We need a new GC

Advancing GC technology for D into the modern age

Dconf ‘24

Amaury Séchet (@deadalnix)   Steven Schveighoffer (@schveiguy)

The current GC is not state of the art

- The current GC is based on mostly the same code that was present in D1/Tango
- We have had several improvements:
  - Precise scanning – no longer as important with 64-bit addressing
  - Forked scanner – Optional performance improvement, but still from Sociomantic days
  - Multi-threaded scanner – Default scanner, Utilizes CPU more efficiently
- Scanner is not optimized for today’s architectures
  - Binary search for blocks
  - Too many touches of various pages per mark
  - Does not take enough advantage of modern addressing
  - Does not try to reduce TLB pressure

Implementation limitations

- All allocations are done from a global structure
- A global lock protects threads from invalidating memory allocation
  - No allocations while collecting
- All threads must be stopped during scanning (except for forking collector)
  - Fork creates a write barrier on all memory to perform COW functionality.
- Implementation is tied to druntime
  - All type data is fetched via TypeInfo
  - Array runtime is dependent on implementation details of GC

Replacing C allocator with our allocator – in user code

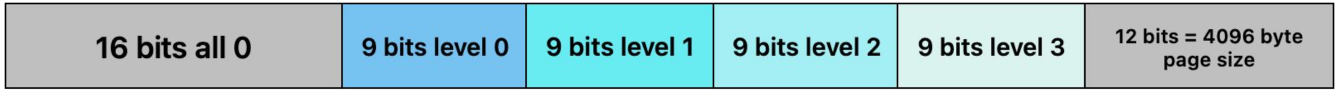
- Many people use the C allocator as a “faster” allocator.
    - But you still have to pin the memory if it might contain pointers, or you might accidentally free something.
- D’s allocator should be just as fast, and performant.
- std.container.array.Array
    - Allocate with C malloc, pin in the GC – Shouldn’t be necessary!
- std.typecons.refCounted
    - Allocate with C malloc, pin in the GC – Shouldn’t be necessary!
- std.stdio.File
    - Allocate with C malloc, pin in the GC – you know the drill.

Only advanced features for 64-bit arch

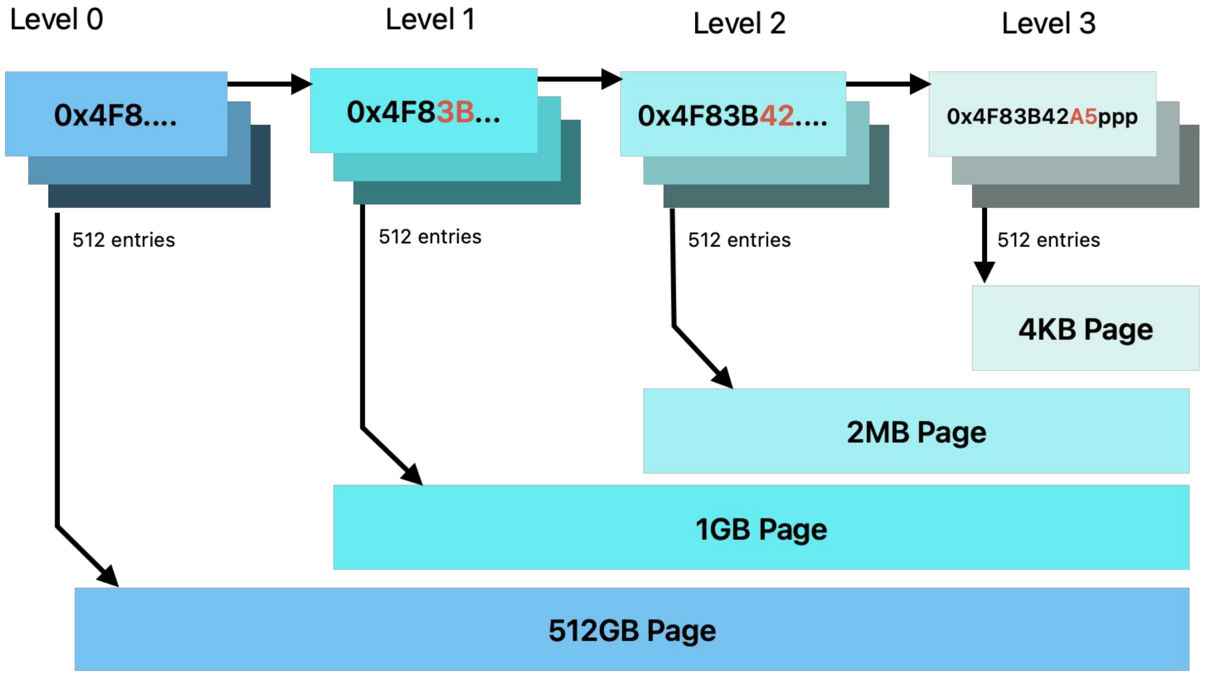
Our GC takes advantage of 64 bit addressing

- All current archs use only 48 bits for addressing (256TB of address space)
- Using memory mapping, we can plan out how memory is laid out to take advantage of hardware architecture
- We can use sparse data structure
  - Uses a lot of address space, but most of it is virtual zeros

How memory HW and OS work


16 bits all 0
9 bits level 0
9 bits level 1
9 bits level 2
9 bits level 3
12 bits = 4096 byte page size

Page Table

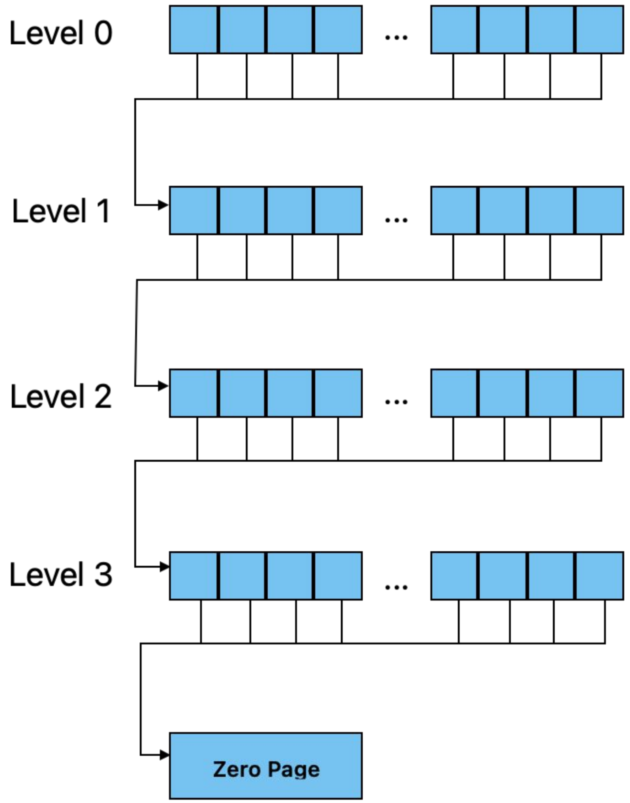

Level 0
Level 1
Level 2
Level 3
0x4F8....
0x4F83B...
0x4F83B42....
0x4F83B42A5ppp
512 entries
512 entries
512 entries
512 entries
4KB Page
2MB Page
1GB Page
512GB Page

Transparent Huge pages

- Huge pages (2MB, 1GB) are profitable because one entry in the TLB maps to more memory.
  - TLB space is finite, we can cache more memory in less space.
  - Temeraire shows that this is very profitable for programs using a lot of RAM.
- We try to cluster allocation on aligned 2MB blocks.
- When enough consecutive 4k pages are used to form a 2MB huge page, the OS can swap these pages for a huge one transparently.

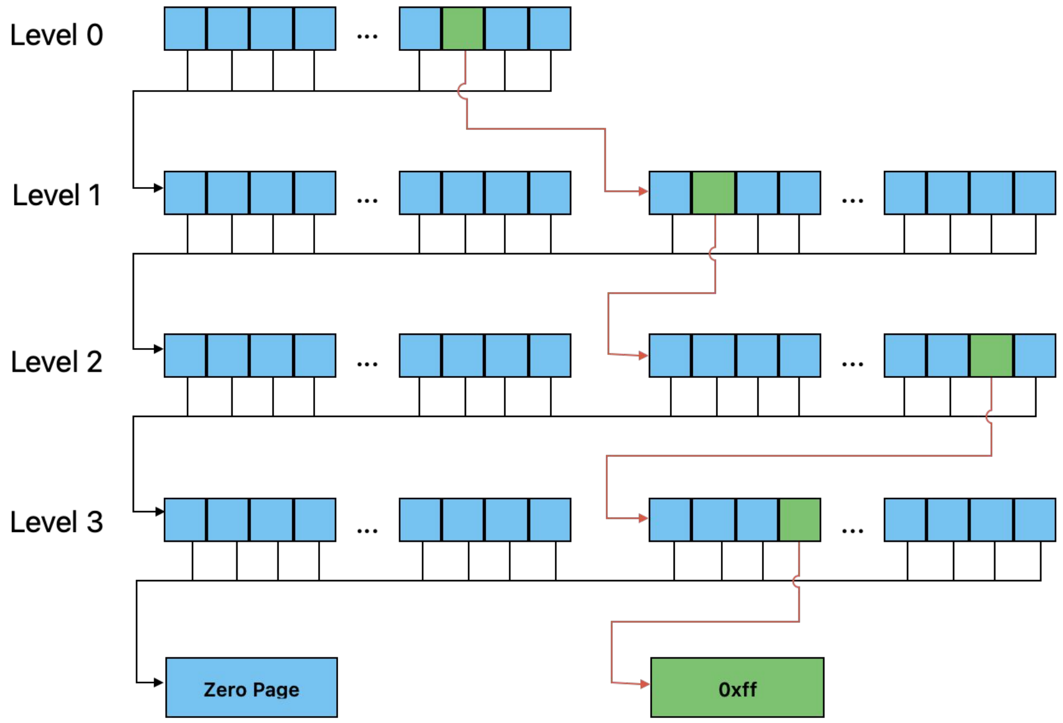
All zero memory page table

- Each level points to the same next level page
- All pages point to a read-only zero page the kernel uses for all processes
- Entire memory can be described using 4 4KB pages



Add one page that is not zero

- Level 0 always one page
- Most entries still point to zero page
- Only one path leads to the non-zero page
- More efficient to keep modified pages together



How the new GC Works

Part 1: Large allocations

Use own mutex to avoid initialization or allocation

- We want our mutex implementation to never allocate.
  - For obvious reason, this would be a problem within the GC.
- We want all zero to be a valid state for the mutex.
  - We want to use virtual zeros to provision static data structures that will only be materialized if used.
  - Some data structures in the allocator need mutexes, so all zero mutexes need to be valid.
- When not contended, atomic operations are enough to take/release the mutex.
- Under contention, if a thread needs to be paused, we the kernel’s futex API.

Region allocator

- Responsible for getting address space from the OS
- Requests allocations of 1GB at a time
  - All memory requests from OS are virtual all-zero pages
- Track ranges of memory that are not used in a Red-black tree
- 2 regions – one for pointer-containing blocks, one for non-pointer blocks
- Region allocator provides chunks of memory in 2MB blocks

Blocks: 2MB pages

- Blocks are managed by a Page Filler
- Requested from the appropriate region allocator, keeping pointer-containing blocks separate from pointer-free blocks
- Tracked in heaps ordered by longest-free-range and number of allocations
- Heaps divided into small allocation heaps, and large allocation.
    - Track which heaps have any available blocks.
    - Only 32 heaps, so constant bitmask operation to find best non-empty heap
    - Heaps operations take amortized constant time
- Selecting the block to use takes amortized constant time

Metadata Tracking

- Some allocations dedicated to tracking metadata about all allocated memory
- BlockDescriptor
  - Keeps track of allocated pages (which ones, and how many)
  - Remembers the longest free page run
  - Allocation class (which bucket of longest free range)
  - How many allocations in the block
  - Dirty pages (which ones, and how many)
  - Generation (to help keep descriptors close together)
- All metadata is allocated outside normal allocation path, never freed
- Unused metadata goes into a heap (ordered by generation/address)

Pairing heap data structure

- Standard heap with N children per node
- When inserting a new child:
    - Better than the root? It becomes the new root
    - Worse than the root? It goes into an auxiliary heap
- Whenever a child is added to the auxiliary heap:
    - If count(Heap_{aux}) is even, perform N merges, where N is the number of trailing zeros
    - In practice, this amortizes to 1 merge per insertion.
- When removing the root:
    - Perform a merge on the root’s children (generally 1-2 children)
    - Perform a merge of the new root with the aux list root

Pairing Heap Demo!

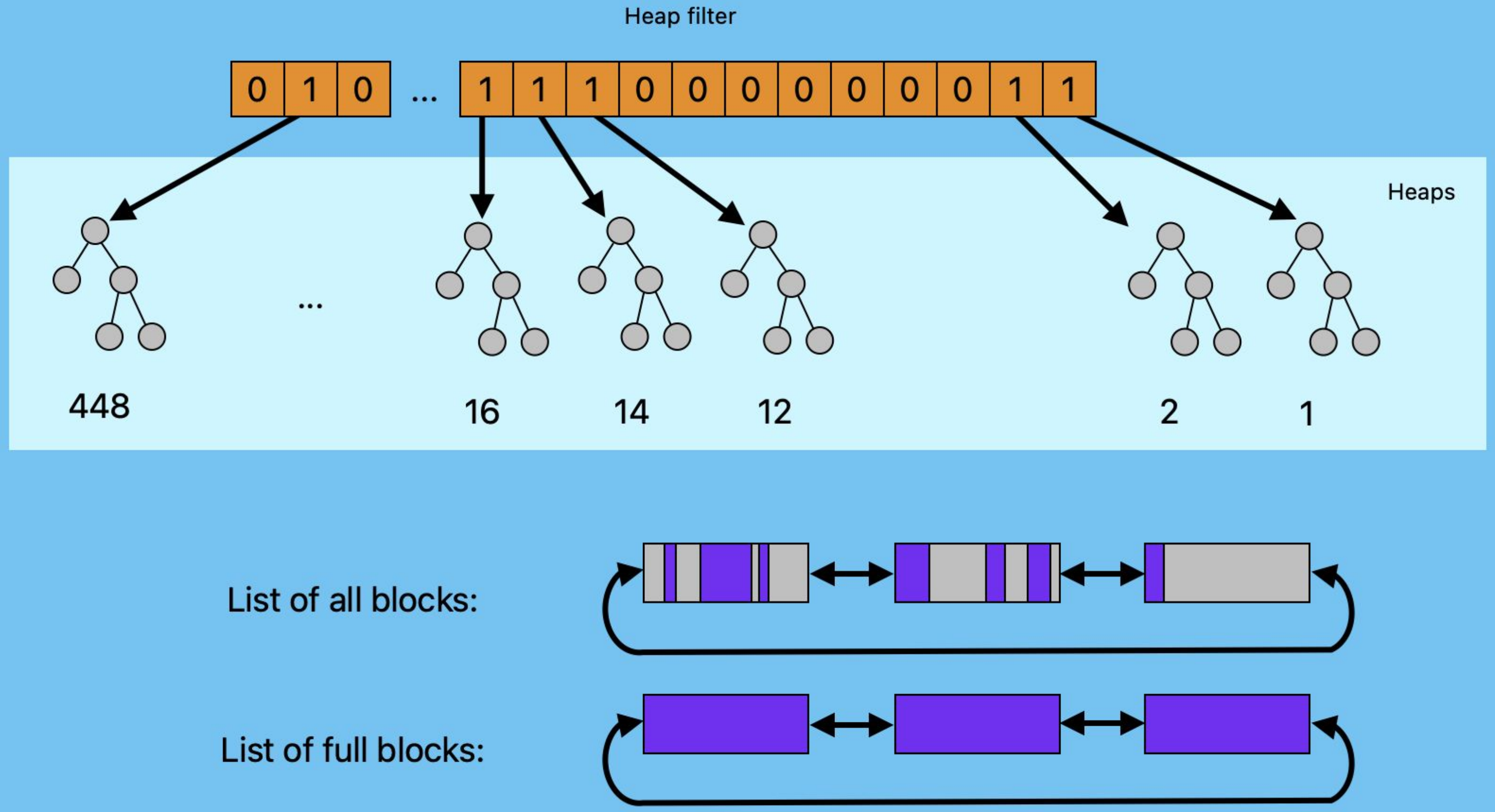
Allocation classes

- We bucket allocation sizes (in pages) in classes.
- Growth rate from one class to the next must be
  - Greater than 1 (or it doesn’t grow)
  - Smaller than φ, the golden ratio, to ensure that space left behind can eventually be reused.
- We settled on size classes with two bits of precision.
  - Each time we use one more bit, we grow the step between classes.
  - 1, 2, 3, 4, 5, 6, 7, 8
  - 10, 12, 14, 16
  - 20, 24, 28, 32,
  - 40, 48, 56, 64
  - 80, 96, 112, 128
  - 160, 192, 224, 256
  - 320, 384, 448, 512 (empty block, we release it to the region allocator).
- In total, 31 allocations classes are used.

Page Filler

- The page filler gets blocks from the region allocator and allocates pages out of them.
- There are 4096 pre allocated page filler as static data.
    - The initial state of a page filler is all zeros, so it doesn’t use any actual memory.
    - The page filler is selected based on the core the the thread runs on.
        - This ensures no contention, unless the OS reschedules the thread mid allocation.
    - When a new page filler is used, the OS automatically provides memory for it.
- The page filler maintains a set of heaps of blocks for each allocation class.
    - Favor blocks with shorter free ranges to give the larger runs the opportunity to grow.
    - Favor blocks with fewer allocations as they are more likely to be freed.

Page Filler


Heap filter
0 1 0 … 1 1 1 0 0 0 0 0 0 0 1 1
Heaps
448
…
16
14
12
2
1
List of all blocks:
List of full blocks:

Large allocations

- Large allocations are done at a page granularity.
- They are used when allocating > 14kB
- The allocation class required is computed and the correct heap selected
    - This creates some slack. This on purpose. Preferring best fit tends to create more very small runs than the application demands.
    - For instance, a 21 page allocation will look into buckets starting with the 24 pages one.

Large allocation: Extent

- An extent tracks an allocation of any number of pages (page-level granularity).
- Allocations less than 2MB are allocated using the Page Filler heaps, finding the best matching block
- An extent “descriptor” uses 128 bytes to describe everything about a large allocation.
  - Allocated and never freed, like Block Descriptors

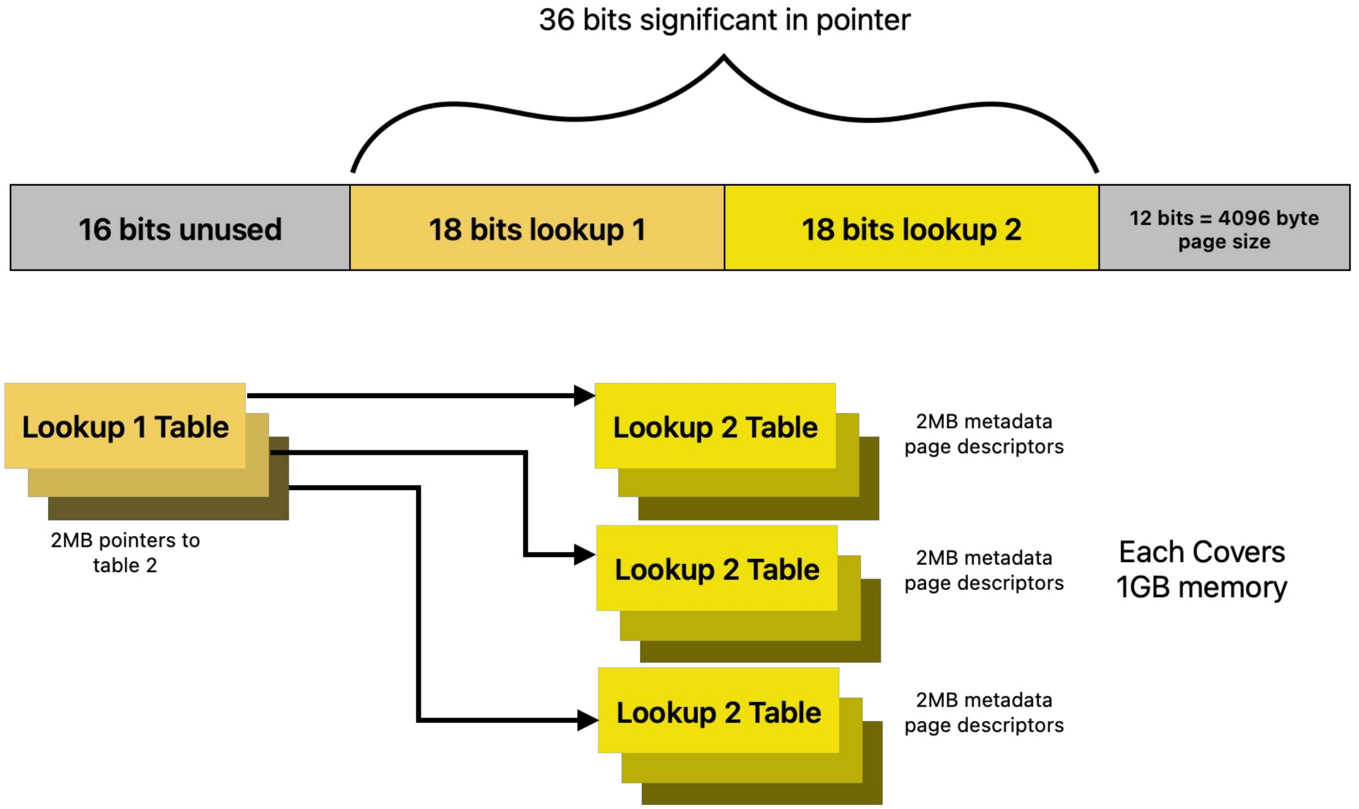
Huge allocations

- Allocations larger than 2MB, requests consecutive new blocks from the page filler
- Last page can be filled with other smaller allocations if there is room.
- Should be deprioritized for filling, as freeing one large allocation can return a large amount of used memory to the system.

Reverse lookup of allocations

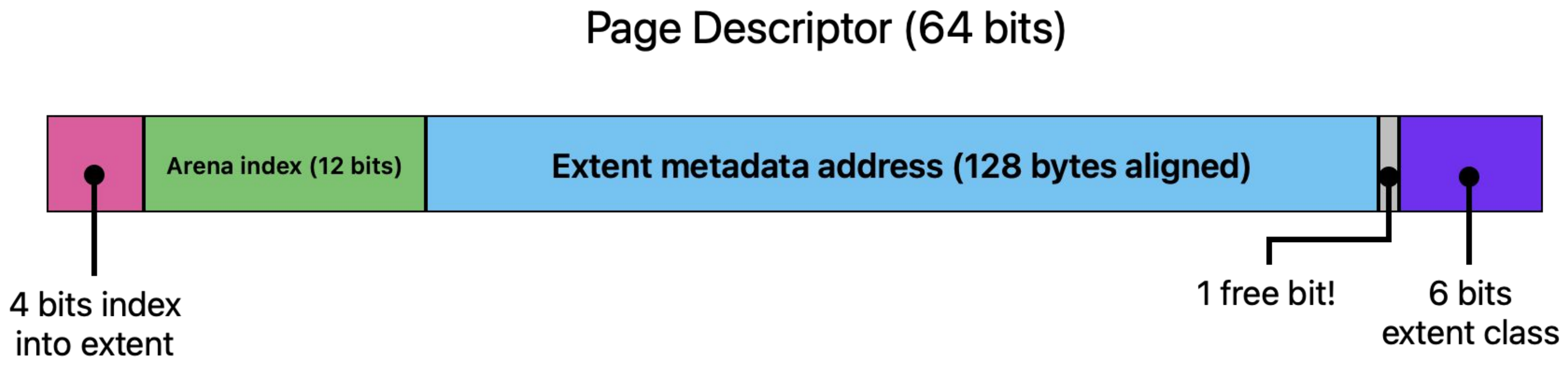
- When allocating, we maintain a 2 level radix tree for reverse lookup.
- Level 1 is a 2MB buffer of static data filled with virtual zeros containing pointers to the next level in the radix tree.
- Level 2 is made of 2MB buffers containing pointer size page descriptor, that contain information about what’s allocated on a given page.
- The whole data structure is lock free and can be accessed by multiple threads at once.

Extent Map (2-level Radix Tree)



Page descriptor

Information about the allocation behind a pointer



Procedure to free large allocations

- Use the radix tree to find the extent, the block and the page filler.
- Clear the page descriptor from the radix tree.
- Remove the block from its heap in the page filler.
- Free the pages in the block, and recompute associated metadata.
- Reinsert the block in the appropriate heap, or release it to the region allocator if it is now empty.

Part 1: The end

?

We need a new GC
Episode II

Advancing GC technology for D into the modern age

Dconf ‘24

Amaury Séchet (@deadalnix) Steven Schveighoffer (@schveiguy)

How the new GC Works

Small allocations

Size classes

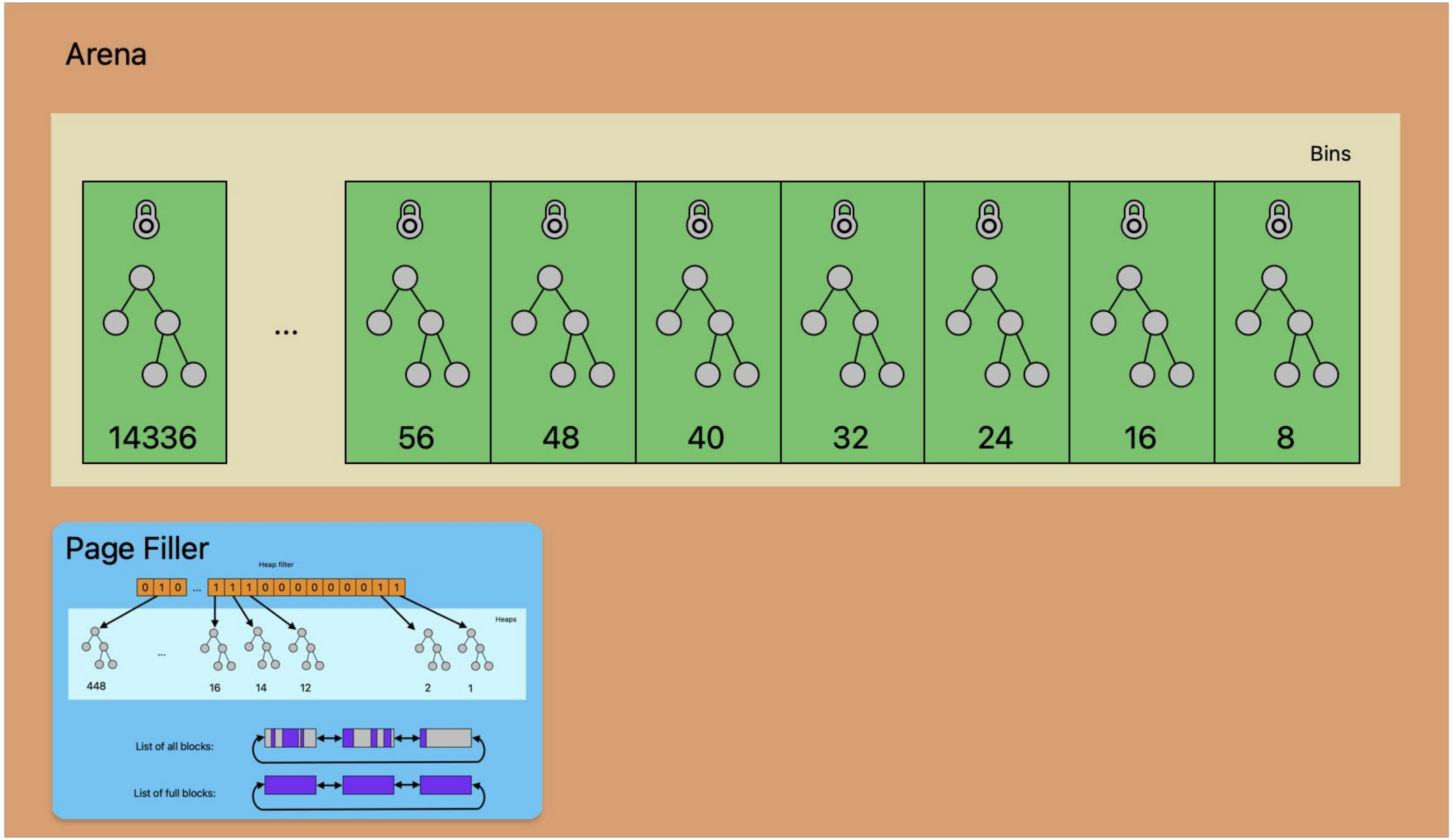
- Similar to allocation classes used for blocks.
- Uses 2 bits of precision for a total of 38 size classes.
  - 8, 16, 24, 32, 40, 48, 56, 64
  - 80, 96, 112, 128
  - 160, 192, 224, 256
  - 320, 384, 448, 512
  - …
  - 10240, 12288, 14336
- Balances internal fragmentation (space lost due to requested size not matching the size class) and external fragmentation (empty slot in slabs due to insufficient demand).
- Extent class is 0 for large, otherwise, size class + 1
- https://github.com/snazzy-d/sdc/blob/master/docs/sizeclass.md

Slab

- A slab is a special kind of Extent that contains several small allocations.
- It also contains a bit field to know which slots are allocated.
- If slot count allows for it, we have another bit field to know whether a slot has associated metadata (append support, finalizer).
- Slab with more than 16 slots are considered “dense” other are “sparse”.
  - A slab with 16+ elements is likely to be long lived or immortal.
  - Dense slabs are allocated on their own set of blocks.
  - Sparse slabs are mixed with large allocations.
- Slabs are allocated from the Page Filler, just like large allocations are.

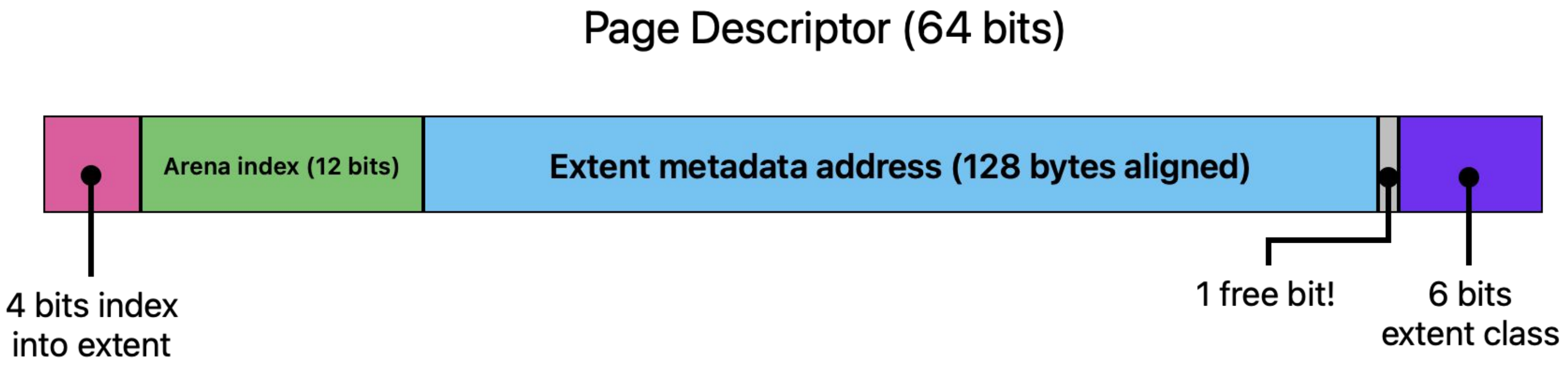
Arena and Bins

- There are 4096 pre allocated arenas with virtual zeros. They contain:
    - One bin per small size class.
    - A Page Filler.
- When needed an arena, and it’s page filler, is picked based on the CPU core used by the thread.
    - This ensures no contention unless the OS reschedule the thread mid allocation.
- Each bin contain a heap of the slabs for a given size class which are not completely full.


Arena
Bins
14336
...
56
48
40
32
24
16
8
Page Filler
Heap filter
0 1 0 ... 1 1 1 0 0 0 0 0 0 0 1 1
Heaps
448
...
16
14
12
2
1
List of all blocks:
List of full blocks:

Page descriptor

Information about the allocation behind a pointer



Find the index of an allocation in a slab

- For each size class, we precompute M and s such as $M / 2^s \approx 1 / size$
  - We chose M and s such as rounding works out for any possible offset in a slab.

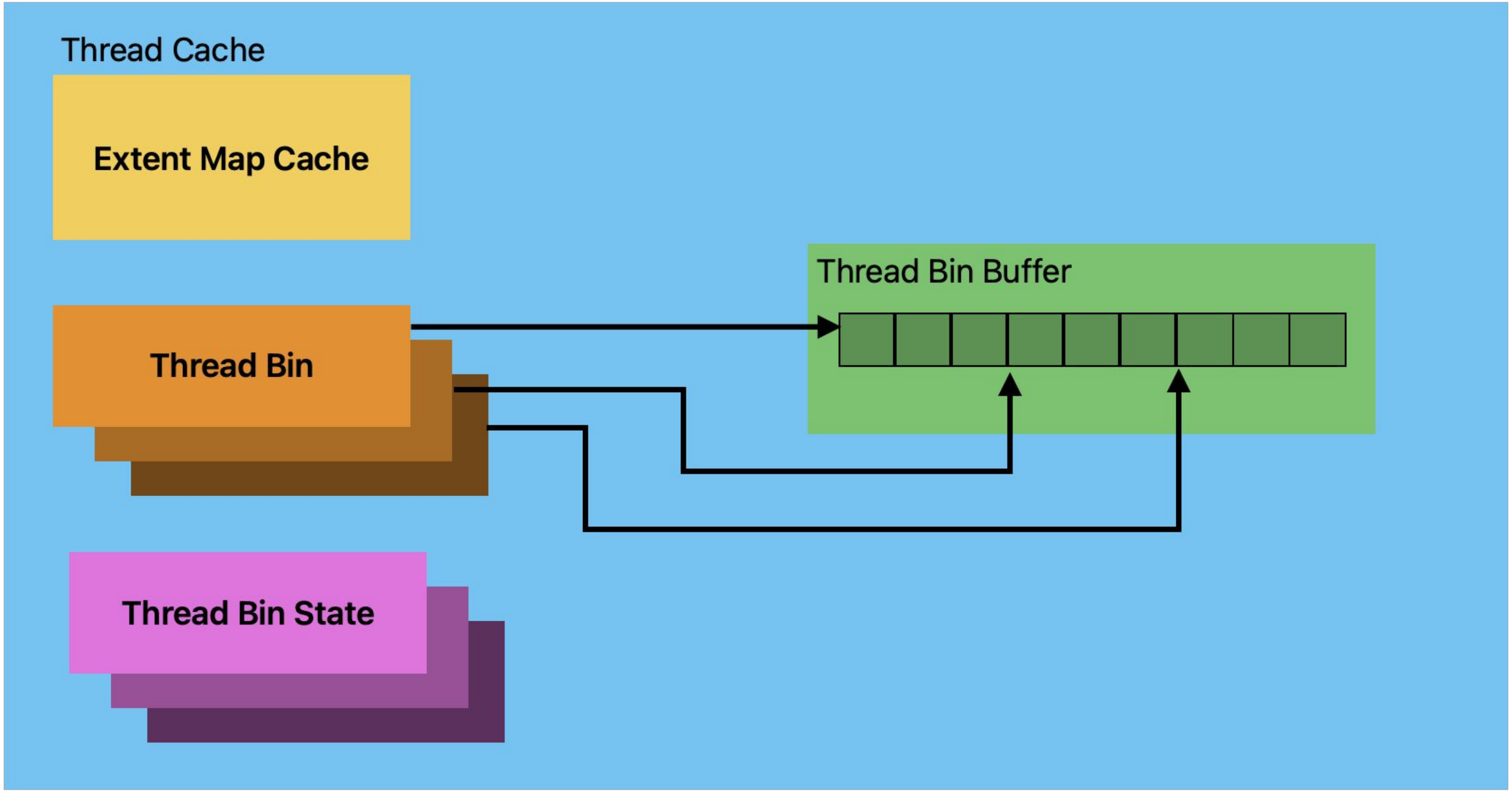
```
auto computeIndex(void* ptr, PageDescriptor pd) {
  auto page = alignDown(ptr, PageSize);
  auto slab = page - pd.index * PageSize;
  auto offset = ptr - slab;

  auto b = binInfos[pd.sizeClass];
  return (offset * b.multiplier) >> b.shift;
}
```

Thread cache

- Each thread has a Thread Cache living in thread local storage.
- The thread cache contains a bin for each small size class.

- When allocating, we look into the cache first.
  - If there isn’t a suitable element, we refill the cache via the arena.
- When freeing, we try to put the element back into the cache.
  - If the cache is full, then we flush it to the bin.

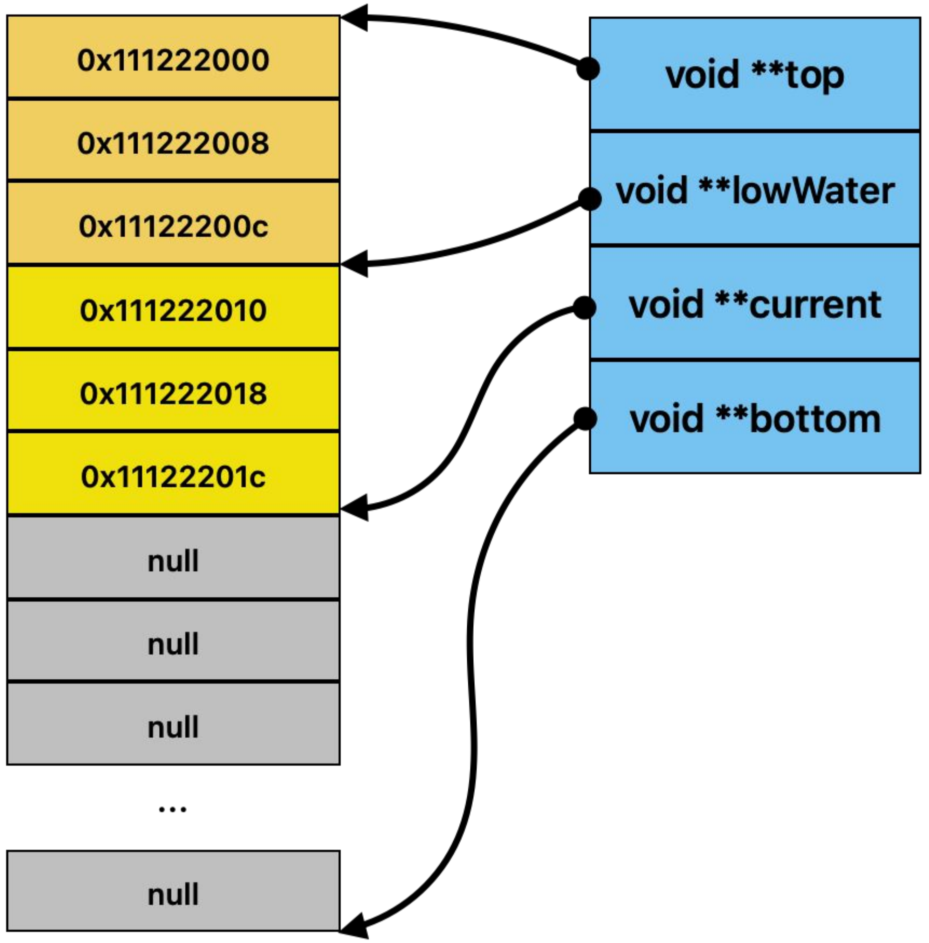
- Batching operations in the arena ensure fast operations most of the time.


Thread Cache
Extent Map Cache
Thread Bin
Thread Bin Buffer
Thread Bin State

Thread bin management

- Periodically, we “recycle” a bin.
  - If we refilled the bin since last time, we increase the batch size for that size class.
  - If the low water mark didn’t reach 0, we flush some of the elements and reduce the batch size.
- At the end of the recycling process, we reset the low water mark for the bin.
  - When we allocate from the bin and are at the low water level, we reduce it.
- This process ensures that the usage of the thread cache adapts to the behavior of the application at run time.
- Elements in the thread cache appear to be allocated from the perspective of the rest of the GC.

Thread Bin



0x111222000
0x111222008
0x11122200c
0x111222010
0x111222018
0x11122201c
null
null
null
...
null
void **top
void **lowWater
void **current
void **bottom

Fast path touches very few pages

- The data about the thread bins are all kept together. If there is a suitable element in them, pick it, return, done.
- That’s only 1 page touched for the allocation fast path!
- We cache entries for the first level of the radix tree in the thread cache.
  - We cannot skip the last level lookup.
  - But we end up staying in the thread cache for the first level most of the time.
  - The cache has 24 entries, which maps to 24GB of address space. If your thread has more than that that is very hot, then what are you doing?
- Then, if there is room, we put the pointer back in the thread bin.
- That’s only 2 pages touched when freeing on the happy path!

How the new GC Works

Appending and finalization

Appendable and finalizers

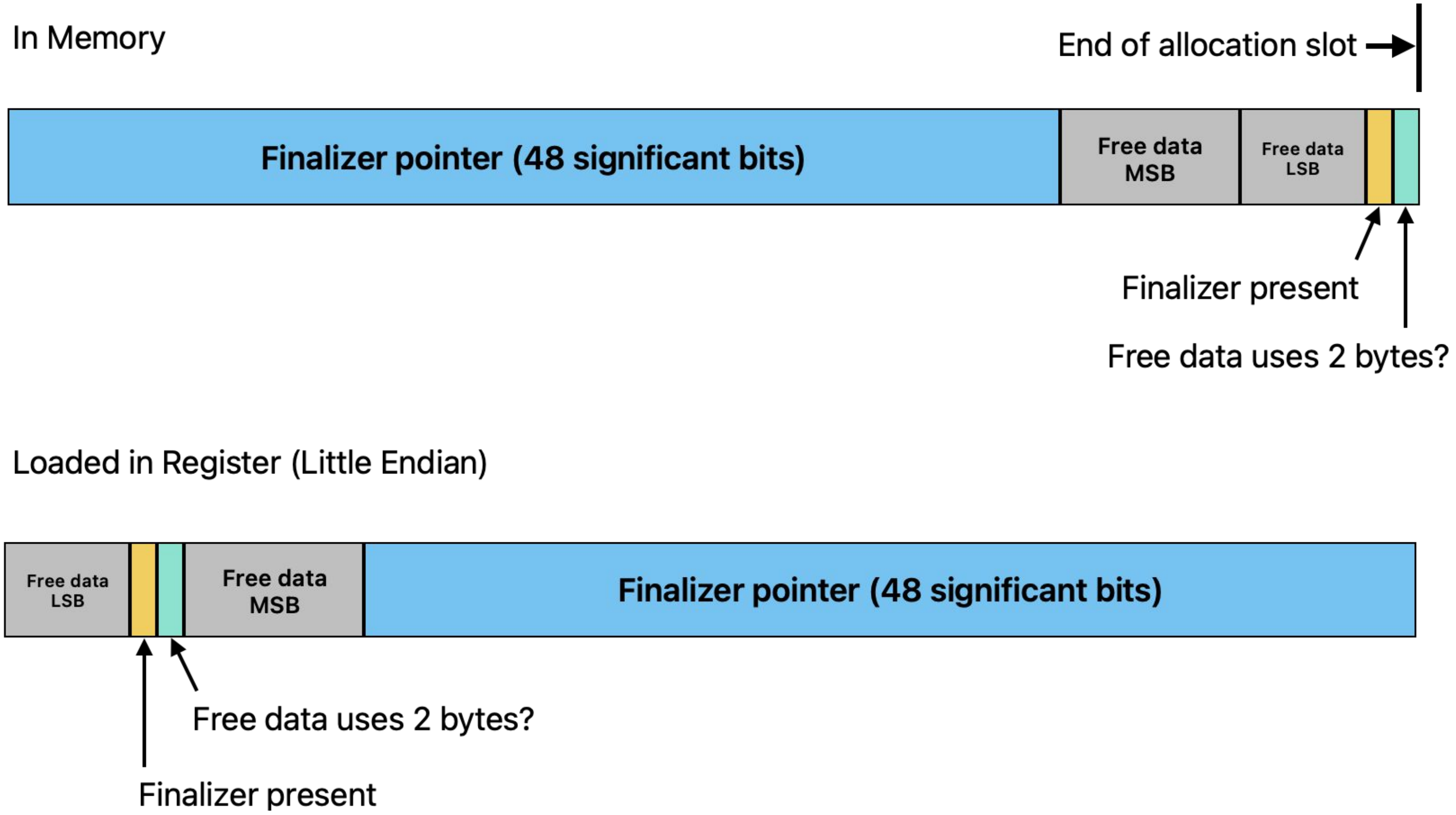
- Allocations/arrays might need to be finalized
- Appending to an array should grow in the allocation if possible
- Need to store a finalizer and a “used” size.
- Druntime always stores these in the allocated space.
- New GC stores data based on the size of the allocation.

Appendable and finalizers

- Large allocations are easy
  - Plenty of space in the Extent record itself, most GC metadata not needed (marking bits, etc.)
  - Store finalizer, used size directly in the Extent
  - Can extend a large Extent without having to touch allocated memory.
- Slabs do not have enough space in the extent to store a pointer per slot
  - 8 bytes needed to store finalizer pointer
  - Store this inside the allocation – at the end
  - 8-byte allocations are never appendable – always reallocate.
- Store “unused” space instead of “used” space.
  - The more space used, the less space we need to store the size.
  - Can fill up the entire allocation, just remove the metadata

Appendable and finalizers - slab metadata

- Max alloc size for slabs is 14336 – only requires 14 bits for “unused” space
- We have 16 free bits in the finalizer pointer! Stuff it in there.
- Set a bit in the Slab Extent to flag which slots have metadata
- With Finalizer present, must always use all 8 bytes
- Without Finalizer present, can use one or two bytes for “unused” space
- 2 flags to determine what state we have – 14 bits + 2 bits for flags.


In Memory
End of allocation slot
Finalizer pointer (48 significant bits)
Free data MSB
Free data LSB
Finalizer present
Free data uses 2 bytes?
Loaded in Register (Little Endian)
Free data LSB
Free data MSB
Finalizer pointer (48 significant bits)
Free data uses 2 bytes?
Finalizer present

Finalizing an allocation

- If metadata is present, and finalizer not null, pass finalizer and used size into handler function
- SDC handler just calls the finalizer with the pointer and valid size
- Druntime hook must do something different (that pesky TypeInfo…)

How the new GC Works

Garbage Collection

Garbage collection

- The GC uses stop the world, parallel, mark & sweep garbage collection.
- The thread invoking the collection process stops all other threads.
- Some preparatory work is done, such as allocating buffers for mark bits.
- Several worker threads are started and marking process begins.
- We then go over all the initialized arenas, go over all blocks, and free all unmarked allocations.

- All in all, fairly standard mark and sweep stuff!

Stopping the world

- We hijack pthread_create to register all threads in the GC and setup signal handling.
- We send SIGPWR to suspend a thread, SIGXCPU to resume.
  - These signals have been used by Boehm for a long time so we assume they are a “de facto” standard.
- We do not want to suspend a thread in a middle of a complex operation.
  - Druntime solves this with a global lock, but we do not want that either.
  - We introduced a busy state to thread where they cannot be suspended.
- When threads aren’t busy, the signal handler for SIGPWR simply uses sigsuspend to wait for SIGXCPU
  - Because we use SA_SIGINFO, the CPU state has been pushed on the stack so we can scan as this.

Hijacking pthread_create to register threads

- We define our own pthread_create function in the GC. The linker picks this one over the system's.
- This function does the preparatory work for the thread to be initialized properly for the GC to manage.
- It then calls a trampoline (function pointer) initialized to do the following:
  - Resolve the system's pthread_create using dlsym.
  - Redefine the trampoline with the system's pthread_create.
  - Forward the call to the system's pthread_create.

Busy state

- When doing critical operation that cannot be suspended, the thread enters the “busy” state.
- When the signal to suspend is received by a busy thread, the signal handler sets the thread for delayed suspension and resume execution immediately.
- When the busy state is exited, the thread checks if delayed suspension was requested.
  - If so, the thread pushes the CPU state on the stack and suspends itself using sigsuspend.
- More complex in practice: this has race conditions galore.
- But we can prevent thread suspension at an undesirable time without any kind of locking mechanism!

Mark phase: Overview

- One thread is created for each core on the machine.
- Thread wait for work to be put on a worklist. Work consist in a range of memory to scan for pointers.
- The main thread start feeding the worklist with global segment, suspend thread’s stack, and various other roots, then wait itself on the worklist.
- When the worklist is empty, and all other threads ran out of work, the thread returns.
- By that time, every alive allocation is marked.

Mark phase: Worker threads

- When the worker’s worklist is empty, it picks 1 range on the shared worklist.
- The worker then goes over each potential pointers in the range.
  - It first does a bound check to see if it is in the range of heap addresses.
  - If it is, it does a lookup in the Extent map to find what’s there.
  - If there is a hit, the worker marks the allocation.
- After marking, if an allocation contains pointers, the worker will add it:
  - If its worklist is empty, it add the range in it.
  - For large allocations, it adds the new range to the shared worklist.
  - For small allocations, it add the new range to its own worklist. This allows the worker to blast through graphs of small objects.
  - If the worker’s worklist grows bigger than 16 elements, it gets partially flushed in the shared worklist.

Sweep phase

- We go over all initialized arenas, over all blocks in these arenas, and over all extents in these blocks.
- We free all the allocations which are not marked.
  - If they have one, we run the finalizer.
- If the block is now empty, we give it back to the region allocator.
- If the block contain more than 16 dirty pages, we purge unused pages from the block.
  - They turn back into virtual zeros.

- Good job! Now we can restart the world!

Druntime Integration

Getting SDC and DMD to play nice

Hooking SDC GC from Druntime is… complicated

- SDC GC does not have access to druntime
- No understanding of TypeInfo
- Supplies its own scheme for array metadata/finalizers
- `qalloc` function returns implementation details of the Druntime GC, but is a public interface
- Odd API functions required (e.g. `collectNoStack`)
- SDC ABI is different from DMD’s!

Migrate Druntime array runtime into GC

- How to manage array “used” size is now GC dependent – must now ask the GC to do all the work.
- New type `ArrayMetadata` which becomes the interface to update the array used size.

```
struct ArrayMetadata {
    ... // private fields
    void *base();
    size_t size();
    size_t _gc_private_flags(); // GC specific flags
    bool setUsed(size_t used, size_t existingUsed = ~0UL);
    bool contains(void *ptr) const @trusted;
}
```

Remove implementation details from interface

- GC interface functions accept `TypeInfo`
- DRuntime GC `malloc` uses `TypeInfo` for precise scanning bitmap and calling finalizer
- Nothing else effectively uses `TypeInfo`, even if it is passed
  - For example, extend accepts `TypeInfo` but ignores it
- But all allocation hooks are now templates!
  - Extract the pointer bitmap from the compiler, pass it directly. We can eliminate `RTInfo`.
  - Have a generic callback data pointer for finalization. But we still use `TypeInfo` here
- Deprecate `qalloc` function. `ArrayMetadata` is now the API for arrays.
- STRUCT_FINAL bit no longer part of API, but used internally by Druntime.

Miscellaneous hook problems

- Most must be extern(C) to be ABI compatible
- GC is not directly referenced from code, so there must be a trick reference to avoid the linker from pruning it.
- Had to add some extra hooks to druntime (e.g. pre/post stop the world) to implement proper thread stopping from SDC
- Class finalization does not fit with SDC scheme – the finalizer pointer is part of the classinfo at the front. So we store a “finalizer” of `cast(void*)1`

Future Possibilities

What else can we pile on?

Bohem’s short pauses.

- During the mark phase, add write barriers over the allocations that contain pointers.
  - This can be done making them write protected with mprotect
  - Alternatively, we can use the dirty bits the CPU fill for us in the page table, but OS API to access these are especially bad and high overhead.
- Stop the world.
- Rerun a mark phase, using the dirty pages as a starting point.
  - This is typically very short, because almost everything alive is already marked.
- Resume the world, collect.

Replacing C allocator with our allocator – for all code

- C code in pre-built libraries would now also allocate using the GC
- Theoretically, you would no longer have to worry about pinning memory you are passing to a C function to keep.
- Still experimental, does not work properly when combined with Druntime.

Miscellaneous items

- PRs for DMD to enable a dub package to try the GC.
- Support more platforms (Windows and OSX are priorities).
- Tweak heuristics based on real world feedback.
- Telemetry and other statistics.

Part 2: The end

?