# 1. Preface

*this is not as complicated as it looks, promise!* 😃

JEP 387 "Elastic Metaspace" is the rewrite of the Metaspace allocator with the following goals:

- to reduce memory consumption
- to return unused memory back to the OS after class unloading
- to have a clean and maintainable implementation

The corresponding proposal is JEP 387.

This document is both a review guide and a short architectural description of this project.

## 1.1. High Level Overview

Metaspace is used to manage memory for class metadata. Class metadata are allocated when classes are loaded (mostly) and their lifetime is scoped to that of the loading classloader (mostly). Once a class loader gets collected all class metadata it ever accrued get released back to the metaspace allocator. This alleviates the need to track individual allocations so we have a buld delete scenario.

Hence, in its core metaspace is a arena- or region-based allocator. It is optimized for fast and memory efficient allocation of native memory at the cost of not being able to (easily) arbitrarily delete blocks at random times.

**Seen from a very high level:**

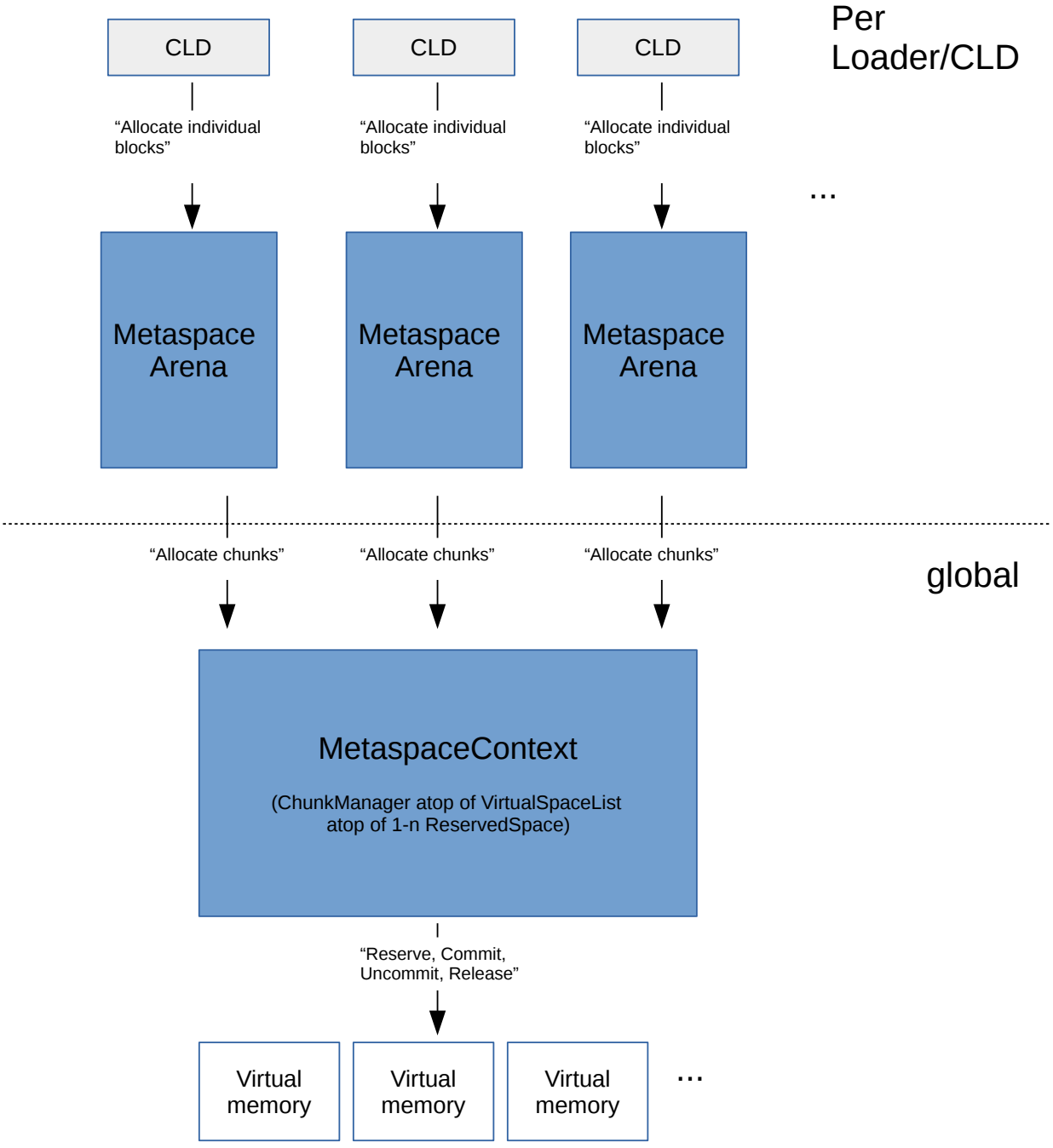
A CLD owns a `MetaspaceArena`. From that arena it allocates memory for class metadata and other purposes via cheap efficient pointer bump. As they are used up the arenas grow dynamically in semi-coarse steps (tuning the growth size is one of the challenges of a good implementation.)

When the CLD is deleted, the arena gets deleted and its memory returned to the metaspace allocator.

Globally there exist a `MetaspaceContext`: the metaspace context manages the underlying memory at the OS level. To arenas it offers a coarse-grained allocation API, where memory is handed out in the form of chunks. It also keeps a freelist of said chunks which had been released from deceased arenas.

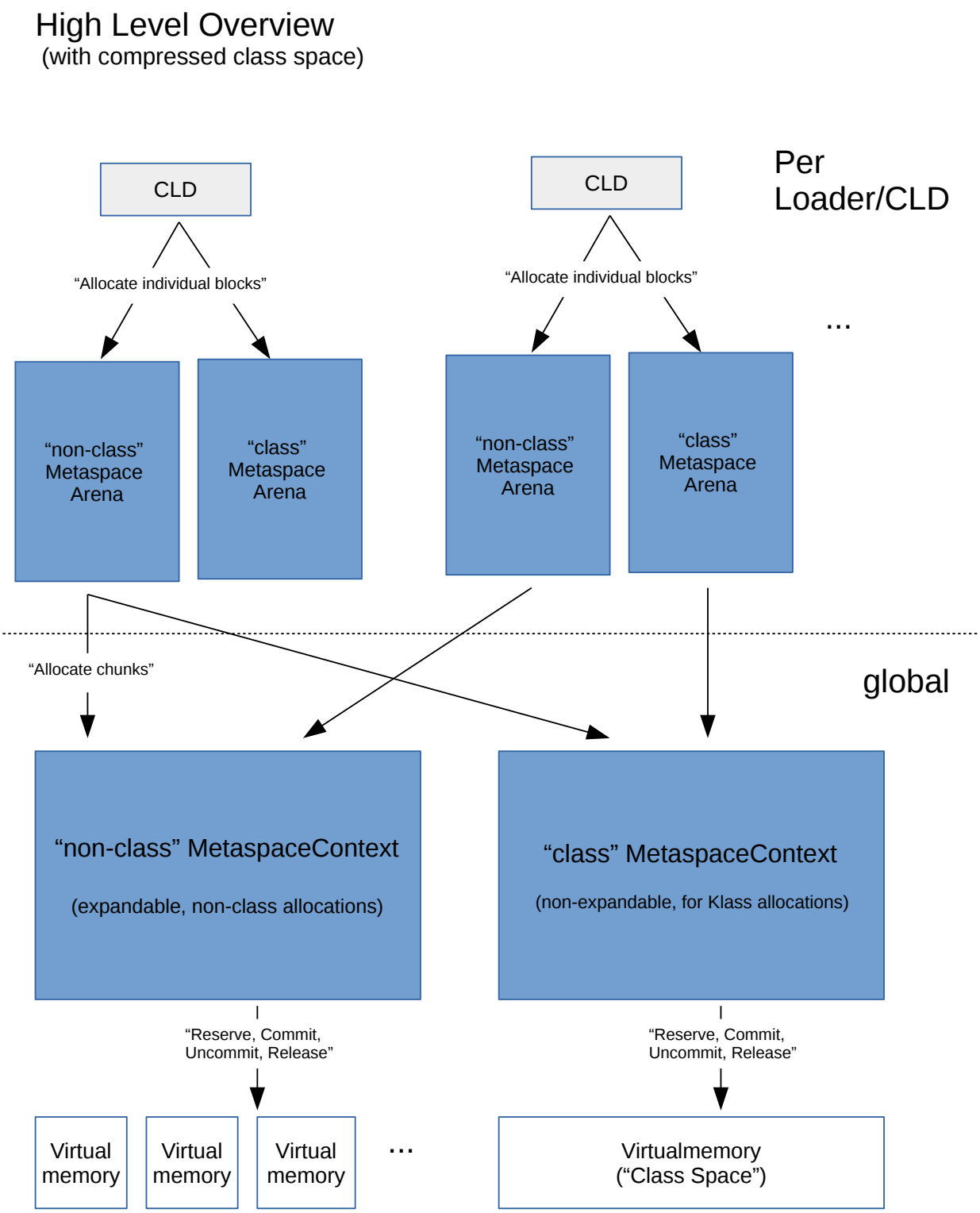
Only one global context exists if compressed class pointers are disabled and we have no compressed class space:

High Level Overview
(no compressed class space)




If compressed class pointers are enabled we need two global metaspace context instances: one holding allocations of Klass structures (the "compressed class space"), one holding everything else (the "non-class"

metaspace). Each CLD also has two arenas to manages these different memory allocations:



# 1.2. Core Concepts

## 1.2.1. Commit Granules

One of the differences of Elastic Metaspace is that memory is committed on demand and uncommitted when not needed. That means we cannot have a contiguous committed region but need to deal with interleaved committed and uncommitted areas.

Implementation-wise this is done via introduction of "commit granules". These are homogenously sized memory units, power-of-two sized, and the whole metaspace address range is split up into these granules. Commit granules are the basic unit of committing and uncommitting memory in Metaspace.

While commit granules may in theory be as small as a single page, in practice they are larger. By default they are 64K. That size is a compromise between virtual memory area fragmentation and uncommit return rates.

The smaller a commit granule is, the more likely it is to be unoccupied and eligible for uncommitting. But at the same, uncommitting very small areas will increase the number of memory mappings of the VM process.

The default size is 64K with `-XX:MetaspaceReclaimStrategy=balanced`. Switching to `-XX:MetaspaceReclaimStrategy=aggressive` switches granule size to 16K (4 pages on most platforms). The latter gives better results in scenarios with heavy usage of anonymous classes, e.g. Lambdas.

## 1.2.2. Metachunks and the Buddy Style Allocator

Metaspace arenas are internally lists of variable-sized memory chunks, the `Metachunk`s. These are the unit of allocation from the lower levels. Arenas obtain these chunks from their respective metaspace context and return all chunks back to the context when they die.

Chunks are variable power-of-two sized. Largest size is 4M ("Root Chunk"). Smallest size is 1K.

Chunks are managed by a buddy allocator. A buddy allocator is a simple old efficient algorithm useful to keep fragmentation at bay, at the cost of limiting the size of managed areas to power of two units. This restriction does not matter for Metaspace since the chunks are not the ultimate unit of allocation, just an intermediate.

In code (see `chunklevel.hpp`), chunk size is given as "chunk level" (`typedef .. chklvl_t`). A root chunk - the largest chunk there is - has chunk level 0. The smallest chunk has chunk level 13. Helper functions and constants to work with chunk level can be found at chunk_level.hpp.

#### 1.2.2.1. Merging chunks

In buddy style allocation, a chunk is always part of a pair of chunks, unless the chunk is a root chunk. We call a chunk a "leader" if it is the first chunk (lower address) of the pair.

```
+------------------+------------------+
| Leader           | Follower         |
+------------------+------------------+
```

A free chunk can be merged with its buddy if that buddy is free and unsplit (which is synonymous if buddy style rules are followed):

```
+-------------------+-------------------+
|         A         |         B         |
+-------------------+-------------------+
                    |
                    v
+-------------------+-------------------+
|                   C                   |
+-------------------+-------------------+
```

If the buddy is not free, or split (in which case on of the splinters will not be free), we cannot merge. In this example, B cannot merge with its buddy since it is splintered:

```
+-------------------+-------------------+
| D1 | D2 |    C    |         B         |
+-------------------+-------------------+
```

#### 1.2.2.2. Splitting chunks

To get a small chunk from a larger one a larger one can be split. Splitting always happens at pow2 borders:

```
+-------------------+-------------------+
|                   A                   |
+-------------------+-------------------+
                    |
                    v
+-------------------+-------------------+
| D1 | D2 |    C    |         B         |
+-------------------+-------------------+
```

## 1.3. Outside interface

The outside interface to the Metaspace (ignoring reporting/monitoring for now) are:

- the `ClassLoaderMetaspace` class
- the Metaspace static "namespace"

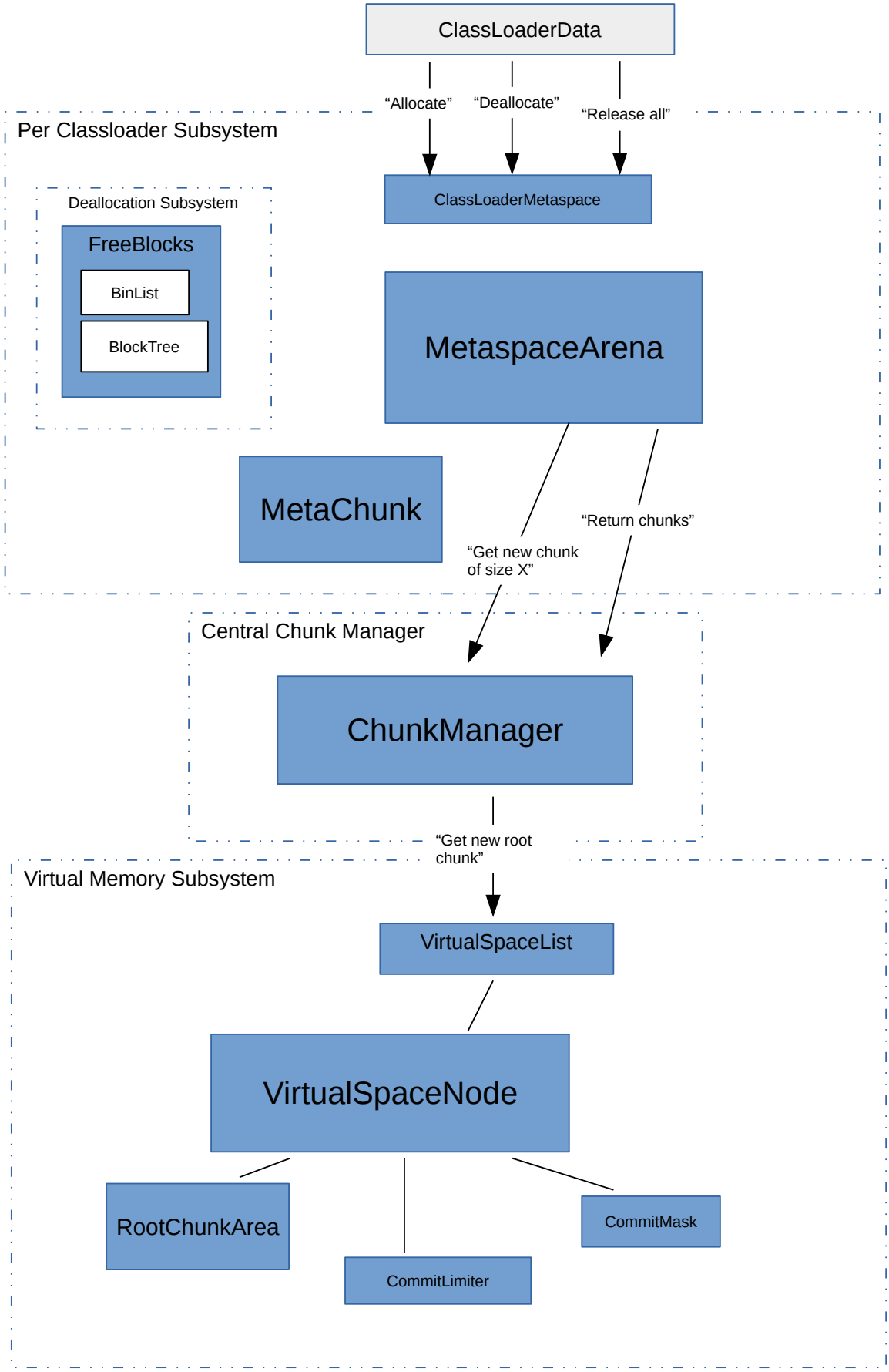
class `ClassLoaderMetaspace` is the holder for above mentioned arenas; it belongs to a CLD. When released (in the wake of a GC collecting the owning loader and its CLD) it will release all Metaspace back to the system:

```
+------+       +----------------------+       +-------------------+
| CLD  | --->  | ClassLoaderMetaspace | ----> | (non class) Arena |
```

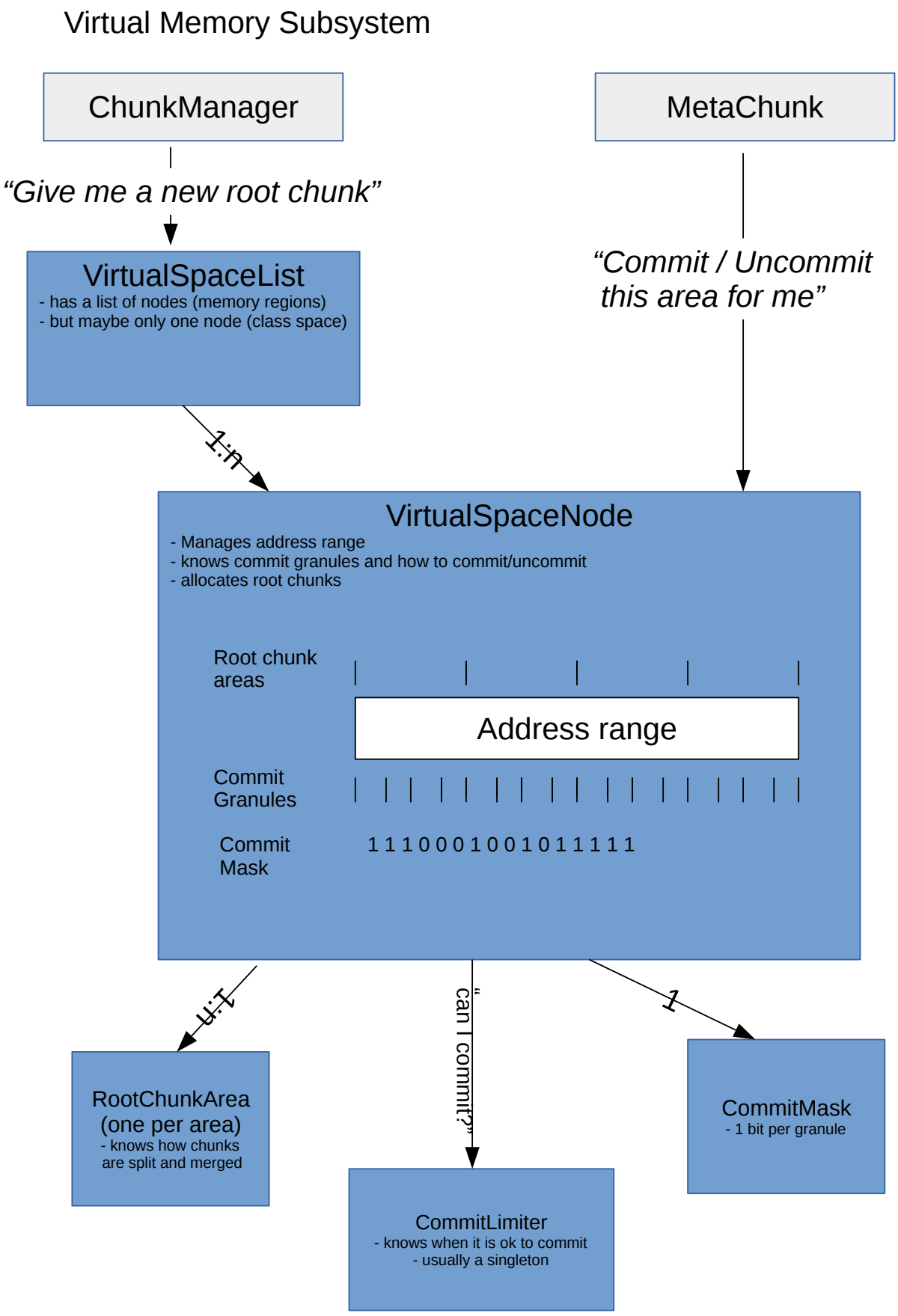
```
+------+       +----------------------+  |    +-------------------+
allocation top
                                         |       |
v
                                         |       + chunk -- chunk -- chunk
-- chunk
                                         |
                                         |    +-------------------+
                                        +--> | (class) Arena     |
                                             +-------------------+
                                                |
                                                + chunk -- chunk
                                                              ^
                                                              top
```

# 2. Subsystems

Elastic Metaspace is divided into separate sub systems which are quite isolated from each other and can be reviewed independently.




## 2.1. The Virtual Memory Subsystem


Virtual Memory Subsystem
ChunkManager
MetaChunk
“Give me a new root chunk”
VirtualSpaceList
- has a list of nodes (memory regions)
- but maybe only one node (class space)
“Commit / Uncommit this area for me”
1:n
VirtualSpaceNode
- Manages address range
- knows commit granules and how to commit/uncommit
- allocates root chunks
Root chunk areas
Address range
Commit Granules
Commit Mask
1 1 1 0 0 0 1 0 0 1 0 1 1 1 1 1
1:1
“can I commit?”
1
RootChunkArea (one per area)
- knows how chunks are split and merged
CommitLimiter
- knows when it is ok to commit
- usually a singleton
CommitMask
- 1 bit per granule

Classes:

- `VirtualSpaceList`
- `VirtualSpaceNode`
- `RootChunkArea` and `RootChunkAreaLUT`
- `CommitMask`
- `CommitLimiter`

The Virtual Memory Layer is the lowest subsystem of all. It is responsible for reserving and committing memory. It has knowledge about commit granules (the granularity at which we commit in Metaspace). Its outside interface to upper layers is the `class VirtualSpaceList`; some operations are also directly accessed via a node in this list (`VirtualSpaceNode`).

A `VirtualSpaceList` is a list of reserved regions (`VirtualSpaceNode`). It is a global structure: only one instance of this structures exists per process. It grows on demand (new reserved regions are added when more space is needed). Regions in this list are typically several MB sized (atm 8M = 2 Root chunks areas, see below).

If we use CompressedKlassPointers, a second global instance of `VirtualSpaceList` exists, which holds the compressed class space. In that case the `VirtualSpaceList` is degenerated: it only ever has one node, sized as big as the CompressedClassSpaceSize (1G).

### 2.1.1. Essential operations

- "Allocate new root chunk"

  ```
  VirtualSpaceList::Metachunk* allocate_root_chunk();
  ```

  This carves out a new root chunk (a chunk of of 4M) from the reserved space and hands it to the caller. This operation is independent on any committed/uncommitted notion. Memory below this chunk does not have to be, and often is not, committed.

- "commit this range"

  ```
  VirtualSpaceNode::ensure_range_is_committed()
  ```

  Memory is divided into "commit granules". This is the basic unit of committing/uncommitting. Only this subsystem knows about these.

  The subsystem knows which granules are committed - it keeps the commit state of granules in a bitmask.

  In contrast to old Metaspace, the committed areas do not have to be contiguous. Any granule can be committed or uncommitted independent from their neighbors.

  Upper layers can request that a given address range should be committed. Subsystem figures out which commit granules are affected and makes sure those are committed. This may be fully or partly a NOOP if the range is already committed.

  When committing, subsystem honors limits (either GC threshold or MaxMetaspaceSize).

- "uncommit this range"

  ```
  VirtualSpaceNode::uncommit_range()
  ```

Similar to committing. Subsystem figures out which commit granules are affected, and uncommits those.

- "purge"

  `VirtualSpaceList::purge()`

  This unmaps all completely empty memory regions.

### 2.1.2. Other operations

The Virtual Memory Subsystem takes care of a number of operations which do not necessarily have to do with virtual memory management. This is a bit historic (earlier versions of the Elastic Metaspace prototype worked differently).

These operations have to do with the Buddy Style Allocator behind the chunk management:

- "split this chunk, maybe repeatedly" `VirtualSpaceNode::split()`
- "merge up chunk with neighbors as much as possible" `VirtualSpaceNode::merge()`
- "enlarge chunk in place" `VirtualSpaceNode::attempt_enlarge_chunk()`

These operations the subsystem does on behalf of the ChunkManager.

### 2.1.3. Classes

#### 2.1.3.1. class VirtualSpaceList

`VirtualSpaceList` is a list of reserved regions (VirtualSpaceNode).

It is a global structure: only one or two `VirtualSpaceList` instances exist per process.

VirtualSpaceList grows on demand - new reserved regions are added when more space is needed. Regions are several MB sized (atm 8M = 2 Root chunks areas, see below).

If `-XX:+UseCompressedClassPointers`, a second global instance of `VirtualSpaceList` exists, which holds the "compressed class space" (see Concepts). That instance is a degenerated version of a list; it only ever has one node which is sized as big as the CompressedClassSpaceSize (1G). New nodes cannot be added.

#### 2.1.3.2. class VirtualSpaceNode

`VirtualSpaceNode` manages one contiguous reserved region of the Metaspace. In case of the compressed class space, it contains the whole compressed class space.

It knows which granules in this region are committed (`class CommitMask`).

VirtualSpaceNode also knows about root chunks: the memory is divided into a series of root-chunk-sized areas (`class RootChunkArea`). This means the memory has to be aligned (both starting address and size) to root chunk area size of 4M.

```
| root chunk area               | root chunk area               | root
chunk area               | <-- root chunk areas

+-------------------------------------------------------------------------
--------------------+
|
|
|                                  `VirtualSpaceNode` memory
|
|
|
+-------------------------------------------------------------------------
--------------------+

|x| |x|x|x| | | | |x|x|x| | | |x|x| | | |x|x|x|x| | | | | | | |x| | |
|x|x|x|x| | | |x| | | |x| <-- commit granules

(x = committed)
```

One root chunk area could contain a whole, unsplit root chunk or a number of smaller chunks. E.g. splitting off a 64K chunk from a 4M root chunk will split the chunk into: 2x64K, 1x128K, 1x256K, 1x512K, 1x1M, 1x2M. But note that the VirtualSpaceNode has no knowledge of this, nor does it care.

The concepts of commit granules and of root chunks and the buddy allocator are almost completely independent from each other.

#### 2.1.3.3. class CommitMask

Very unexciting. Just a bit mask holding commit information (one bit per granule).

#### 2.1.3.4. class RootChunkArea and class RootChunkAreaLUT

`RootChunkArea` encapsulates the Buddy Style Allocator implementation. It is wrapped over the area of a single root chunk and manages buddy operations in this area.

It knows how to split and merge chunks buddy-style-allocator-style. It also has a reference to the very first chunk in this area (needed since `Metachunk` chunk headers are separate entities from their payload, see below, and it is not easy to get from the metaspace start address to its `Metachunk`).

`RootChunkAreaLUT` (for "lookup table") just holds the sequence of `RootChunkArea` classes which cover the memory region of the `VirtualSpaceNode`. It offers lookup functionality "give me the `RootChunkArea`for this address".

#### 2.1.3.5. class CommitLimiter

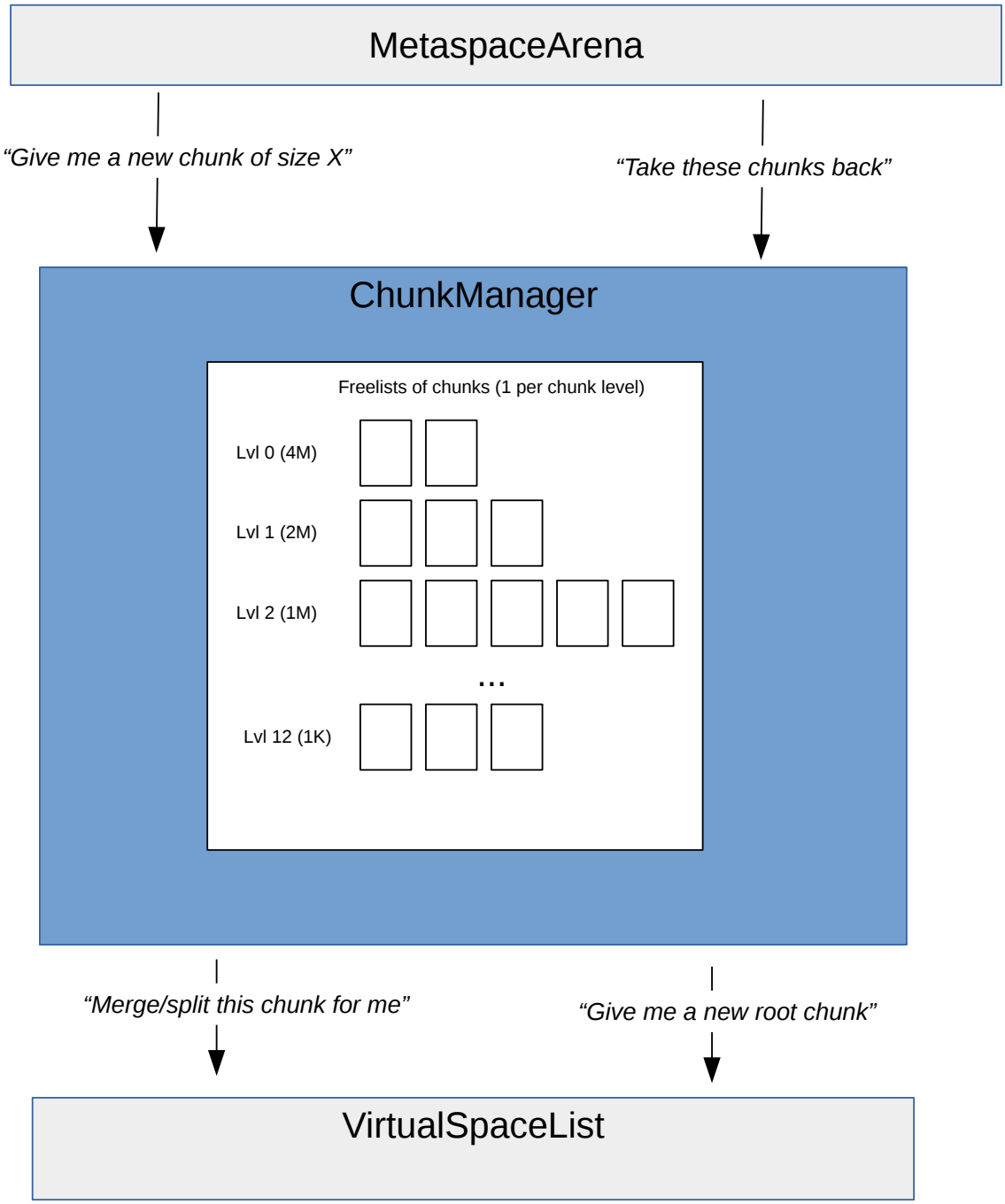
The commit limiter exists to separate the logic of "am I allowed to commit X words more for Metaspace purposes, or would I hit GC threshold or MaxMetaspaceSize?". This is to remove knowledge about GC limit mechanisms and MaxMetaspaceSize from the implementation. It makes the implementation easier to test

and also will allow it, should the need arise, to easily use the metaspace allocator for other things than class data.

Under normal circumstances, only one instance of the `CommitLimiter` ever exists, see `CommitLimiter::globalLimiter()`, which encapsulates the GC threshold and MaxMetaspace queries.

## 2.2. The Central Chunk Manager Subsystem

Central Chunk Manager




Classes

- `ChunkManager`

This subsystem plays a very central role. It only consists of one class, `class ChunkManager`.

Metaspace arenas ask the `ChunkManager` for new chunks if their space is exhausted.

It keeps lists of free unused chunks. Memory of these chunks may or may not be committed.

It sits atop of the Virtual Memory Subsystem. If needed, it will request new root chunks from it to satisfy chunk requests and refill the free lists.

### 2.2.1. Basic operations

- "Give me a chunk of level X"

  `ChunkManager::get_chunk(..)`

  This will provide a chunk to the upper layer of the requested size. If a fitting chunk is found in the freelists, it will reuse that one, splitting larger chunks if needed. Otherwise it will allocate a new root chunk from the Virtual Memory Subsystem and use that to satisfy the request.

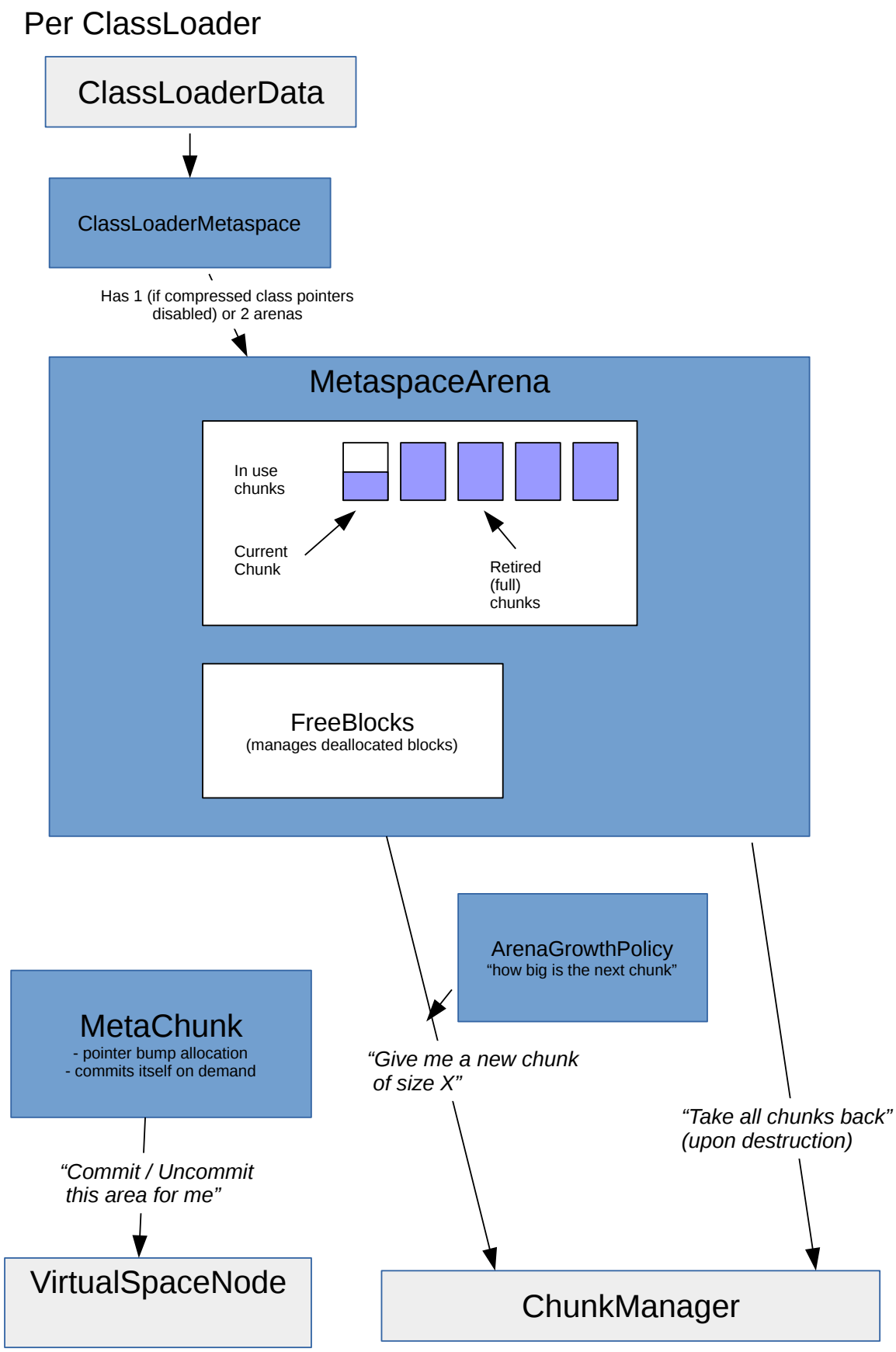
- "I do not need this chunk anymore, keep it"

  `ChunkManager::return_chunk()`

  Callers call this (typically a `MetaspaceArena` before its death) to hand down newly free chunks to the ChunkManager for safekeeping. ChunkManager will put them into the freelist. Before doing this, it will attempt to merge the chunks Buddy-Allocator style with its neighbors to arrive at larger chunks.

  If, after merging with neighbors, the resulting free chunk surpasses a certain threshold, its memory is uncommitted.

## 2.3. Classloader-local Subsystem


Per ClassLoader
ClassLoaderData
ClassLoaderMetaspace
Has 1 (if compressed class pointers disabled) or 2 arenas
MetaspaceArena
In use chunks
Current Chunk
Retired (full) chunks
FreeBlocks
(manages deallocated blocks)
ArenaGrowthPolicy
“how big is the next chunk”
MetaChunk
- pointer bump allocation
- commits itself on demand
“Give me a new chunk of size X”
“Take all chunks back” (upon destruction)
“Commit / Uncommit this area for me”
VirtualSpaceNode
ChunkManager

Classes

- ClassLoaderMetaspace
- MetaspaceArena
- Metachunk
- ArenaGrowthPolicy

The previous sub systems were all global structures. In contrast to that, this subsystem encompasses all Classes whose instances are tied to a class loader.

It builds atop the Central Chunk Manager and indirectly atop the Virtual Memory Subsystem.

It offers fine granular allocation to the caller. A caller needing 240 bytes for a constant pool will get this memory from this layer. Therefore it can be seen as the topmost layer of Metaspace.

### 2.3.1. Basic operations

- "Give me n words of memory from class space / non class space".

  `ClassLoaderMetaspace::allocate()`

  This will allocate n words of Metaspace. Internally the memory will be taken from a chunk via pointer bump allocation, similar to a thread stack. If no chunk exists or the current chunk belonging to the class loader is too small, a new chunk is obtained by asking the ChunkManager.

- "Release all Metaspace I ever allocated"

  `ClassLoaderMetaspace::~ClassLoaderMetaspace()`

  Called upon class loader death. This releases all memory ever allocated for this class loader, by returning all chunks it owns back to the underlying ClassLoaderMetaspace.

- "I do not need this piece of memory, please take it"

  `ClassLoaderMetaspace::deallocate()`

  See Deallocation Subsystem for details.

### 2.3.2. Classes

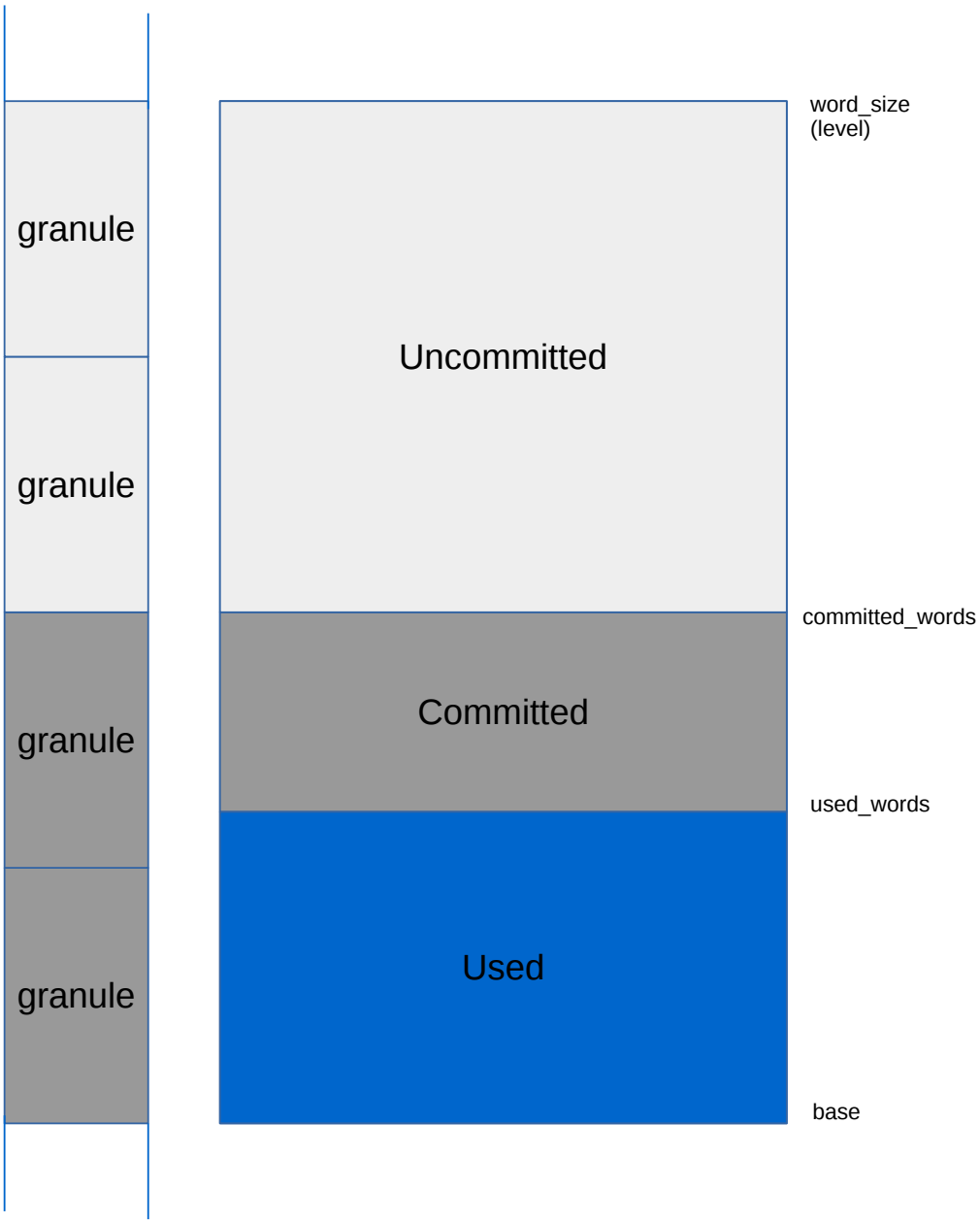
#### 2.3.2.1. class Metachunk

Metachunk wraps one chunk - be it a root chunk of 4M or a small chunk of 1K.

It has a used portion, an unused-but-committed portion and an unused-uncommitted portion:

## Large Metachunk

(spanning multile commit granules)
(can be partly committed, hence committed on demand)




... unless it is smaller or equal to a commit granule, in which case it can only be wholly committed or wholly uncommitted:

## Small Metachunks

(smaller than a single commit granule)
(can only be together committed or uncommitted)


MetaChunk and its payload area are disjunct. In the old Metaspace, `Metachunk` was a header, followed by the chunk payload. Elastic Metaspace separates those two, removing the headers from the payload and

from Metaspace altogether. For details, see `class ChunkPoolHeader` below.

Metachunk knows its chunk memory area (base address and size aka level). It also knows the underlying`VirtualSpaceNode`whose address range the payload resides in. This is needed since it takes care of committing portions of itself on demand.

Metachunk has a state:

- *"in-use"*: A chunk in use is owned by a class loader and carries live metadata.
- *"free"*: A free chunk is not owned by anyone, but awaits re-use in the chunk manager freelist. Its memory
- *"dead"*: A "dead" chunk is just an unused header, without payload.

Metachunk always lives in a linked list - live chunks live in the in-use list of their MetaspaceArena, free chunks in the freelists of the ChunkManager, dead chunk headers live in the ChunkHeaderPool. Therefore `Metachunk` has a prev/next member.

In order to easily do buddy style operations to a chunk (split and merge) it is needed to easily access the neighboring chunks in memory. Therefore `Metachunk` has also references to its lower and upper neighbors.

#### 2.3.2.1.1. Metachunk Memory

A `Metachunk` which is "in-use" gets allocated from via pointer bump allocation, starting at base. So it has a used an unused part:

```
+------------------------------+--------------------------------------+
|     used                     |     unused                           |
+------------------------------+--------------------------------------+

^                              ^                                      ^
base                         used_words                             end.
```

The memory underlying a `Metachunk` may consist of any number of commit granules, which can be committed or uncommitted independently from each other. So the memory below a chunk could be "checkered".

Of course, the used portion of a `Metachunk` has to be committed, otherwise we could not store data in them. Therefore, when allocating new memory from the Chunk, before moving the top-pointer, `Metachunk` ensures the newly used memory is committed by asking the underlying`VirtualSpaceNode`.

But since this is costly - we do not want to bother`VirtualSpaceNode`for every single allocation - `Metachunk` also keeps record of the highest committed address in its range. Note that does not mean there could not be committed granules in higher areas; it just means it does not know better:

```
+------------------------------+--------------------+------------------+
|     used                     | unused committed   | unused unknown   |
+------------------------------+--------------------+------------------+
```

```
^                       ^                ^                ^
base                    used_words       committed_words  end.
```

So, space below committed_words is guaranteed to be committed; beyond that `Metachunk` has to make sure by bothering `VirtualSpaceNode`.

---

A chunk can of course be smaller than a commit granule. In that case it shares that granule with its neighboring chunks. Since a commit granule can only be committed or uncommitted this means that if one of these chunks is "in-use" and needs to be committed, all chunks in this granule are committed.

---

Note that a chunk knows nothing about granules beyond their size, as an alignment hint for talking to `VirtualSpaceNode`. It just asks `VirtualSpaceNode` to commit a range which may or may not cover multiple granules.

##### 2.3.2.1.2. Metachunk::allocate()

`Metachunk::allocate()` is the central access to pointer bump allocation from a chunk. It takes care of on demand committing the underlying memory and moves the top pointer up.

#### 2.3.2.2. class MetaspaceArena

`MetaspaceArena` manages the in-use chunk list for a class loader.

It has a current chunk, which is used to satisfy ongoing Metadata allocations. It also has a list of "retired" chunks, which are chunks which are completely or almost completely filled with Metadata. It safekeeps the chunks until the class loader dies and the `MetaspaceArena` is destroyed, to return them to the ChunkManager for reuse.

It also has a `FreeBlocks` object, which takes care about deallocated blocks - see *Deallocation Subsystem* below for details.

##### 2.3.2.2.1. MetaspaceArena::allocate()

`MetaspaceArena::allocate()` is the central access point to allocate a piece of Metadata for a class loader.

It will first attempt to take memory from the `FreeBlocks` structure (see below).

Failing that, it will first attempt to take memory from the current chunk via pointer bump allocation - see `Metachunk::allocate()`.

Failing that, it will employ various strategies to get more memory: it may try to enlarge the current chunk, or it may try to get a new chunk from the chunk manager.

##### 2.3.2.2.2. Retiring chunks

When the `MetaspaceArena` gets an allocation request and is unable to fulfill it from the current chunk, because the space left in the current chunk is too small, it will acquire a new chunk. However, we do not want to loose the remainder space in the current chunk.

The remainder space is added to the `FreeBlocks` structure and managed the same way as space deallocated from the outside would - getting reused for later allocations as soon as possible.

#### 2.3.2.3. class ClassLoaderMetaspace

`ClassLoaderMetaspace` is just the connection between a CLD and one or two instances of SpaceManger - normally just one, but if `-XX:+UseCompressedClassPointers`, we need two MetaspaceArenas, one for class space allocations (to put Klass* structures), one for the rest.

It also takes care of increasing the GC threshold when necessary.

Beyond that, it does not have a lot of own logic.

##### 2.3.2.3.1. class ArenaGrowthPolicy

`ArenaGrowthPolicy` encapsulates the logic of "how big a chunk do I give this class loader?".

When a class loader allocates memory, we give it (via MetaspaceArena) a chunk to gnaw on, which should be fine for this requested allocation as well as a number of future allocations. The open question is how large that chunk should be. This is basically a guess toward the future loading behavior of this class loader.

If we know the class loader will only load one or very few classes (e.g. Lambdas, Reflection glue code etc), it makes sense to give the `MetaspaceArena` a small chunk. If we know the loader may load a lot of classes (e.g. the Boot Class loader), we may want to give it a larger chunk.

There is also the notion involved that a class loader "has to prove itself": a standard class loader which we know nothing else about will first be given a few small chunks until we give it larger chunks. How much sense this makes is questionable but as a strategy this seems to work reasonably well.

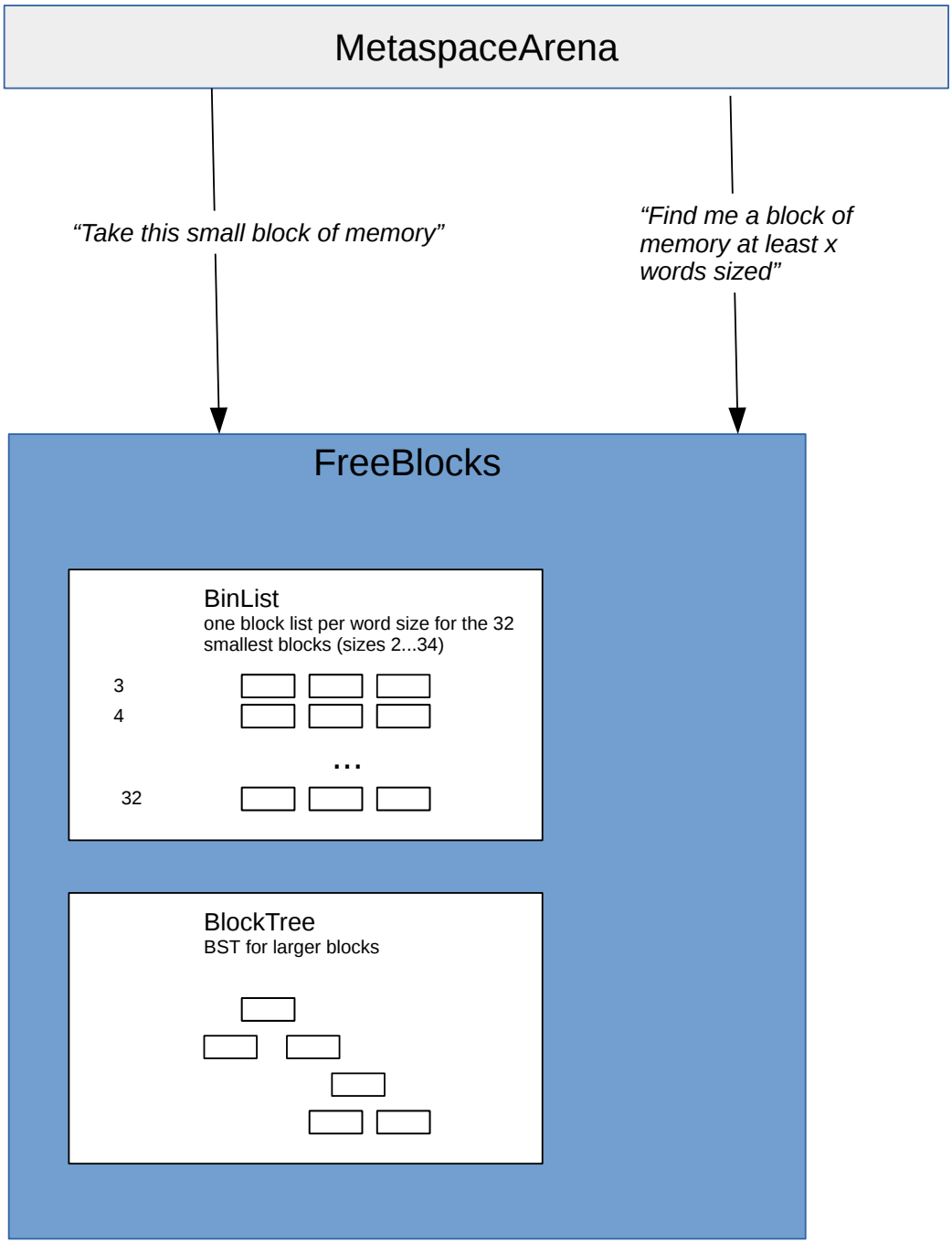
This logic existed in old Metaspace too, in a somewhat convoluted fashion, see `MetaspaceArena::get_initial_chunk_size()` and `MetaspaceArena::calc_chunk_size()`.

In Elastic Metaspace, this logic lives in `ArenaGrowthPolicy`. This is basically just a fancy hard-coded array of chunk sizes marking the handout progression depending on how many chunks the loader already got. One of these arrays exist per use case.

Note that with Elastic Metaspace, one important difference is that we now commit larger chunks on demand. This means when handing larger chunks to a loader we do not have to pay the memory cost upfront, which reduces the penalty for given larger chunks to loaders. So, we can give e.g. a full 4MB root chunk over to the boot class loader even though it may use less (maybe a lot less with CDS involved) and it only will commit the parts it needs.

## 2.4. Deallocation subsystem

## Deallocation Subsystem




Classes:

- FreeBlocks
- BinList
- BlockTree

This is a bit of a sideshow but still important.

The general assumption behind Metaspace is that we deal with arena-style allocation: we have a burst-free scenario and all Metadata go poof when their loader gets collected. However, there are cases when, after allocating Metadata, upper layers decide that memory may not needed after all.

One example is when class load errors happen and the Metadata already loaded are orphaned.

Another example is when classes are redefined and the memory holding the old bytecode is not needed anymore.

In all these cases we have to deal with premature deallocation. These are uncommon, usually rare cases (if they were not we would not use arenas). The caller returns the memory to the Metaspace via `Metaspace::deallocate()`.

Metaspace will attempt to reuse these returned blocks. However, since the blocks are embedded into Metachunks which are in use by a live class loader, these blocks can only be reused by that class loader. Therefore, each class loader (as part of its MetaspaceArena) keeps a structure (`FreeBlocks`) to managed returned blocks. Normally this structure does not see much action, therefore it is only allocated on demand.

Note that this mechanism is also used to manage remainder space from almost-used-up blocks.

The interface is very simple:

- "keep block for future reuse"

  `FreeBlocks::add_block()`

  Adds this block to the manager.

- "give me a block of size x"

  `FreeBlock::get_block()`

  This will attempt to return a block of at least size x. The block may be larger. Internally, the best fit is searched for, and if the best fit is found but considered too large to waste for size x, it is split and the remainder is put back into the manager.

### 2.4.1. Classes

The outside interface is the `FreeBlocks` structure. It itself contains two structures, `BinList` and `BlockTree`.

`BinList` is a simple mechanism to manage small to very small memory blocks and store/retrieve them efficiently. It is somewhat costly in terms of memory (one pointer size per block word size), therefore it only covers the first 16 small block sizes. But since these block sizes are the vast majority of deallocated blocks, it makes sense to pay this cost.

`BlockTree` is a binary search tree used to manage larger blocks. It is unbalanced (though it may be a good idea in the future to make it a red black tree).

## 2.5. Auxiliary code

A collection of miscellaneous helper classes.

### 2.5.1. class ChunkHeaderPool

`ChunkHeaderPool` manages `Metachunk` structures.

Since `Metachunk` structures are separated from the chunk payload areas, they need to live somewhere. We could just allocate them from C-Heap but that would be suboptimal since with buddy style chunk merging and splitting a lot of temporary headers are used.

Therefore `ChunkHeaderPool` exists, which is just a growable array of `Metachunk` structures. It keeps a list of free structures. The underlying memory is allocated from C Heap.

In this sense it is roughly similar to a kernel slab allocator (but not as complex, really).

This not only makes for more efficient allocation and deallocation of Metachunk, it also provides better locality - the chance that headers of linked chunks are allocated close to each other in this pool is high - which makes walking these chunks cheaper.

### 2.5.2. Counters

In Metaspace, a lot of things are counted. This is a lot of boilerplate coding. Helper classes exist which provide counting and various check functions (e.g.overflow- and underflow checking).

These classes live in counter.hpp:

- class SizeCounter
- class IntCounter
- class MemoryCounter

### 2.5.3. MetachunkList and MetachunkListVector

`MetachunkList` is a linked list of Metachunks.

`MetachunkListVector` is a list of `Metachunk` lists. One list per chunk level. The lists only contain chunks of their level.

### 2.5.4. Allocation guards

This is an optional feature controlled by `-XX:+MetaspaceGuardAllocations`. Normally off, if switched on it will add a fence after every Metaspace allocation, and test these fences in regular intervals (e.g. when a GC purges the Metaspace). This can be used to capture memory overwriters.

# 3. Locking and concurrency

Locking in Elastic Metaspace is a simple a two-step mechanism which is unchanged from the old Metaspace.

There is locking at class loader level (`ClassLoaderData::_metaspace_lock`) which guards access to the `ClassLoaderMetaspace`. Ideally the brunt of Metaspace allocations should only need this lock. It guards the access to the current chunk and the pointer bump allocation done with it.

The moment central data structures are accessed (e.g. when memory needs to be committed, a new chunk allocated or returned to the freelist), a global lock is taken, the `MetaspaceExpand_lock`.

(Note: in the future this lock may actually be changed to a lock local to the `MetaspaceContext`.)

# 4. Tests

A lot of tests have been written anew to test the Metaspace.

## 4.1 Gtests

In `test/hotspot/gtest/metaspace` reside a large number of new tests. These are both stress tests and functional tests. These tests are valuable, and in order to get the most out of them, they are executed with different settings (all metaspace reclamation policies, with allocation guards etc) as part of the jtreg tests (see `test/hotspot/jtreg/gtest/MetaspaceGtests.java`).

## 4.2 jtreg tests

Apart from the gtests, new jtreg have been written to stress test multithreaded allocation, deallocation and arena deletion. These tests live inside `test/hotspot/jtreg/runtime/Metaspace/elastic`. They use a newly introduced set of Whitebox APIs which allows for creation of metaspace contexts and metaspace arenas which are independent on the global Metaspace/class space. That allows for isolated testing without interfering with or getting interfered by global VM metaspace.

# 5. Further information

Not vital for this review but may give more information.

- The JEP explains core concepts in greater detail.
- A presentation we gave in March 2020: https://github.com/tstuefe/JEP-Improve-Metaspace-Allocator/blob/master/pres/metaspace2.pdf
- A brief talk we gave at Fosdem 2020: https://www.youtube.com/watch?v=XqaQ-z70sQs
- A series of articles with a bit more depth describing the old Metaspace implementation: https://stuefe.de/posts/metaspace/what-is-metaspace