.NET GC Internals

Allocations

@konradkokosa / @dotnetosorg

.NET GC Internals Agenda

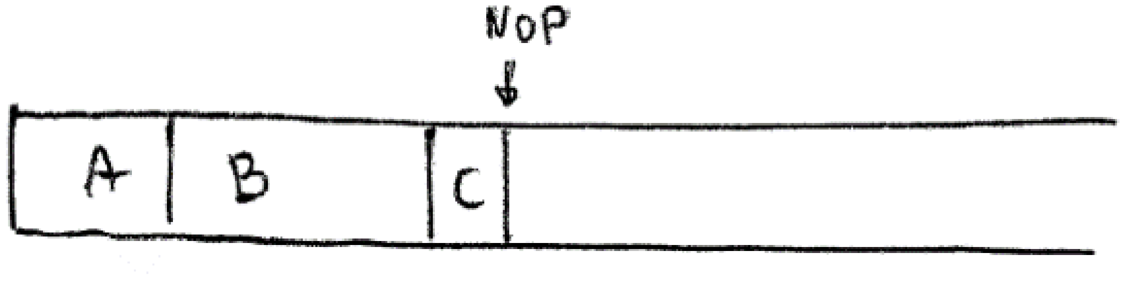
- Introduction - roadmap and fundamentals, source code, ...
- **Mark** phase - roots, object graph traversal, *mark stack*, mark/pinned flag, *mark list*, ...
- **Concurrent Mark** phase - *mark array/mark word*, concurrent visiting, *floating garbage*, *write watch list*, ...
- **Plan** phase - *gap*, *plug*, *plug tree*, *brick table*, *pinned plug*, *pre/post plug*, ...
- **Sweep** phase - *free list threading*, concurrent sweep, ...
- **Compact** phase - *relocate* references, compact, ...
- **Generations** - physical organization, *card tables*, ...
- **Allocations** - *bump pointer allocator*, free list allocator, *allocation context*, ...
- **Roots internals** - stack roots, *GCInfo*, *partially/full interruptible methods*, statics, Thread-local Statics (TLS), ...
- **Q&A** - "but why can't I manually delete an object?", ...

.NET GC Internals Agenda

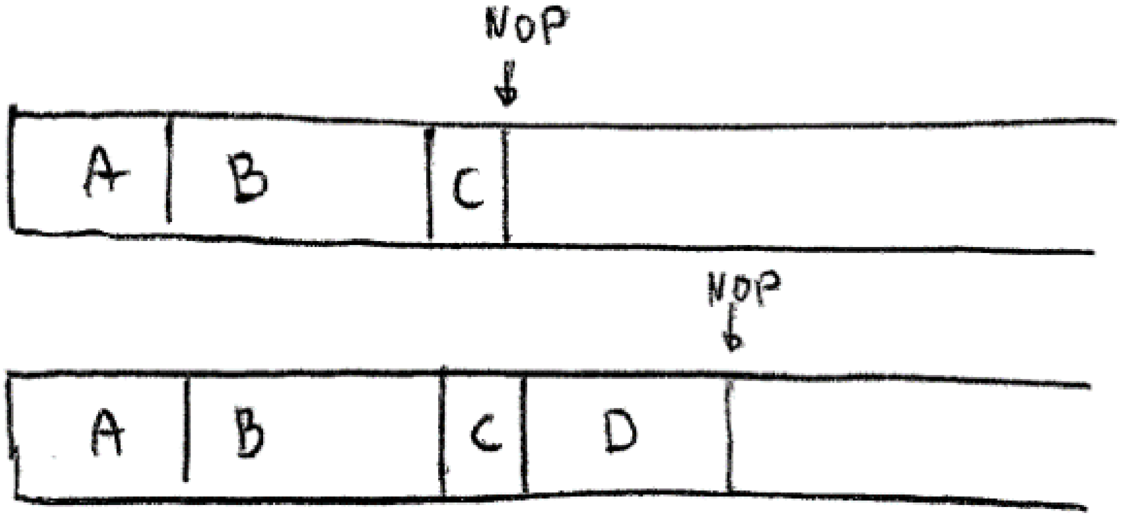
- Introduction - roadmap and fundamentals, source code, ...
- **Mark** phase - roots, object graph traversal, *mark stack*, mark/pinned flag, *mark list*, ...
- **Concurrent Mark** phase - *mark array/mark word*, concurrent visiting, *floating garbage*, *write watch list*, ...
- **Plan** phase - *gap*, *plug*, *plug tree*, *brick table*, *pinned plug*, *pre/post plug*, ...
- **Sweep** phase - *free list threading*, concurrent sweep, ...
- **Compact** phase - *relocate* references, compact, ...
- **Allocations** - *bump pointer allocator*, free list allocator, *allocation context*, ...
- **Generations** - physical organization, *card tables*, ...
- **Roots internals** - stack roots, *GCInfo*, *partially/full interruptible methods*, statics, Thread-local Statics (TLS), ...
- **Q&A** - "but why can't I manually delete an object?", ...

Bump pointer allocator

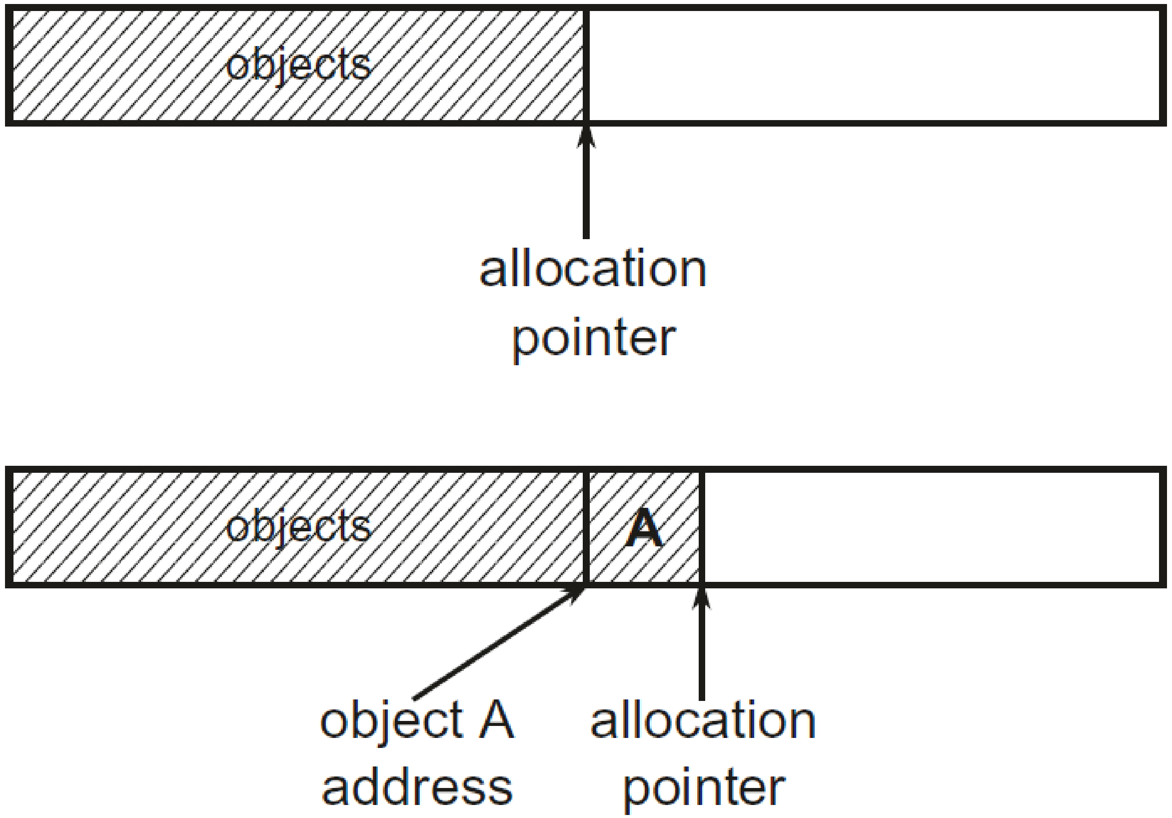
Bump pointer allocator



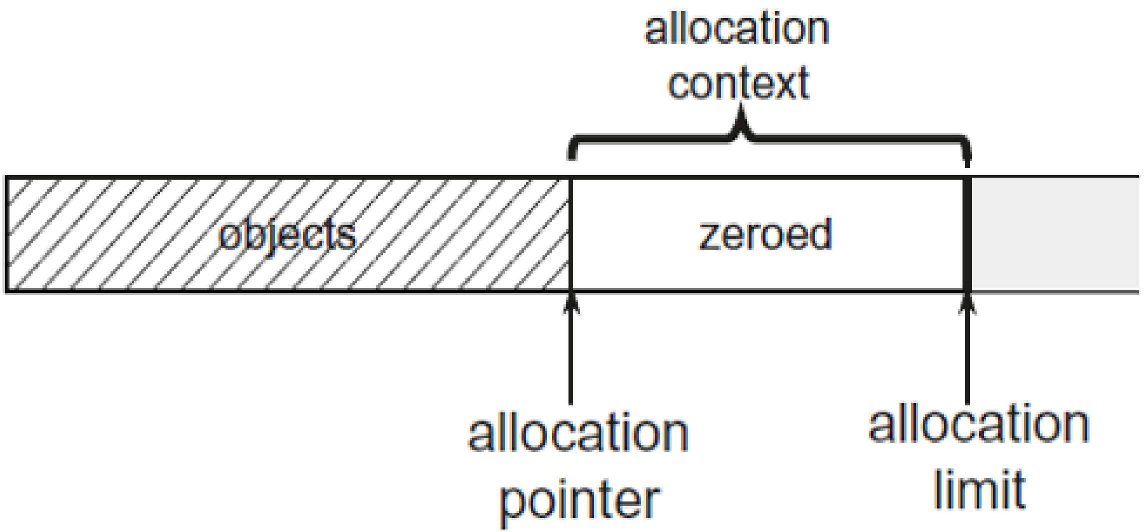
Bump pointer allocator



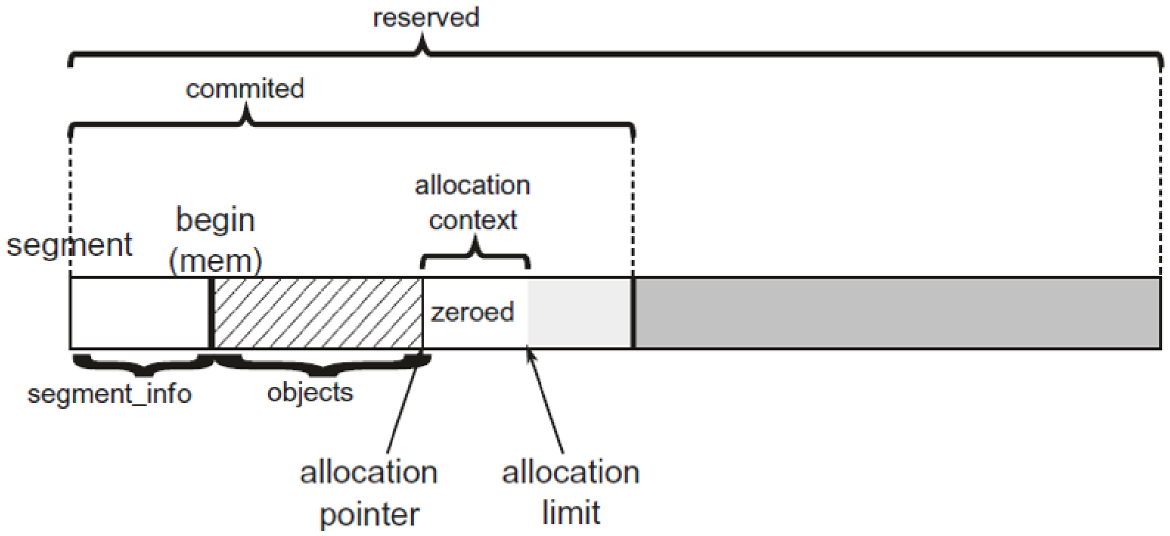
Bump pointer allocator

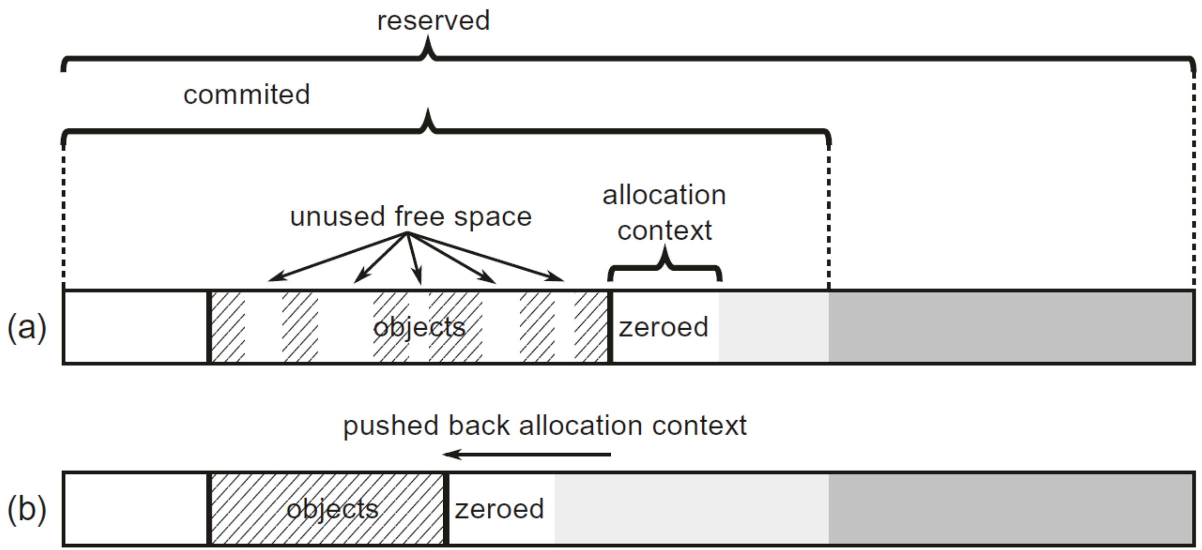


Bump pointer allocator



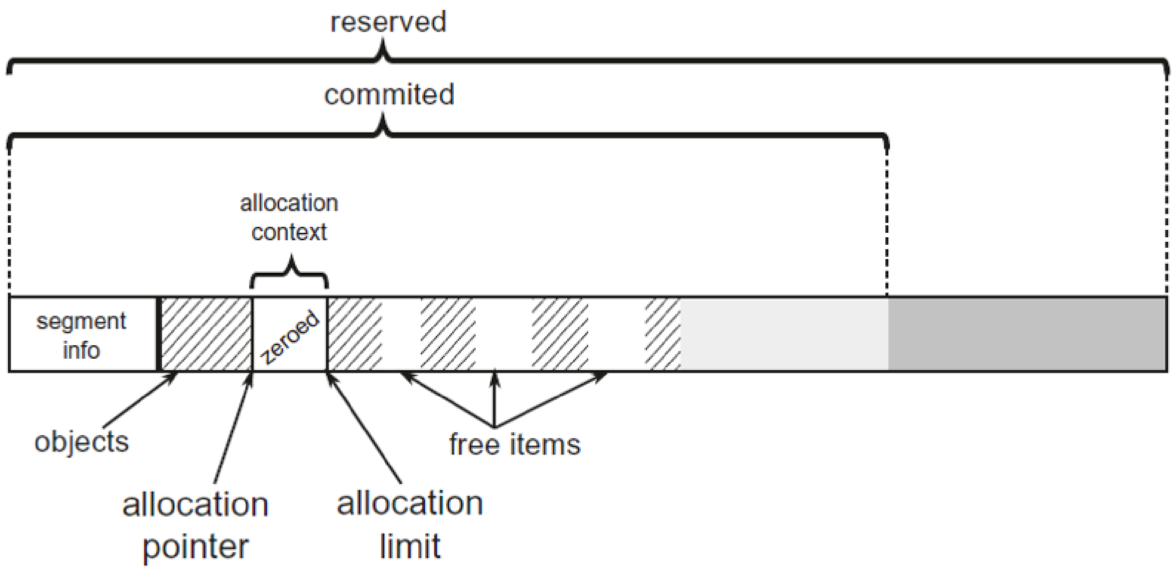
Allocation quantum – 8 kB (1-8kB)


reserved
commited
allocation
context
segment
begin
(mem)
zeroed
segment_info
objects
allocation
pointer
allocation
limit

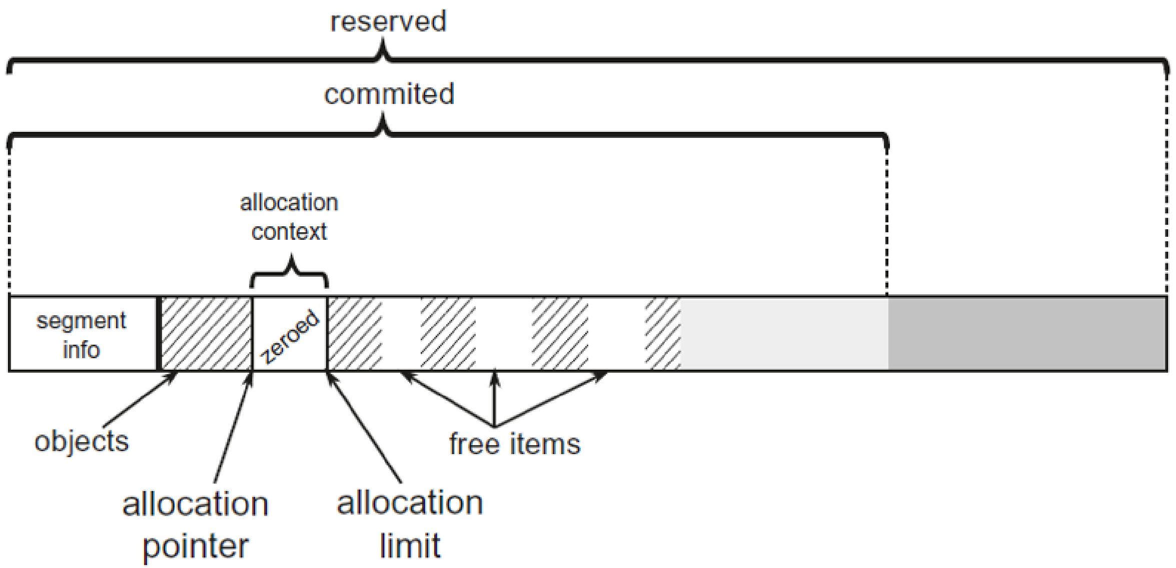


"Dummy" bump pointer allocation and fragmentation problem:

- (a) Sweeping Garbage Collection produces fragmentation and if allocation context is not aware of free memory - sad :(,
- (b) Compact Garbage Collection reclaims memory by pushing back allocation context but requires a lot of memory copying

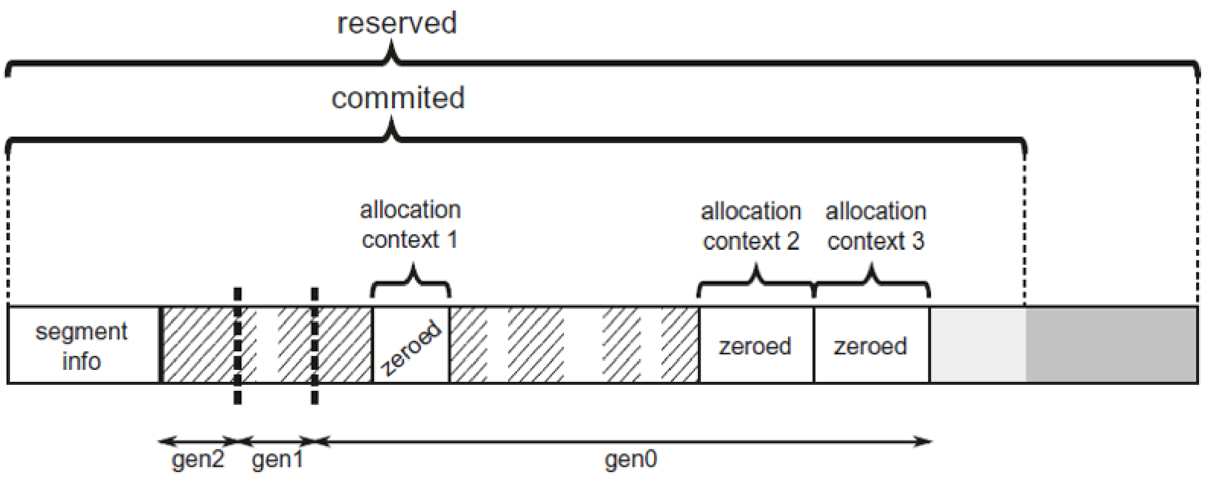


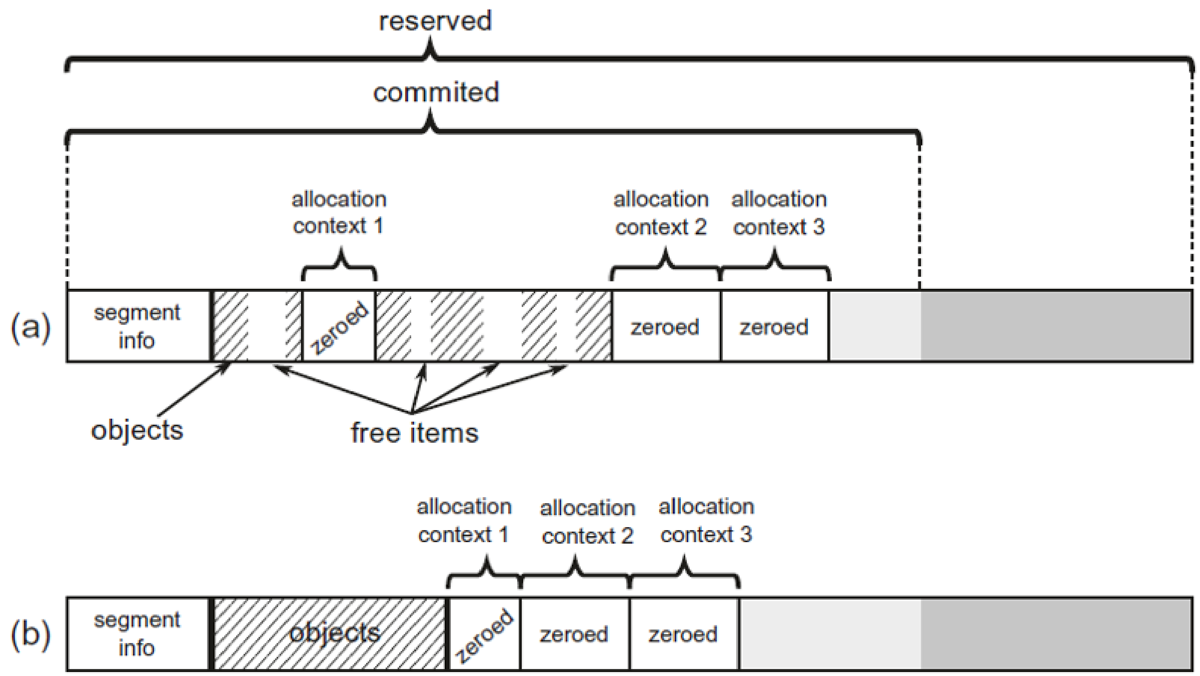
"Smart" bump pointer allocation reuses free space!



"Smart" bump pointer allocation reuses free space!

(*) we will return to that!


reserved
commited
allocation context 1
allocation context 2
allocation context 3
segment info
zeroed
zeroed
zeroed
gen2
gen1
gen0



Compacting still makes sense - from time to time!

Free-list allocator

Free-list allocator

- searching through a free items list to find a gap big enough

Free-list allocator

- searching through a free items list to find a gap big enough
    - *best-fit* - the smallest block fitting (little leftovers)

Free-list allocator

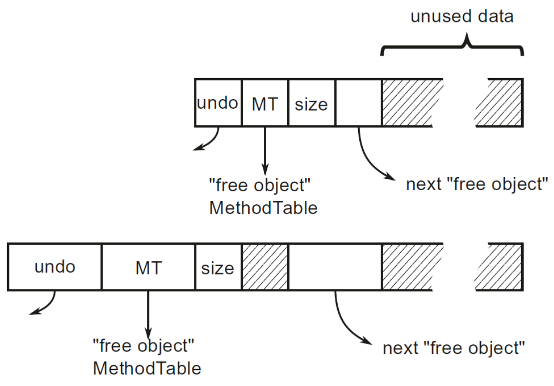
- searching through a free items list to find a gap big enough
  - *best-fit* - the smallest block fitting (little leftovers)
  - *first-fit* - the first block fitting (faster but leftovers)

Free-list allocator

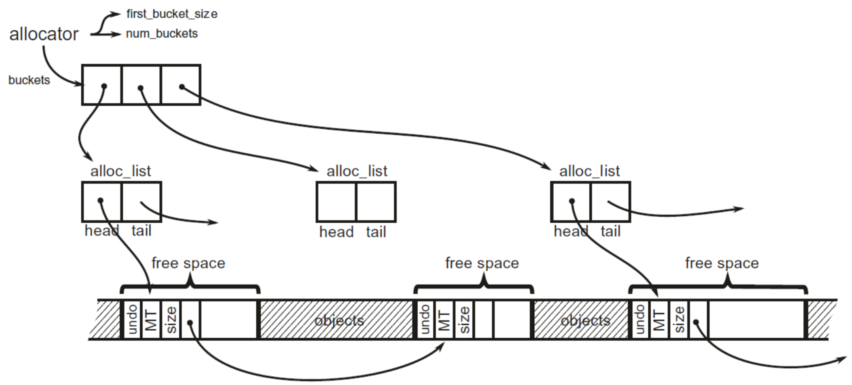
- searching through a free items list to find a gap big enough
    - *best-fit* - the smallest block fitting (little leftovers)
    - *first-fit* - the first block fitting (faster but leftovers)
    - buckets - first-fit into of buckets of various size ranges

Free-list allocator

- searching through a free items list to find a gap big enough
  - *best-fit* - the smallest block fitting (little leftovers)
  - *first-fit* - the first block fitting (faster but leftovers)
  - buckets - first-fit into of buckets of various size ranges
- in .NET free list is (partially) stored on the heap itself
  - "free object" with a predefined MT
  - keeps size as an array
  - keeps address of the next "free object" (single-linked list)
  - keeps special "undo" address
  - for sizes >= 2*minimum object size



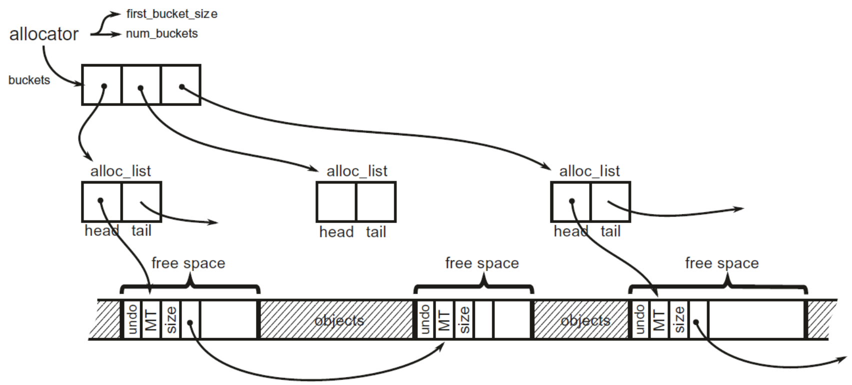
Free-list allocator - Buckets as metadata



Region	First bucket size	Number of buckets
Generation 0	Int.Max	1
Generation 1	Int.Max	1
Generation 2	256 B (64-bit) 128 B (32-bit)	12 12
LOH	64 kB	7

For gen 0 and 1 - free item is being discarded (becomes unusable fragmentation) **if it fails to fit the required size**.

Free-list allocator - Buckets as metadata



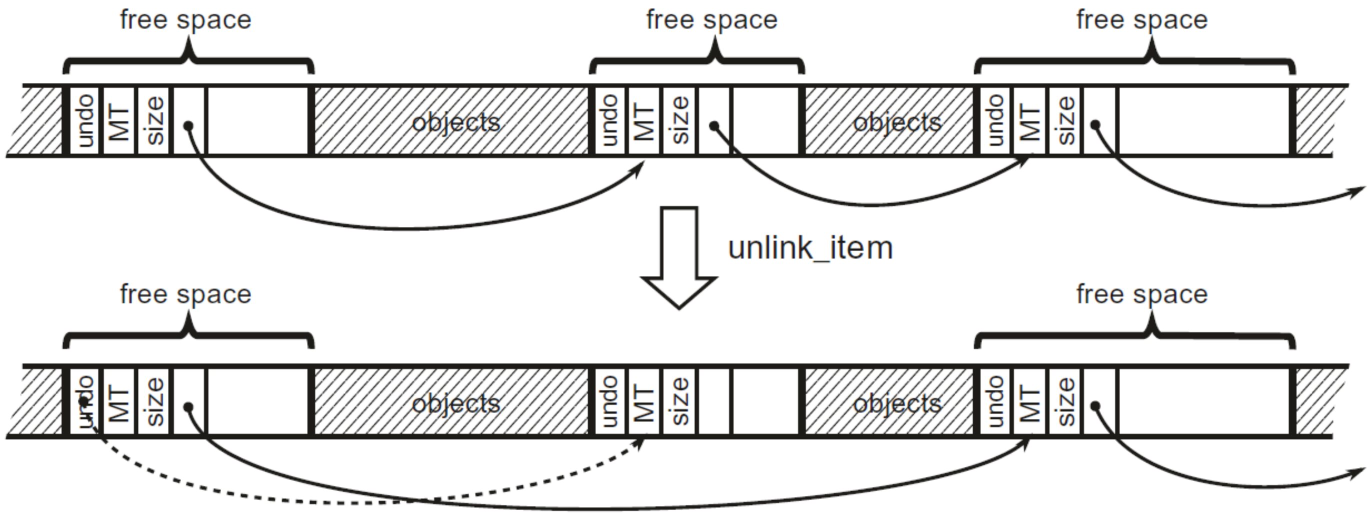
Region	First bucket size	Number of buckets
Generation 0	Int.Max	1
Generation 1	Int.Max	1
Generation 2	256 B (64-bit) 128 B (32-bit)	12 12
LOH	64 kB	7

For gen 0 and 1 - free item is being discarded (becomes unusable fragmentation) **if it fails to fit the required size**.

Hola! Why we need gen 1 and 2 for free-list allocation?!

Free-list allocator

Undo is used to... undo planned free-items usage (for compacting) if sweeping has been decided. In other words - to revert typical "unlink" operation on single-linked list element.



Allocation... creating a new object

Creating a new object

```
var obj = new SomeClass();
```

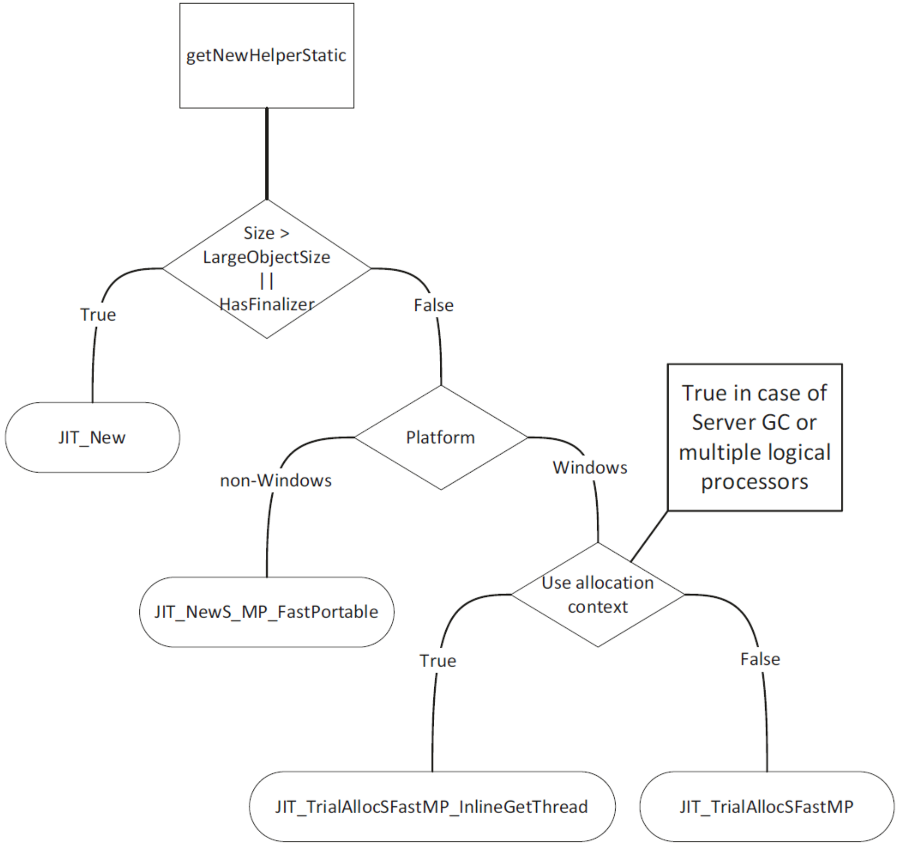
becomes

```
newobj instance void SomeClass::.ctor()
```

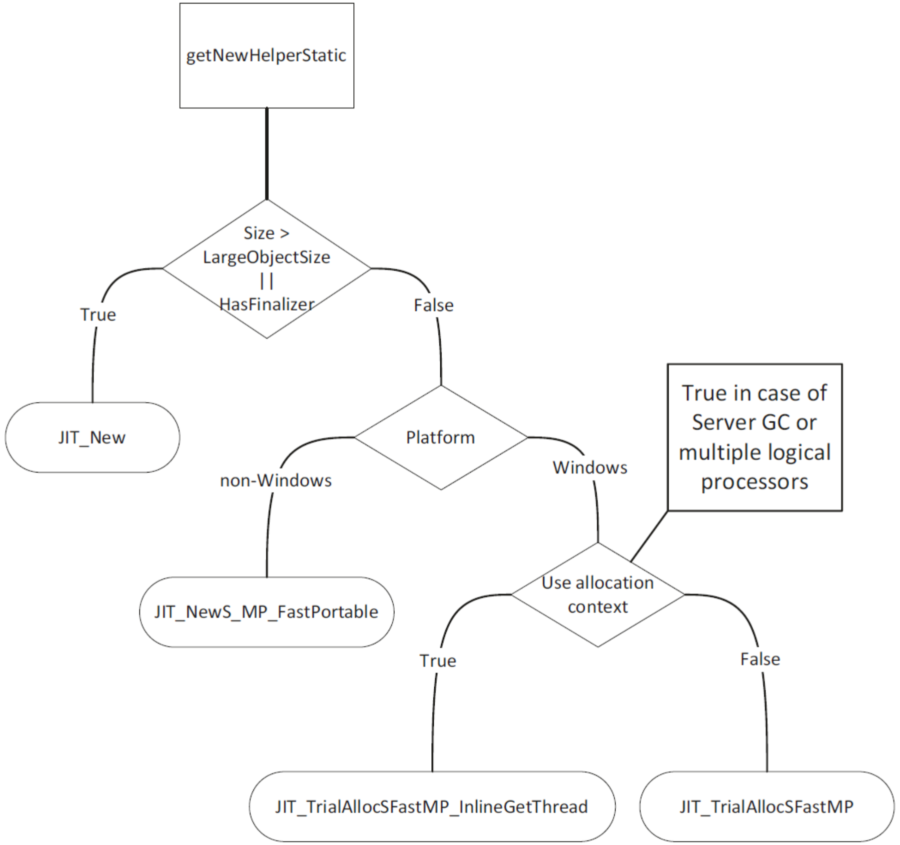
Question:

- who resets object's fields to defaults?
- who decides where to allocate (SOH/LOH)?

`newobj`'s JIT decision path



`newobj`'s JIT decision path



`InitJITHelpers1` initializes "fast helpers" in JIT, like **`CORINFO_HELP_NEWSFAST`** or **`CORINFO_HELP_NEWARR_1_VC`**. BTW, **`JIT_NewS_MP_FastPortable`** on non-Windows also uses allocation context.

Small Object Heap allocation

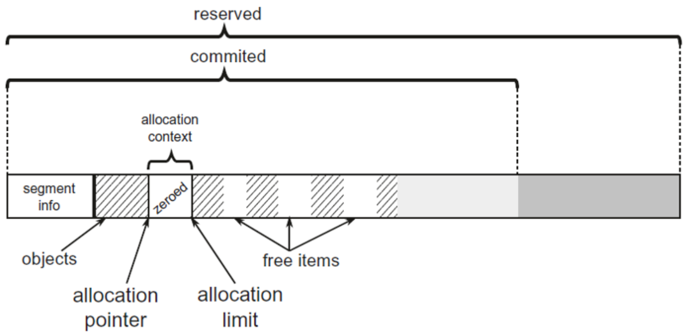
- mostly - **bump-pointer allocation** inside the current allocation context
  - `JIT_TrialAllocSFastMP_InlineGetThread`
- fallbacks to `JIT_NEW` in case of allocation context being full

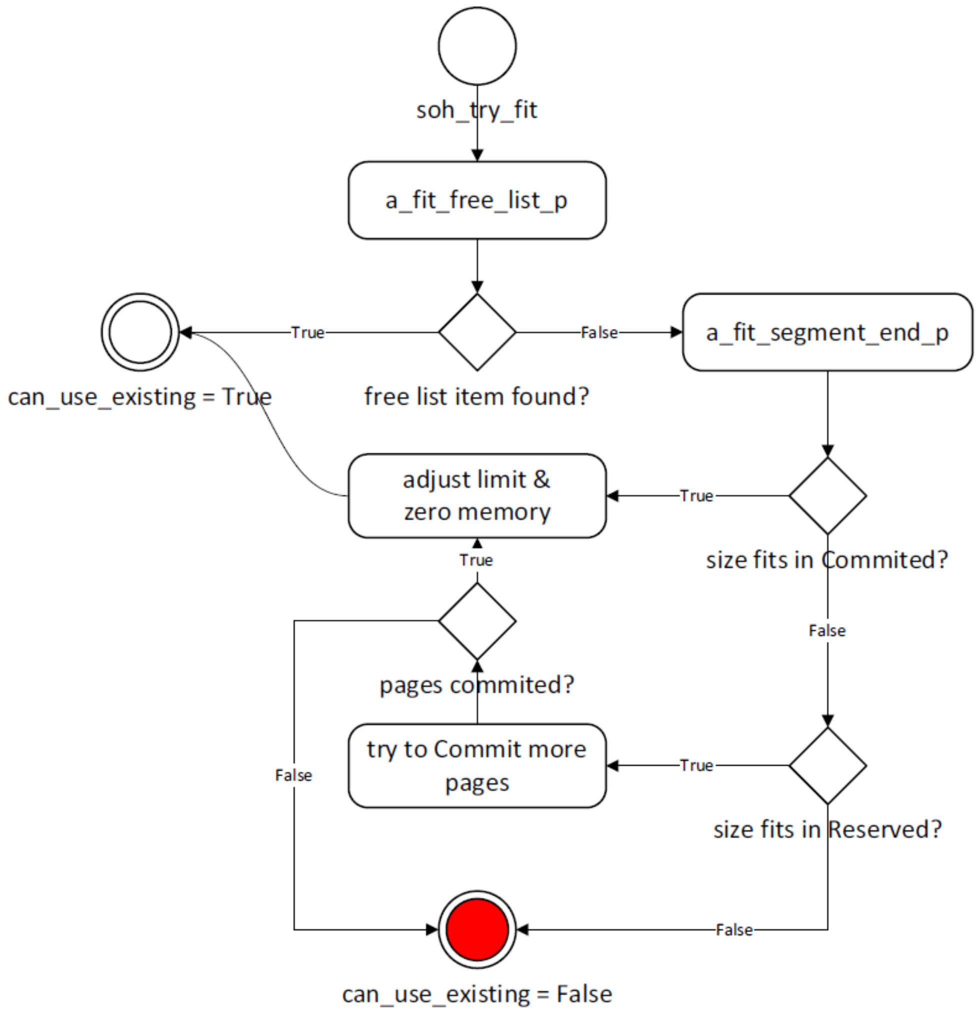
```
; As input, rcx contains MethodTable pointer
; As result, rax contains new object address

LEAF_ENTRY JIT_TrialAllocSFastMP_InlineGetThread, _TEXT
  ; Read object size into edx
  ; m_BaseSize is guaranteed to be a multiple of 8.
  mov edx, [rcx + OFFSET__MethodTable__m_BaseSize]

  ; Read Thread Local Storage address into r11
  INLINE_GETTHREAD r11

  ; Read alloc_limit into r10
  mov r10, [r11 + OFFSET__Thread__m_alloc_context__alloc_limit]

  ; Read alloct_ptr into rax
  mov rax, [r11 + OFFSET__Thread__m_alloc_context__alloc_ptr]

  add rdx, rax ; rdx = alloc_ptr + size
  cmp rdx, r10 ; is rdx smaller than alloc_limit
  ja AllocFailed

  ; Update alloc_ptr in TLS
  mov [r11 + OFFSET__Thread__m_alloc_context__alloc_ptr], rdx

  ; Store MT under alloc_ptr address (constituting new object)
  mov [rax], rcx
  ret

AllocFailed:
  jmp JIT_NEW ; fast-path failed, jump to slow-path

LEAF_END JIT_TrialAllocSFastMP_InlineGetThread, _TEXT
```

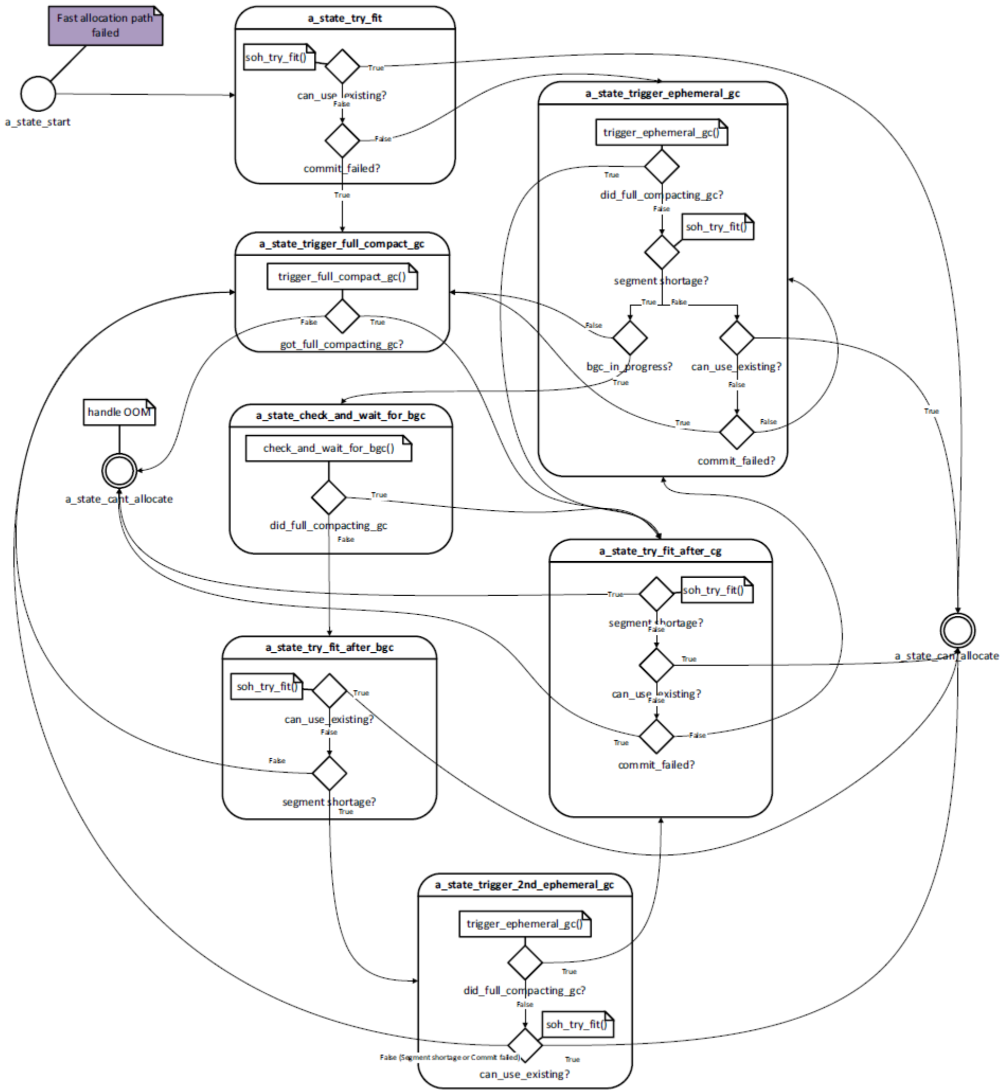
`JIT_NEW` helper

The same as used for objects with finalizer or in LOH.

- "slower" C++ bump-pointer allocator (because it is generic for both SOH/LOH)
- if fails, the whole story begins - the true "slow-path":
    - trying to use existing, unused space in. It will:
        - Try to use free list to find a suitable free gap for a new allocation context - **free-list allocation of a new allocation context**
        - Try to adjust allocation limit inside already Commited memory
        - Try to Commit more memory from Reserved memory and adjust allocation limit inside.
    - If all above fails, GC will be triggered
    - If all above fails - `OutOfMemoryException` :(




soh_try_fit
a_fit_free_list_p
True
False
a_fit_segment_end_p
can_use_existing = True
free list item found?
adjust limit &
zero memory
True
size fits in Commited?
True
False
pages commited?
try to Commit more
pages
True
False
size fits in Reserved?
False
can_use_existing = False



Fast allocation path failed
a_state_start
a_state_try_fit
soh_try_fit()
can_use_existing?
commit_failed?
a_state_trigger_ephemeral_gc
trigger_ephemeral_gc()
did_full_compacting_gc?
soh_try_fit()
segment shortage?
bgc_in_progress?
can_use_existing?
commit_failed?
a_state_trigger_full_compact_gc
trigger_full_compact_gc()
got_full_compacting_gc?
handle OOM
a_state_cant_allocate
a_state_check_and_wait_for_bgc
check_and_wait_for_bgc()
did_full_compacting_gc
a_state_try_fit_after_cg
soh_try_fit()
segment shortage?
can_use_existing?
commit_failed?
a_state_can_allocate
a_state_try_fit_after_bgc
soh_try_fit()
can_use_existing?
segment shortage?
a_state_trigger_2nd_ephemeral_gc
trigger_ephemeral_gc()
did_full_compacting_gc?
soh_try_fit()
can_use_existing?
False (Segment shortage or Commit failed)

Large Object Heap allocation

- **free-list allocation** and simplified bump-pointer at the end of the segment
  - no use of allocation context
  - ... thus synchronization overhead
  - ... and memory zeroing overhead
- only "slow-path":
  - try to use free list to find a suitable free gap for an object
  - in each segment containing LOH:
    - try to adjust allocation limit inside already Committed memory,
    - try to Commit more memory from Reserved memory and adjust allocation limit inside
  - if all above fails, GC will be triggered.
  - If all above fails - **`OutOfMemoryException`** :(

Pinned Object Heap allocation

- new allocation API: [`T[] GC.AllocateArray<T> (int length, bool pinned = false)`]
- it adds `GC_ALLOC_FLAGS.GC_ALLOC_PINNED_OBJECT_HEAP` flag to `AllocateNewArray`
- in the end it calls `allocate_uoh_object` on `poh_generation` (#4)
- which is shared between LOH and POH

Allocation overhead - summary

- SOH - super-fast bump-pointer inside allocation context (AC) but…
  - fallback to free-list finding of new AC or extending commit/reserve segment
  - … which requires zeroing such a new AC
  - or the GC
- LOH & POH - dominated by zeroing cost (now, optional) and…
  - additionally synchronized
  - even more painful in LOH with the Concurrent GC - LOH allocations blocked for (part) of the time of the Concurrent Sweep
    - `"LOH Allocation Pause (due to background GC) > 200 Msec"` section in PerfView's `GCStats`
- `stackalloc` - only memory region zeroing cost (if not disabled 😍)

Allocations

"`AllocateObject` is calling in the end `Object* GCHeap::Alloc` (with flags like `GC_ALLOC_FINALIZE` or `GC_ALLOC_LARGE_OBJECT_HEAP`), calling `allocate_uoh_object` for UOH (User Old Heap) - LOH & POH. Or calling `gc_heap::allocate` for SOH.

If the current allocation context is not enough, it calls `gc_heap::allocate_more_space` and then `gc_heap::try_allocate_more_space` internally."