Metaspace

Note: this Wiki page describes Metaspace in its current form, which has substantially changed in the wake of JEP 387 Elastic Metaspace. Some information in this page may not be applicable for earlier JDK releases.

--- WIP ---

What is Metaspace?

Metaspace is a native (as in: off-heap) memory manager in the hotspot.

It is used to manage memory for class metadata. Class metadata are allocated when classes are loaded. Their lifetime is usually scoped to that of the loading classloader - when a loader gets collected, all class metadata it accumulated are released in bulk. The memory manager does not need to track individual allocations for the purpose of freeing them. Hence, the metaspace allocator is an Arena- or Region-Based Allocator. It is optimized for fast, low-overhead allocation of native memory at the cost of not being able to (easily) delete arbitrary blocks.

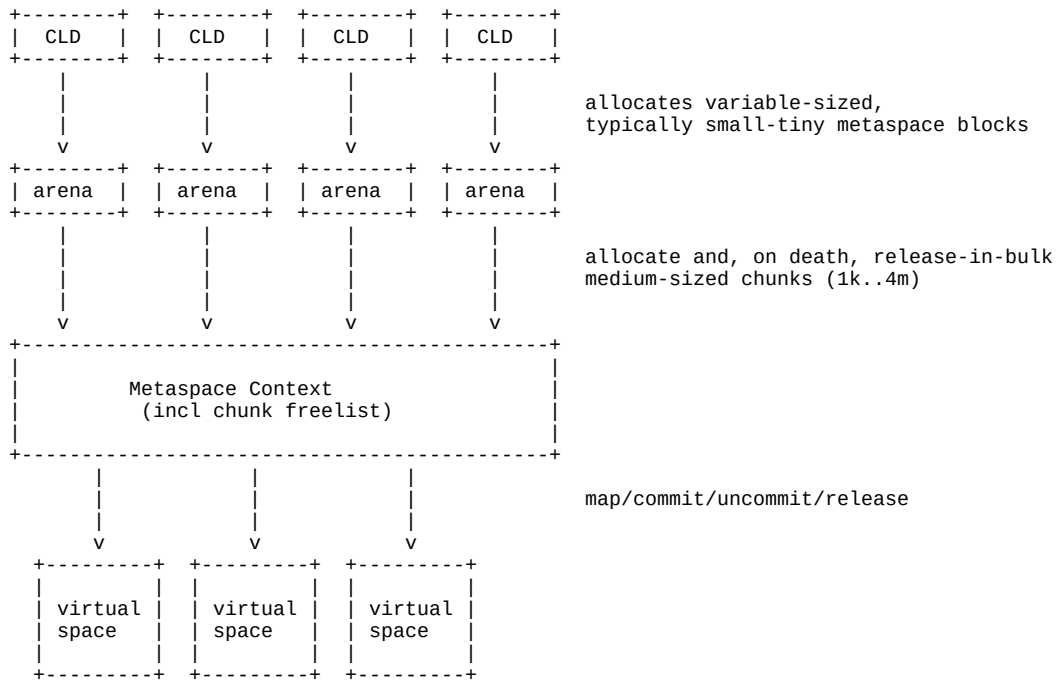
High-level functional overview

A CLD (`ClassLoaderData`) instance owns a `MetaspaceArena`. From that arena it allocates memory for class metadata and other purposes via pointer bump. As it is used up, the arena grows dynamically in semi-coarse steps. When the class loader is unloaded, its CLD is deleted, the arena gets deleted and its memory returned to the metaspace. This memory is kept inside metaspace for later re-use, but metaspace may decide to uncommit parts or all of it as it sees fit.

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

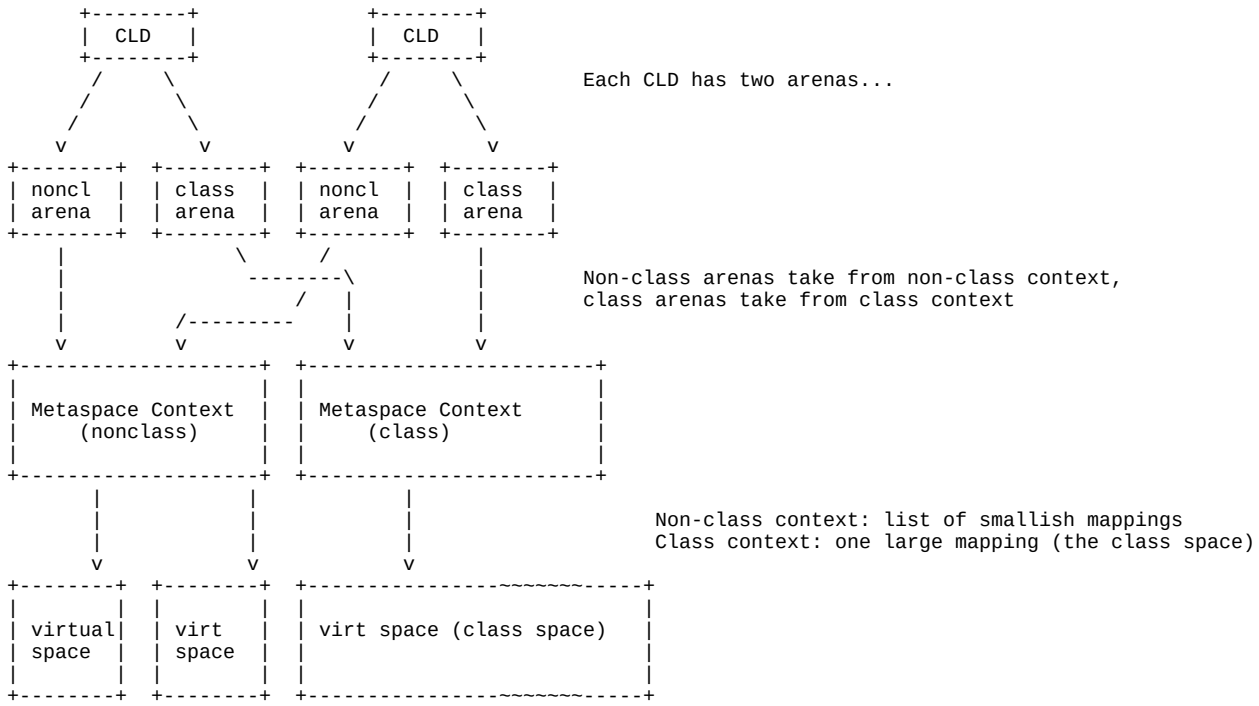
Only one global context exists if compressed class pointers are disabled and we have no compressed class space. If compressed class pointers are enabled, we keep class space allocations separate from non-class space allocations. So we have two global metaspace contexts: one holding allocations of Klass structures (the "compressed class space"), one holding everything else (the "non-class" metaspace). Mirroring that duality, each CLD now owns two arenas as well.

No compressed class space:



With compressed class space enabled, we have two Metaspace contexts (one normal, one wrapping the class space), and each CLD has now two arenas, one associated with non-class context, one with the class space.

In this mode, memory for `Klass` structures is allocated from the class space, all other metaspace allocations from the non-class metaspace:



Core Concepts

Commit Granules

One of the key points of *Elastic* Metaspace is elasticity, the ability to return unneeded memory to the OS, and commit memory only on demand.

Metaspace address space is divided into homogeneously power-of-two sized memory units called *commit granules*. Commit granules are the basic unit of committing and uncommitting memory in Metaspace and therefore dictate the coarseness of committing.

While commit granules may be technically as small as a single page, in practice they are larger (defaulting to 64K). When memory is returned to the metaspace, commit granules which are completely unoccupied are uncommitted.

The commit granule size is a trade-off between efficiency of memory reclamation and certain costs associated with fragmenting the memory map. The smaller a granule is, the more likely it is to be unoccupied and eligible for uncommitting, but at the same time, uncommitting many small areas will increase the number of mappings of the VM process. The default size is 64K, which is a compromise which seems to work very well, only moderately increase the number of mappings while giving us good elasticity.

Granule size can indirectly be influenced via the `MetaspaceReclaimStrategy` switch (see below).

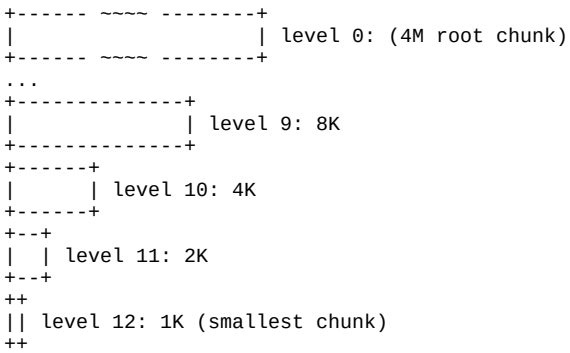
Metachunks and the Buddy Style Allocator

Metaspace arenas will dynamically grow, in semi-coarse steps. Internally they are lists of variable-sized memory areas called *Metachunk* (see `metachunk.hpp`). Arenas obtain these chunks from their respective metaspace context, to which they return all chunks in bulk when they die.

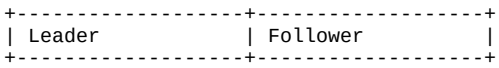
Chunks are variable power-of-two sized, ranging from the largest possible chunk size, 4M - the *Root Chunk* - down to the smallest chunk size of 1K.

Chunks are managed by a power-two-based buddy allocator. A buddy allocator is very efficient in keeping fragmentation at bay, at the cost of limiting the size of managed areas to power of two units. This restriction does not matter in metaspace since these chunks are not the ultimate - user level - unit of allocation, just an intermediate.

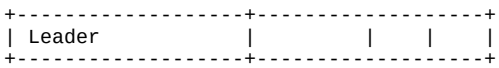
Throughout the metaspace implementation, chunk size is indicated not as size but given as "chunk level" (`chunklevel_t`, see `chunklevel.hpp`). A root chunk has chunk level 0, the next smaller chunk level 1 and so on, down to the smallest chunk with level 13. Helper functions and constants to work with chunk level can be found at chunk_level.hpp.



In buddy style allocation, a chunk is always one part of a neighboring pair of chunks, unless the chunk is a root chunk. In code we use the term *leader* for the chunk with the lower address of the pair, his partner is called *follower*.



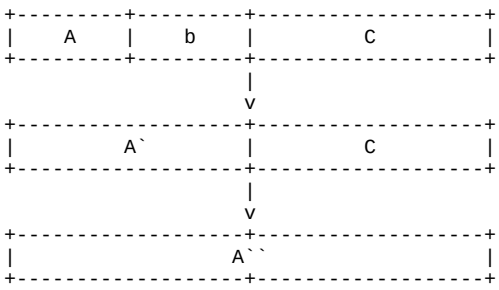
One or both of which could be split into smaller chunks, of course.



Merging chunks

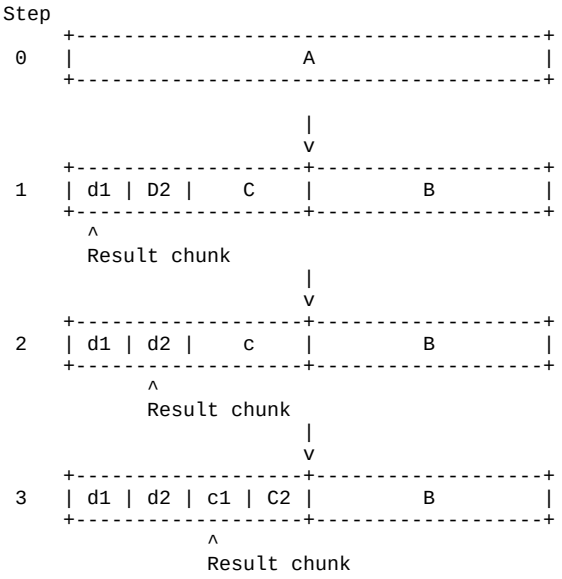
A free chunk can be merged with its buddy if that buddy is free and unsplit. This is done recursively, until either one of the partners in the chunk pair are not free and unsplit, or until the largest chunk size - root chunk size - is reached.

This crystallizes a range of free chunks into one larger chunk quite effectively. In the following figure, chunk b becomes free, melds with free chunk A, then with chunk C:



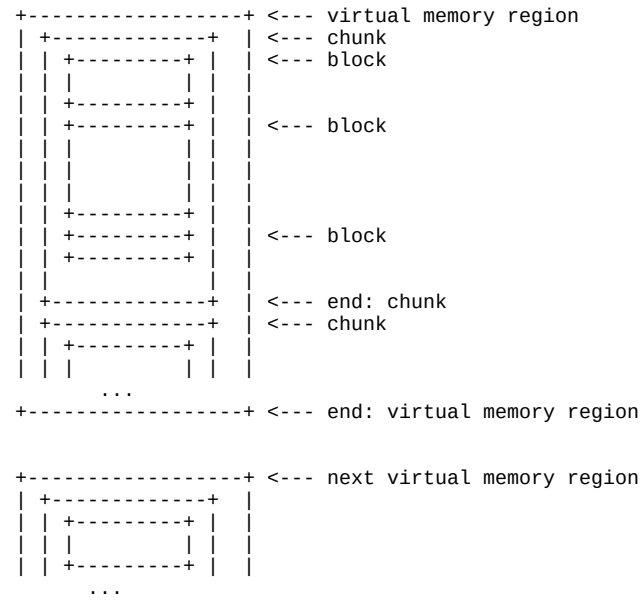
Splitting chunks

To get a small chunk from a larger chunk, a large chunk can be split. Splitting happens in power-of-2 sizes. A split operation yields the desired smaller chunk as well as 1-n splinter chunks.



How it all looks in memory

Allocated metaspace blocks (the user-level unit of allocation) reside in chunks; chunks reside in mappings called `VirtualSpaceNode`, of which multiple may exist:



Subsystems

Metaspace implementation is divided into separate sub systems, each of which is isolated from its peers and has a small number of tasks.

The Virtual Memory Subsystem

Classes:

- `VirtualSpaceList`

- `VirtualSpaceNode`

- `RootChunkArea` and `RootChunkAreaLUT`

- `CommitMask`

- `CommitLimiter`

The Virtual Memory Layer is the lowest subsystem. It forms one half of a metaspace context (the upper half being the chunk manager).

It is responsible for reserving and committing memory. It knows about commit granules. Its outside interface to upper layers is the `VirtualSpaceList` wh ile some operations are also directly exposed via VirtualSpaceNode.

Essential operations

- "Allocate new root chunk"

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

  This carves out a new root chunk from the underlying reserved space and hands it to the caller (nothing is committed yet, this is purely reserved memory).
- "commit this range"

  ```
  bool VirtualSpaceNode::ensure_range_is_committed(MetaWord* p, size_t word_size);
  ```

  Upper layers request that a given arbitrary address range should be committed. Subsystem figures out which granules would be affected and makes sure those are committed (which may be a noop if they had been committed before).

  When committing, subsystem honors VM limits (`MaxMetaspaceSize` resp. the commit gc threshold) via the commit limiter.
- "uncommit this range"

  ```
  void VirtualSpaceNode::uncommit_range(MetaWord* p, size_t word_size);
  ```

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

- "purge"

  void VirtualSpaceList::purge()

  This unmaps all completely empty memory regions, and uncommits all unused commit granules.

Other operations

The Virtual Memory Subsystem takes care of Buddy Allocator operations, on behalf of upper regions:

- "split this chunk, recursivly"
  ```
  void VirtualSpaceNode::split(chunklevel_t target_level, Metachunk* c, FreeChunkListVector* freelists);
  ```
- "merge up chunk with neighbors as far as possible"
  ```
  Metachunk* VirtualSpaceNode::merge(Metachunk* c, FreeChunkListVector* freelists);
  ```
- "enlarge chunk in place"
  ```
  bool VirtualSpaceNode::attempt_enlarge_chunk(Metachunk* c, FreeChunkListVector* freelists);
  ```

Classes

VirtualSpaceList

`VirtualSpaceList` (virtualSpaceList.hpp) encapsulates a list of memory mappings (instances of `VirtualSpaceNode`). This list can be expandable - new mappings can be added on demand - or non-expandable.

The non-expandable latter case is used to represent the class space, which has to be a single contiguous address range for compressed Klass* pointer encoding to work. In that case, the class-space VirtualSpaceList just contains a single node, which wraps the whole of the class space. This may sound complicated but is just a matter of code reuse.

VirtualSpaceNode

`VirtualSpaceNode` (virtualSpaceNode.hpp) manages one contiguous memory mapping for Metaspace. In case of the compressed class space, this encompasses the whole pre-reserved address range of the class space. In case of the non-class metaspace, these are smallish mappings, by default two root chunks in size.

VirtualSpaceNode knows about commit granules, and it knows which commit granules in it are commited (via the commit mask).

VirtualSpaceNode also knows about root chunks: its memory is divided into a series of root-chunk-sized areas (`class RootChunkArea`). To keep coding simple, we require each memory mapping to be aligned to root chunk size in both start address and size.

Note: the concepts of chunks and of commit granules are almost completely independent from each other. The former is a means of handing out/taking back memory while avoiding fragmentation; the latter a way to manage commit state of memory.

Putting this all together, the memory underlying a VirtualSpaceNode looks like this:

```
base                                                                    end
 |                                                                       |
 v                                                                       v

 | root chunk area | root chunk area | root chunk area | root chunk area |

 |x| |x|x|x| | | | |x|x|x| | | |x|x| | | |x|x|x|x| | | | | |x| |x| | | | |    <-- commit granules (x=committed)
```

CommitMask

(commitMask.hpp)

Just a bit mask inside a VirtualSpaceNode holding commit information (one bit per granule).

RootChunkArea and RootChunkAreaLUT

(rootChunkArea.hpp)

`RootChunkArea` contains the buddy allocator code. It is wrapped over the area of a single root chunk.

It knows how to split and merge chunks. 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`).

A `RootChunkArea` object does not exist on its own but as a part of an array within a `VirtualSpaceNode`, describing the node's memory.

`RootChunkAreaLUT` (for "lookup table") just holds the sequence of `RootChunkArea` classes which cover the memory region of the `VirtualSpaceNode`.

CommitLimiter

(commitLimiter.hpp)

When committing memory for Metaspace, we have to observe two limits:

- MaxMetaspaceSize (the total cap on the amount of memory we are allowed to commit for metaspace)
- The GC threshold, which puts a stop-gap into Metaspace growth where, before allowed to grow further, a GC is triggered to attempt class unloading.

(Note: CompressedClassSpaceSize - the reserved size of the class space - is another limit, but it is not explicitly checked in the current implementation simply because it checks itself. If we run out of space in class space we'll notice).

Checking both limits is somewhat complex and also should not be part of a general allocator, therefore limit checking is abstracted away into the class `CommitLimiter`. Allowing potential future reuse of metaspace coding with different limit logics (and also easier testing).

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