Memory allocator for multi-processor systems

Gratian Lup

Contents

1. Specifications ... 3
1.1. Introduction ... 3
1.1.1. Objectives ... 3
1.1.2. Terminology ... 4
1.1.3. Technologies used ... 4
2. Overview ... 5
3. Small and medium locations ... 6
4. Thread context ... 7
5. Groups for small locations ... 8
6. Groups for medium locations ... 10
7. Identifying the type of a location ... 11
8. Memory blocks ... 12
9. Large locations ... 15
10. Abstraction layer ... 17

1. Specifications

1.1. Introduction

This project presents the design and some of the implementation details of a high-performance memory allocator intended to be used on modern multi-core and multi-processor systems with advanced memory architectures based on NUMA.

Nowadays the maximum performance that can be extracted from a single core has reached its peak and CPUs with a large number of cores are being created. To exploit the parallelism provided by these CPUs it's required to select adequate algorithms, together with data structures optimized for high concurrency and high throughput.

In many cases these highly concurrent data structures (which are usually implemented using lock-free synchronization mechanisms) are being limited by the memory allocator. The default memory allocators usually use a global lock which must be taken before an allocation/deallocation can be made. This leads to the situation where multiple threads are waiting to gain access to the memory allocator, slowing down the data structures, and ultimately, the algorithms.

On the other side, a memory allocator optimized for multi-core CPUs allows more threads to allocate/deallocate memory at the same time. Serialization (waiting to gain the lock) cannot be completely prevented, but it can be reduced considerably using a hierarchy of preallocated memory blocks. A similar strategy is used by the memory allocator presented in this document.

1.1.1. Objectives

- Efficient allocation for objects of small, medium and large sizes.
- High concurrency on multi-core and multi-processor systems.
- Reducing the effects of false cache line sharing.
- Exploiting the memory architecture in case of NUMA systems.
- Reducing memory fragmentation and utilization for small objects.
- Alignment on a 16 byte boundary of objects multiple of 16 bytes. Advantageous when using SSE instructions which require 16 byte alignment for maximum performance.
- Independence from the operating system using an abstraction layer.
- Compatibility with both 32 and 64 bit systems.
- Easy integration with new and existing applications.

1.1.2. Terminology

- **Memory policy**: Controls the way memory is allocated from the operating system. Based on the compile flags a standard or a NUMA-optimized policy is selected.

- **Block**: A region having 1MB of memory in the current implementation. It is obtained directly from the operating systems and partitioned into groups. The CPUs from the same NUMA node can share a block.

- **Group**: A subdivision of a block having 16 KB of memory (64 groups/block) for groups containing small locations and having 64KB (16 groups/block) for groups containing medium locations. All locations in a group must have the same size.

- **Location:** A region of memory requested by the client. Locations are split into three categories[1], each with a different allocation strategy:
    - Small locations : between 8 bytes and ~2.6KB
    - Medium locations: between ~2.6KB and ~8KB
    - Large locations: between ~8KB and 1MB.

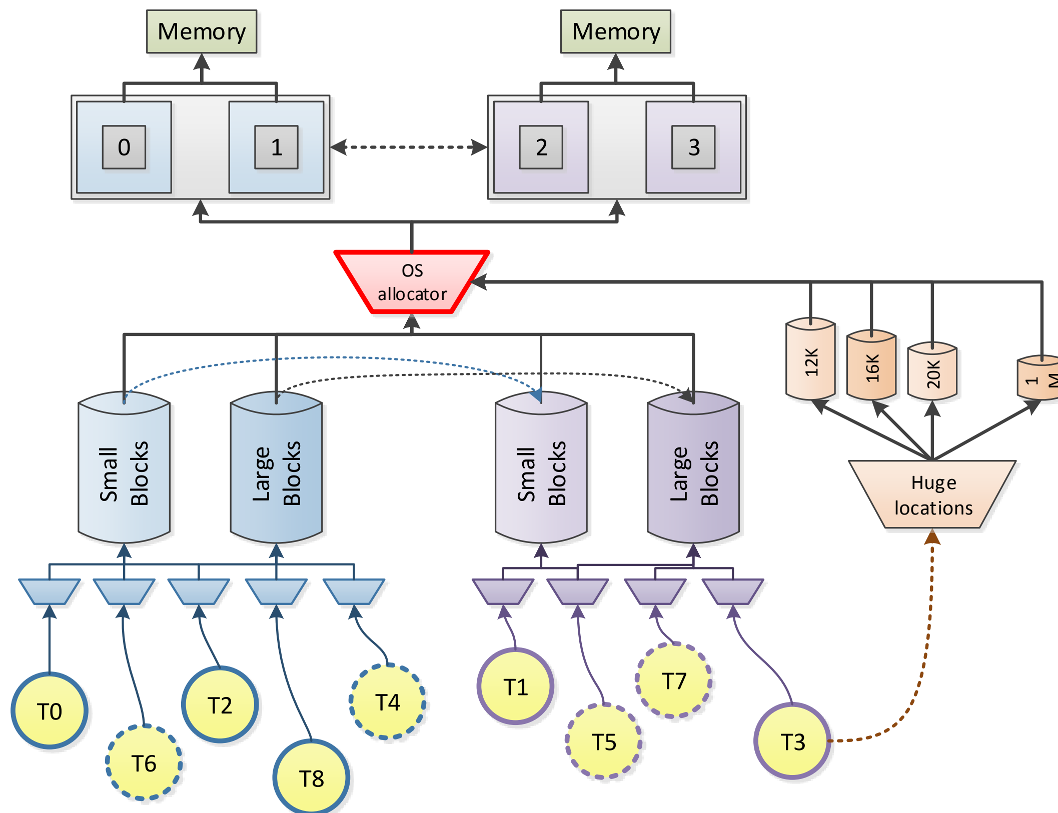
1.1.3. Technologies used

Visual C++ 2008
MASM
WinDBG
Intel VTune
Intel Thread Profiller
Eclipse (with Pydev for Python support)
TortoiseSVN
VMware Workstation
Windows Vista

[1] All allocations with a size larger than 1MB are made directly from the operating system.

2. Overview

The following diagram presents a simplified view of the allocator's components:



The allocator is built around a level-based hierarchy, similar to the one used by modern CPUs for their cache system. Each thread tries first to allocate memory from a regions of private memory (associated and available only to that particular thread). The public regions of memory, available to every thread, are accessed only if there is no more memory available on the private level. The objective is to allocate as much memory as possible from the private memory region.

There is a separate private memory region for each executing thread, allowing access without any synchronization mechanism to be performed. Access to the public memory regions must be synchronized because they can be accessed by all CPUs part of the same NUMA node.

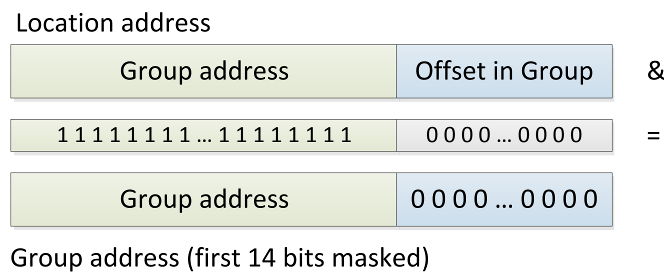
The usage of memory regions highly reduces the probability that two threads serialize their execution when allocating/deallocating memory. Only when there is no more private memory available and the public memory regions must be accessed serialization can occur. For each location size (small, medium and large) a different allocation strategy is selected to reduce the number of times the public memory region must be accessed.

3. Small and medium locations

Small and medium locations use a similar allocation strategy, the main difference being the size of the group and minimum/maximum size of the allocated objects.

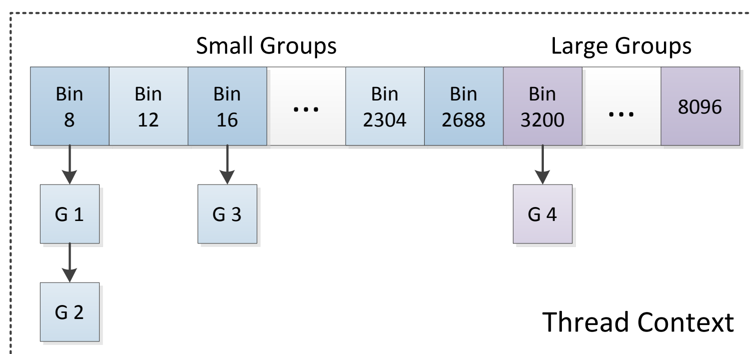
Most memory allocators use for each allocated location a small structure (header) placed exactly before the object. These structures usually specify the size of the location and sometimes include a pointer to the structure describing the previous allocated location. Although this is a simple method to implement, it has the great disadvantage that for small locations (< 256 bytes) much of the allocated memory belongs to these structures and can't be used by the client anymore. This also leads to the inefficient use of cache memory because less locations fit into a cache line.

The proposed allocator doesn't use a separate structure for each location, instead it uses a single one which describes a group of several locations, placed at the beginning of the group. A group can be obtained from the *block allocator* and can be owned by a single thread at a time. To have fast access to the group header knowing only the memory address of the location it is required for the group to be aligned at 16KB and to have a size of 16KB. When the allocator receives the memory address of a location that should be deallocated the first 14 bits are masked and the group address is obtained. Now having access to the group header, the location can be returned to the allocator.



The usage of groups forces all location in a group to have the same dimension, different groups being necessary for other location sizes. There is a series of predefined locations sizes for which groups can be created (for example 8, 12, 16, up to ~2.6KB for small locations). The groups are partitioned based on their size and added to their corresponding partition using a linked list.

When memory is allocated the size is rounded up to the nearest supported location type (for example, 6 bytes -> 8 bytes, 60 bytes -> 64 bytes), then a group having locations of the corresponding size is searched.



4. Thread context

Each execution thread has an associated context that contains an array with the predefined group lists. These group lists represent the private memory region held by the thread. It also contains information about the associated thread and the NUMA node to which it belongs.

The context is created the first time an allocation from an executing thread is made and it is associated with the thread using Thread Local Storage (TLS). For the next allocation, the group is obtained using TLS and an attempt to allocate memory from the existing groups is made. If there is no group with free locations the new group is requested from the *block allocator*.

Simplified algorithms for allocation and deallocation:

```
void* Allocate(int size) {
    context = GetCurrentContext();
    if(context == NULL) {
         context = CreateContext(); // Create the context if it doesn't already exist.
    }

    allocInfo = RoundSize(size);        // Round to a predefined location size.
    bin = context->Bins[allocInfo.Bin]; // Get the list with the associated groups.
    group = bin->First;

    address = group->GetLocation();
    if(address == NULL) {     // The group has no more free locations.
        if(bin->Count >= 2) { // Check if the second group has free locations.
            group = bin->First->Next; // If the second one does not, none does.
            address = group->GetLocation();
            if(address != NULL) {
                bin->MakeFirst(group); // Move the group to the front of the list.
                return address;
            }
        }

        group = blockAlloc.GetGroup(); // A new group is necessary.
        if(group == NULL) return NULL; // No more memory availalble on the system.

        // Initialize the new group.
        group.Init(allocInfo.Size, context.ThreadId, bin);
        bin->AddFirst(group);         // Add the group to the private region.
        return group->GetLocation(); // Allocate from the new group.
    }

    return address;
}

void Deallocate(void *addr) {
    context = GetCurrentContext();
    group = GetGroup(addr); // Get the group address masking the first 14 bytes.
    bin = group->ParentBin; // Each group is associated with its list.
    group->ReturnLocation(addr, context); // Return the location to the group.

    if(group->Unused()) {
        blockAlloc.ReturnGroup(group); // Return the group to the block allocator.
    }
    else if(bin.Count > 2) {
        bin->Remove(group);    // Move the group to the second position in the list.
        bin->AddSecond(group); // Allows fast checking for groups with free locations.
    }
}
```

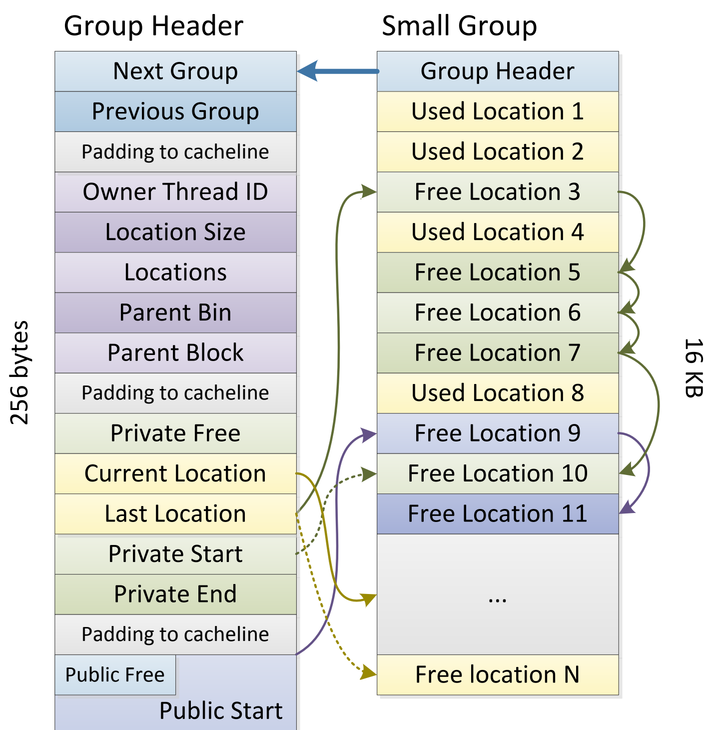
5. Groups for small locations

In the current implementation, group for small locations have a size of 16KB and contain between 2016 (for locations of size 8 bytes) and 6 locations (for locations of size 2688 bytes).

When the group is empty locations are used in increasing order: the location is marked as used by incrementing the *current location* pointer. When the last free location free memory is searched in the lists of locations returned to the group.

There are two lists that store the locations returned to a group: a list with the locations returned by the thread which owns the group (the private list) and a list for all other threads (the public list). Access to the private list requires no synchronization mechanism because only the thread owning the group can modify it. On the other side, access the public list must be synchronized because more threads could be returning locations at the same time. For synchronization an efficient lock-free algorithm based on the CAS instruction is used.

Both lists are based on a FIFO (stack) model, always having the most recent returned location at the top. This offers a performance advantage because the first reused location is always the most recent returned one, making it likely that it still in the cache or at least the associated page file information is still in the TLB. In the image below it can be observed that each list is located in a separate cache line; this separation reduces the amount of synchronization of the caches when both the private and public lists are accessed from different processors.



Each group knows the thread it is associated with, and also the bin in which it resides.

The pointers to the head of the free location lists are found in the header, while the linked list nodes are found inside the free locations (each free location has a *next* pointer, the last one being NULL marking the end of the list). As it can be seen, no additional memory is required to store the private and public free location lists.

The head of the public list is a 64 bit number updated atomically. The first 32 bits contain the address of the first node in the list, the remaining 32 bits contain the number of list nodes. This allows implementing a lock-free linked-list using the CAS instruction for 64 bit numbers.

On 64 bit systems the pointer to the head of the public list is stored on 48 bits, while the number of list nodes in the remaining 16 bits (48 bits are enough to address the memory available in current computers).

Simplified algorithms for allocation and deallocation from a group

```
void* GetLocation() {
    if(CurrentLocation <= LastLocation) { // If there are still locations never allocated,
        address = CurrentLocation;        // use them first. If not only locations from
        CurrentLocation += LocationSize;  // the private or public lists can be allocated.
        PrivateFree--;
        return address;
    }
    else {
        if(PrivateStart != NULL) {  // Check if there are locations in the private list.
            address = PrivateStart; // Note that access synchronization is not required.
            PrivateStart = PrivateStart->Next; // Extract the first location from the list.
            PrivateFree--;
            return address;
        }
        else {
            PrivatizePublic(); // Move the locations from the public to the private list
            if(PrivateStart != NULL) {
                return GetLocation(); // Retry allocation.
            }
        }
    }

    return NULL; // No more free locations in the group.
}


void ReturnLocation(void *addr, ThreadContext *context) {
    if(ThreadId == context->ThreadId) { // If the thread is the owner use the private list.
        address->Next = PrivateStart;   // Make the location the head of the private list.
        PrivateStart = address;
        PrivateFree++;
    }
    else { // The location is added to the public list (access synchronization required).
        while(1) { // Retry until the list head could be changed successfully.
            oldValue = PublicStart;
            newValue = {oldValue.Address, oldValue.Free + 1};
            address->Next = oldValue->Address;

            // The atomic CAS instruction is used to set the new head node.
            if(CAS(&PublicStart, newValue, oldValue) == oldValue) break;
        }
    }
}
```

The *PriavatizePublic* method also uses a loop with a CAS instruction to change the head of the public free location list. Groups also offer methods to test if there are free locations, if they are in the private or public free location lists, if the use of the group is low, etc.

6. Groups for medium locations

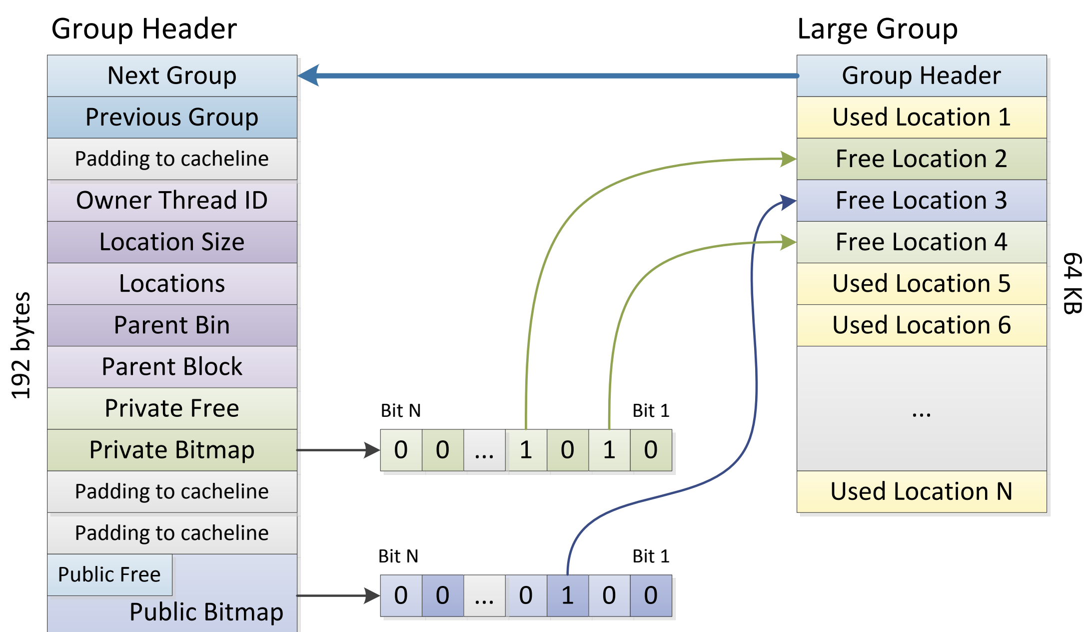
In the current implementation medium-location groups have a size of 64 KB, having between 20 locations (for a location of 3200 bytes) and 8 locations (for a location of 8096 bytes). The header has a size of 192 bytes (3 cache lines). The algorithms for allocating/deallocation medium locations are similar to the ones used for small locations, but there are some differences listed below.

Instead of using linked lists to keep track of the returned locations two bitmaps are used instead: a bitmap for private location and one for public locations. Each bitmap has 32 bits, enough for the maximum number of medium location that can be allocated from a group.

The private bitmap does not require synchronization, while the public bitmap requires synchronization and it is performed using a lock-free algorithm that uses the CAS instruction.

When a location is allocated from a medium group the private bitmap is searched for the first set bit (starting with the least semnificative bit). Based on its index the memory address of the location is computed, the bit is reset and the location is returned. In most cases allocation does not need synchronization because a single thread can allocate from a medium group at a time.

In case no bits are set in the private bitmap the public bitmap is used to check if there are free locations returned by other threads. The public bitmap is merged with the private bitmap using a lock-free algorithm that resets the bits set in the public bitmap after copying them.

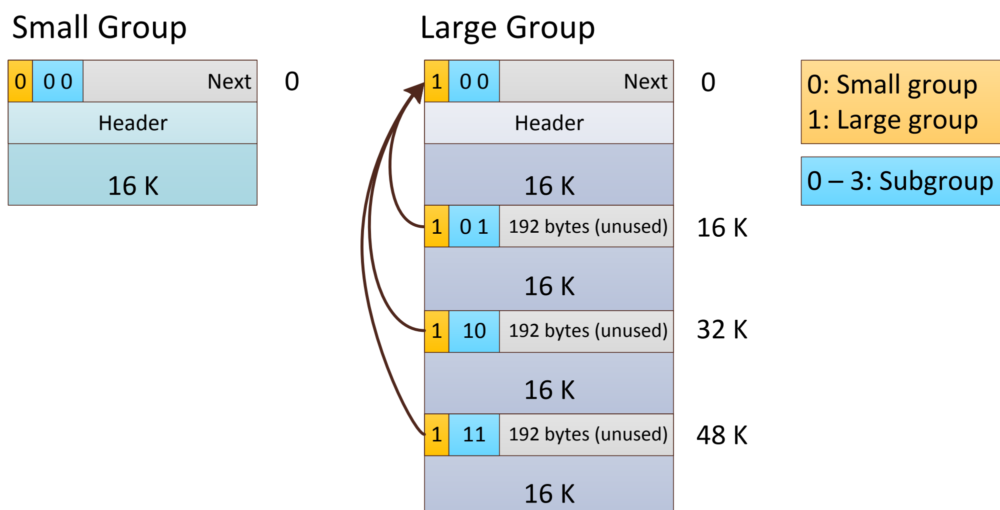


Searching for the first set bit is done using the BSF instruction (Bit Scan Forward), which returns the index of the first set bit starting with the least semnificative position, or -1 if no set bit can be found in the bitmap.

7. Identifying the type of a location

When a location is deallocated the group it is part of must be found to mark the location as being available for future allocations. Because the first 14 bits of the memory address are masked the maximum size of a group is 16 KB. In order to support medium size groups two flags are placed in the first bytes of the group header. These flags indicate the type of the group (small or medium), and for medium groups the

Location type	Offset relative to 16KB alignment
Small location	>= 256 (flags must be checked)
Medium location	>= 192 (flags must be checked)
Large location	= 64
Location directly from the OS	< 64



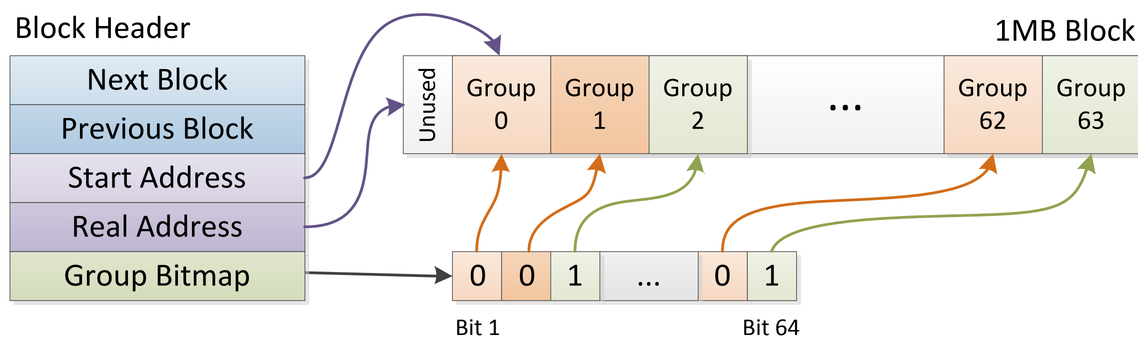
Algorithm to identify the group type:

```
aligned = Mask14(address); // Mask the first 14 bits of the memory address.
diff = aligned - address;
if(diff <= HUGE_HEADER_SIZE) {   // 64
    if(diff <= OS_HEADER_SIZE) { // 16
        // The location is very large, allocated directly from the OS.
    }
    else {
        // The location is large.
    }
}
else {
    if(aligned.Type == TYPE_SMALL) {
        // The location is of small size (part of a small group).
        groupAddress = aligned;
    }
    else {
        // The location is of medium size (part of a medium group).
        // The address of the group header is computed using the subgroup information.
        groupAddress = aligned - (SMALL_GROUP_SIZE * aligned.Subgroup);
    }
}
```

8. Memory blocks

A memory block is a larger amount of memory (1 MB in the current implementation) divided into several groups (64 small groups or 16 medium groups in the current implementation). The memory blocks are partitioned according to their type (small or medium), leading to increased concurrency because blocks of different sizes can be manipulated at the same time.

Each memory block has an associated data structure (the header) that describes it: its start memory address, the number of used/unused groups, the list of groups, etc. These headers are externally allocated (not part of the memory block) by a separate module which tries to pack as many headers as possible in the same memory page. This approach decreases the probability of having a page fault caused by the data structures of the memory allocator.

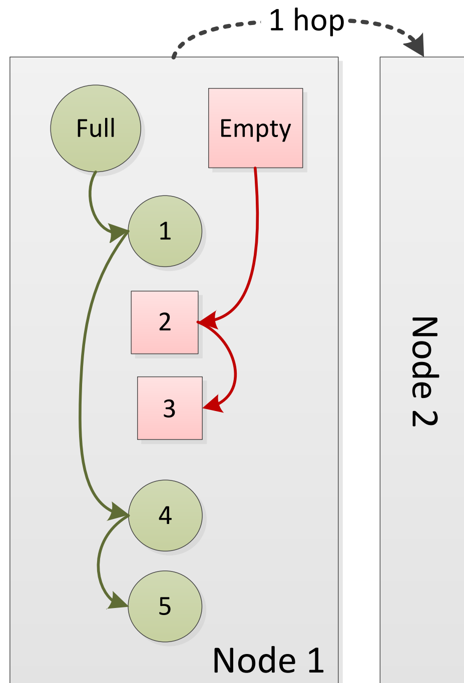


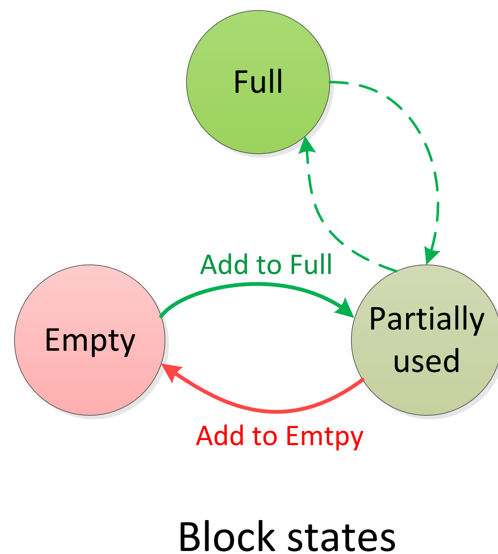
The difference between the *start address* and the *real address* of a group is the fact that the *start address* is guaranteed to be aligned to 16 KB, a requirement that must be met by each group allocated from the memory block. To guarantee the 16 KB alignment the requested memory from the OS is with 16 KB larger than the memory block itself and the *start address* is set to the first value a multiple of 16 KB. It should be noted that this is not required under Windows because the allocated memory block is already found at an address multiple of 16 KB.

Keeping track of the used/unused groups from a memory blocks is done using a 64 bit bitmap. When a thread requests a new group the first set bit is searched in the bitmap. Its index is used to compute the start memory address of the group, then the bit is reset to indicate that the group is used. Because more threads could be requesting groups at the same time access to the bitmap must be synchronized; this is done by a lock-free algorithms using the CAS instruction.

A memory block with no group used is deallocated and returned to the OS. To prevent situations in which a memory block is repeatedly requested and returned from/to the OS a predefined number of memory blocks are always held in memory.

Each memory block is aware of the NUMA node from which it was allocated. This allows a memory block to be returned to the memory block manager associated with the respective NUMA node.





The blocks are kept into two linked lists, based on the number of used groups. If the block still has unused groups in is kept in the *Full* list. Otherwise, if the group has no free groups anymore, the block is kept in the *Empty* list. The advantage of using two distinct lists is that there are fewer tests required to check if there exists at least a block if unused groups (it is enough for the *Full* list to have at least one block). Another advantage is that more than one thread can return groups at the same time, synchronization being required only when a block is moved from a list to the other one.

Required synchronization operations:

- **Obtaining a group:** a single thread can request a group at a time (access must be serialized). If the block transitions from the *Partially used state to the Empty* state the block must be moved from the *Full* list to the *Empty* list.

- **Returning a group:** more than one thread can return blocks at the same time.
  - If the groups are part of separate blocks they can be returned in parallel.
  - If the groups are part of the same block, the synchronization and return operation is implemented using a fast lock-free algorithms (using the CAS instruction).
  - In both cases, if a block transitions from the *Empty* state in the *Partially used* state, the entire block allocator must be locked to move the block from the *Empty* list to the *Full* list. This event appears rarely in practice.

To prevent situations where blocks are repeatedly allocated and deallocated, a predefined number of blocks is always kept allocated and not returned to the OS until the application closes. This prevents the serialization and substantial waiting time introduced by the OS allocator.

Simplified algorithms for obtaining/returning groups from a block:

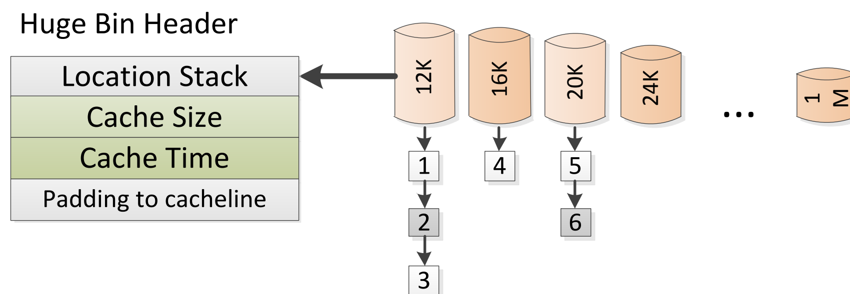
```
Group* GetGroup() {
    Lock(); // Access must always be synchronized when obtaining a group.

    if(Full.Count == 0) {
        block = AllocateBlock(); // Allocate a new block from the OS.
        if(block == NULL) return NULL; // Check if memory could be allocated.

        Full.Add(block); // The block becomes available..
    }

    isEmpty = false;
    block = Full.First;
    group = block->GetGroup(isEmpty);

    if(isEmpty) { // Check if this was the last available group in the block.
        Full.Remove(block); // Move the block to the Empty list.
        Empty.Add(block);
    }

    Unlock();
    return group;
}



void ReturnGroup(Group *group) {
    block = group->ParentBlock; // Return the group to its parent block.
    int prevCount = block->ReturnGroup(group);

    if(prevCount == (GROUPS_PER_BLOCK - 1)) { // Check if the block is completely unused.
        Lock(); // Acces to the two lists must be synchronized.
        if(Full.Count + Empty.Count > CACHE_SIZE) {
            Full.Remove(block);      // If there are enough allocated block
            DeallocateBlock(block); // this one can be returned back to the OS.
        }

        Unlock();
    }
    else if(prevCount == 0) { // Check if the block had no unused groups.
        Lock(); // Access to the two lists must be synchronized.


        Empty.Remove(block); // Move the block to the Full list.
        Full.Add(block);

        Unlock();
    }
}
```

9. Large locations

Because of their large size and the reduced number of allocations in practice, large locations are not associated with each thread anymore and instead are allocated from a global space.

To reduce the number of cases when the OS allocator must be used (it would lead to serialization), a cache of large locations is used. The cache is organized around a *bin* model, the large locations being grouped together with large locations of similar size.

Large locations kept into the cache range from 12KB to 1MB, with a distance of 4KB between the size of each partition (12KB, 16KB, 20KB,...). All allocations are rounded up to 4KB so that even smaller locations can be inserted into the cache and reused.



Each partition uses a list where the unused large locations are held. The partitions also know the size of the large locations they store and the time the locations are kept in the cache. The time values are calculated so that smaller locations (more frequently used) are kept longer in cache than larger locations. Each partition can hold at most 32 unused large locations, partitions with smaller locations being allowed to cache more locations thatn partitions with larger locations.

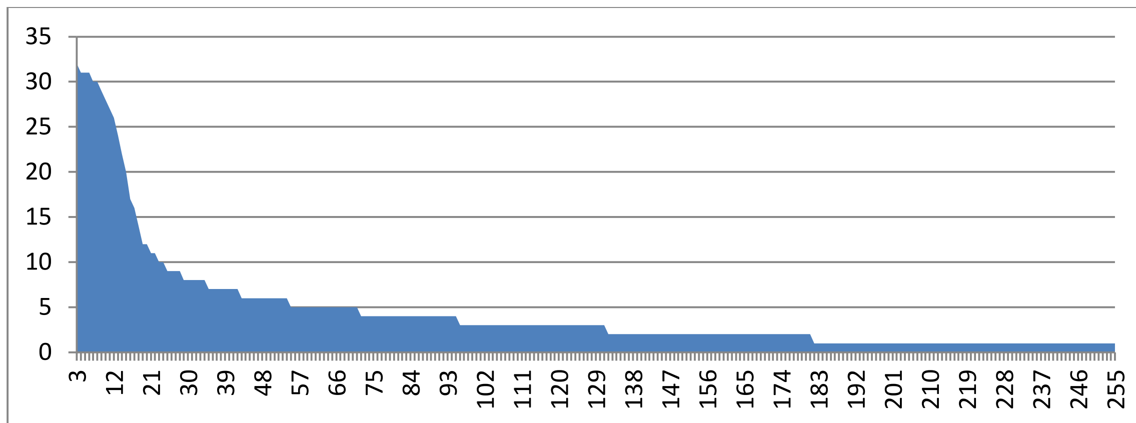
If it is observed that there is high demand for a particular location size, the size of its cache is enlarged by one unit each time four consecutive requests fail to be satisfied from the cache, up to a value eight times larger than the maximum value for the respective location size.

The following formula is used to compute the maximum cached locations and the time they are held in cache for a particular location size:

$$f(x) = \begin{cases} \left[Max - \left(e^{(x-Start)\cdot\ln(Max-Fav)} - 1\right)\right], & x \in [Min, Fav] \\ \left[Fav - \log_{10}(x - FavP) \cdot \dfrac{Fav - Min}{\log_{10}(End - FavP)}\right], & x \in (Fav, Max] \end{cases}$$

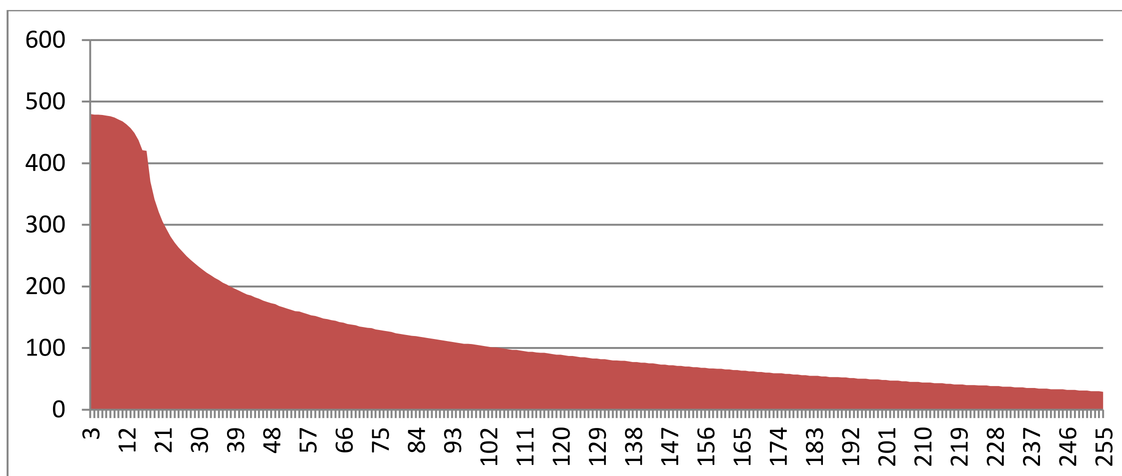
To compute the size of the number of cached locations the following constants are used:

Min = 1, Max = 32 (between 1 and 32 cached locations),
Start = 3, End = 255 (the first and last list index, corresponding to locations size from 12KB to 1MB),
FavP = 16, Fav = 16 (the favored lists end at position 16, having 16 locations)



To compute the maximum time a location is held in cache the following constants are used:

Min = 30, Max = 480 (between 30 seconds and 8 minutes),
Start = 3, End = 255 (the first and last list index, corresponding to locations size from 12KB to 1MB),
FavP = 16, Fav = 420 (the favored lists end at position 16, with a caching time of 7 minutes)



The caching system for large locations increases the speed of allocations substantially, especially in the case when the requests can be satisfied entirely from the cache. To prevent the situation when there is no more demand for locations of a particular size, and locations of that size are still held in cache a thread repeatedly runs a "garbage collection" operation on all lists with cached locations. If the time since the last request for a location is larger than the maximum caching time for a location size half the locations from the associated list are returned to the OS.

The lists with the cached locations behave like a stack, the advantage being that recently returned locations are allocated first, increasing the probability that the memory is still mapped in the virtual memory or even in the CPU cache. Because multiple threads can return large memory locations of the same size synchronization is required for each list. It should be noted that all allocations/deallocations that target different lists of cached locations can be executed in parallel.

Simplified algorithms for allocating/deallocating large locations

```
void* AllocateHuge(int size) {
    allocSize = Round4096(size + HUGE_HEADER_SIZE); // Round up to 4KB.
    index = allocSize / 4096;

    if(List[index].Count > 0) { // Check if there are unused locations in the cache.
        address = List[index].RemoveFirst(); // Extract the last returned location.
        return address;
    }
    else {
        location = AllocMemory(allocSize);  // Allocate memory from the OS.
        location.List = &List[index];       // Associate the location with its parent list.
        return location + HUGE_HEADER_SIZE; // Return the user portion of the location.
    }
}

void DeallocateHuge(void *address) {
    location = address - HUGE_HEADER_SIZE; // Find the start of the location header.
    list = location.List;

    if(list.Count < list.CacheSize) { // Add the location to the cache if space available.
        list.Add(location);
    }
    else {
        DeallocMemory(location); // Return the location to the OS.
    }
}
```

10. Abstraction layer

The abstraction layer simplifies porting the allocator to other platforms and operating systems. It provides wrappers for allocating/deallocating virtual memory, querying information about NUMA nodes, accessing CPU synchronization primitives like CAS, increment, exchange and memory barriers. It also wrapps working with threads, thread local storage and operations on bitmaps like Bit Scan Reverse.

Two implementations of high-performance spin-locks are provided, one which spins around a single bit, the remaining bits being available for other data (useful in constrained memory cases).