *) noexcept;
      void pin_page(void *) noexcept;
      void unpin_page(void *) noexcept;
};
```

Note: functions returning a page (the first 2) return a pointer to the first byte of the page. Anyway functions taking a page as parameter (the others) always align the address: so the user can specify a pointer to any byte within the page.

Raw atomics

Sometimes in lock free algorithms we can't use `std::atomic`, basically because standard atomics are not trivially constructible. So we introduce a set of non-standard of functions for atomic operations on 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;
```

We have omitted the default value for `memory_order` parameters, which is always `std::memory_order_seq_cst`.
These functions behave like the standard counterparts. Anyway raw variables are trivially constructible, and can be zero-initialized (for example with a memset). A second advantage

of raw atomics is that in some (rare) cases we can mix non-atomic and atomic writes to the same variable.
The implementation of `lf_heter_queue` we are going to describe requires row atomics. If for a given compiler or OS raw atomics can't be implemented, the queue can't be used.

Anatomy of the queue

The storage of an `lf_heter_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*.
A `lf_heter_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.
If the queue is not empty, the head pointer points to the first value of the queue. Values are stored as a null-terminated forward linked-list. Values are allocated linearly, in container order, with a granularity specified by the internal constexpr `s_alloc_granularity` (usually 64). The pointer 'next' of the tail element is always zeroed.

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:

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

The layout of a value is composed by:

- an instance of `ControlBlock`, an internal struct of the library. It is always present, and holds the pointer 'next' and the state flags of the value.
- possibly a type eraser, (usually) not present for raw allocations and for the *end-of-page* `ControlBlock`.
- possibly the user storage (the element or the raw block), not present for the *end-of-page* `ControlBlock`.

The bottom of the usable space 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 the first control block of the page, all the other 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.

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':

```
struct ControlBlock
```

```
{
      volatile uintptr_t m_next;
};
```

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

```
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’ and ‘Dead’.
While an element is being produced (before commit is called on the put transaction), the element is not observable on the queue. If commit is not called, no one will ever observe any part of that element. So puts are transactions.
When a thread start consuming an element, it sets the flag ‘Busy’ on the element, which instantly disappears from the queue. If the consume operation is committed, the element is gone forever. Otherwise, if the consume operation is canceled, the element reappears in the queue. Since other consumers have observed 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;
```

By default we provide the most general heterogeneous lock-free queue, but we allow the user to specify some limitations to have better performances.

[1] If we would consider consumes infallibile operations (just like destructors), they would be transactions. Anyway we expect the user to do non-trivial actions during a consume.

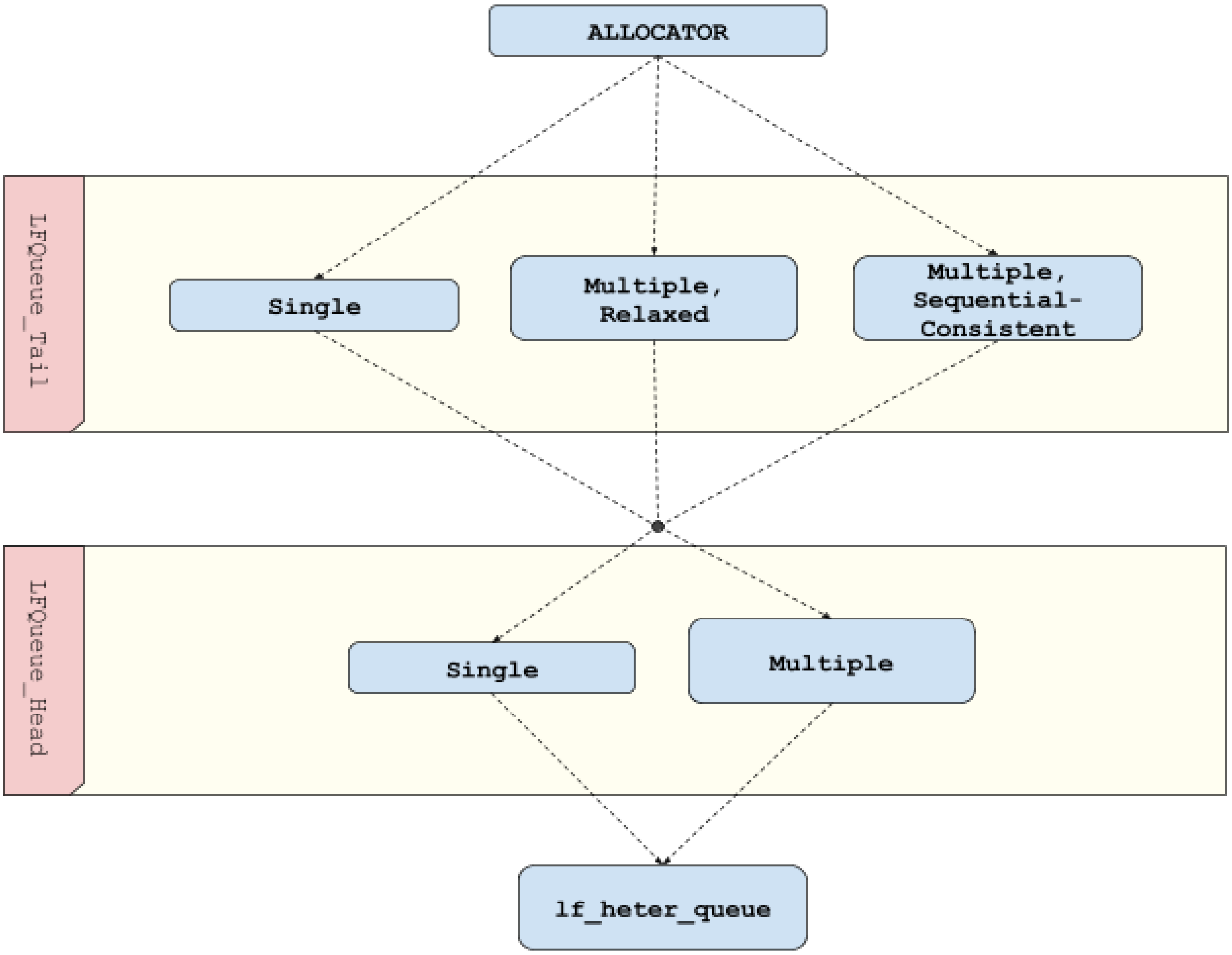
Template parameter	Allowed values

`PROD_CARDINALITY`	`concurrency_multiple,`
	`concurrency_single`

`CONSUMER_CARDINALITY`	`concurrency_multiple,`
	`concurrency_single`

`CONSISTENCY_MODEL`	`consistency_sequential,`
	`consistency_relaxed`

To handle such complexity, we split the implementation in 3 class templates. A `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.

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[2].
- 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 = ...;
```

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

[2] At this time the advantage has not been verified and measured.

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` is immutable.

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:

1. We test for the page overflow in the domain of page offset to avoid having to handle the pointer arithmetic overflows.
2. 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:

1) 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.
2) 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`. [3] All other values are allocated with a legacy heap allocation.

[3] We may avoid legacy allocations for values big up to the size of a page allocating such values in a secondary stream of pages. At the moment density is not doing that.

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.